Brian Malloy
Steffen Staab
Mark van den Brand (Eds.)

LNCS 6563

Software Language Engineering

Third International Conference, SLE 2010
Eindhoven, The Netherlands, October 2010
Revised Selected Papers

Springer

Brian Malloy Steffen Staab
Mark van den Brand (Eds.)

Software Language Engineering

Third International Conference, SLE 2010
Eindhoven, The Netherlands, October 12-13, 2010
Revised Selected Papers

Springer

Volume Editors

Brian Malloy
Clemson University
Computer Science Dept.
Clemson, SC 29634, USA
E-mail: malloy@cs.clemson.edu

Steffen Staab
University of Koblenz
Institute for Computer Science
P.O.Box 201602, 56016 Koblenz, Germany
E-mail: staab@uni-koblenz.de

Mark van den Brand
Eindhoven University of Technology
Mathematics and Computer Science Dept.
Den Dolech 2, 5612 AZ Eindhoven, The Netherlands
E-mail: m.g.j.v.d.brand@tue.nl

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-19439-9 e-ISBN 978-3-642-19440-5
DOI 10.1007/978-3-642-19440-5
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011921519

CR Subject Classification (1998): D.2, D.3, F.3, K.6.3

LNCS Sublibrary: SL 2 – Programming and Software Engineering

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

We are pleased to present the proceedings of the third international conference on Software Language Engineering (SLE 2010). The conference was held in Eindhoven, the Netherlands during October 12–13, 2010. It was co-located with the ninth international conference on Generative Programming and Component Engineering (GPCE) and the workshop on Feature-Oriented Software Development (FOSD). An important goal of SLE is to integrate the different sub-communities of the software-language-engineering community to foster cross-fertilisation and strengthen research overall. The doctoral symposium at SLE 2010 contributed towards this goal by providing a forum for both early and late-stage PhD students, who presented their research and got detailed feedback and advice from other researchers.

The SLE conference series is devoted to a wide range of topics related to artificial languages in software engineering. SLE is an international research forum that brings together researchers and practitioners from both industry and academia to expand the frontiers of software language engineering. SLE's foremost mission is to encourage and organize communication between communities that have traditionally looked at software languages from different, more specialized, and yet complementary perspectives. SLE emphasizes the fundamental notion of languages as opposed to any realization in specific technical spaces. In this context, the term "software language" comprises all sorts of artificial languages used in software development including general-purpose programming languages, domain-specific languages, modeling and meta-modeling languages, data models, and ontologies. Software language engineering is the application of a systematic, disciplined, quantifiable approach to the development, use, and maintenance of these languages. The SLE conference is concerned with all phases of the lifecycle of software languages; these include the design, implementation, documentation, testing, deployment, evolution, recovery, and retirement of languages. Of special interest are tools, techniques, methods, and formalisms that support these activities. In particular, tools are often based on, or automatically generated from, a formal description of the language. Hence, the treatment of language descriptions as software artefacts, akin to programs, is of particular interest – while noting the special status of language descriptions, and the tailored engineering principles and methods for modularization, refactoring, refinement, composition, versioning, co-evolution, and analysis that can be applied to them.

The response to the call for papers for SLE 2010 was very enthusiastic. We received 79 full submissions from 108 initial abstract submissions. From these submissions, the Program Committee (PC) selected 24 papers: 17 full papers, five short papers, and two tool demonstration papers, resulting in an acceptance rate of 32%. To ensure the quality of the accepted papers, each submitted paper was reviewed by at least three PC members. Each paper was discussed in detail during the electronic PC meeting. A summary of this discussion was prepared by members of the PC and provided to the authors along with the reviews.

SLE 2010 would not have been possible without the significant contributions of many individuals and organizations. We are grateful to the organizers of GPCE 2010 and FOSD 2010 for their close collaboration and management of many of the logistics. This will allow us to offer SLE participants the opportunity to take part in three high-quality research events in the domain of software engineering. We also wish to thank our supporters, ACM, ASML, Jacquard, IBM, and NWO.

The SLE 2010 Organizing Committee, the Local Chairs, and the SLE Steering Committee provided invaluable assistance and guidance. We are also grateful to the PC members and the additional reviewers for their dedication in reviewing the large number of submissions. We also thank the authors for their efforts in writing and then revising their papers, and we thank Springer for publishing the papers and the proceedings.

December 2010

Brian Malloy
Steffen Staab
Mark van den Brand

Conference Organization

General Chair

Mark van den Brand

Programme Chairs

Brian Malloy
Steffen Staab

Programme Committee

Uwe Assmann
Colin Atkinson
Sonia Bergamaschi
John Boyland
Jordi Cabot
Silvana Castano
Anthony Cleve
Michael Collard
Charles Consel
James Cordy
Stephen A. Edwards
Gregor Engels
Jean-Marie Favre
Aldo Gangemi
Chiara Ghidini
Jeff, Gray
Peter Haase
Gorel Hedin
Geert-Jan Houben
Adrian Johnstone
Nicholas Kraft
Ivan Kurtev
Julia Lawall
Marjan Mernik
Ralf Moeller
Pierre-Etienne Moreau
Peter Mosses
Istvan Nagy
Daniel Oberle
Richard Paige
Jeff Z. Pan
Bijan Parsia
James Power
Alexander Serebrenik
Fernando Silva Parreiras
Tony Sloane
Eleni Stroulia
York Sure
Gabriele Taentzer
Eric Van Wyk
Jurgen Vinju
Eelco Visser
Steffen Zschaler

External Reviewers

Andias Wira-Alam
Andreas Classen
Antoine Reilles
Antonio Sala
Arjan van der Meer
Arnaud Hubaux
Brandon Bonds
Christian Gerth
Christian Kästner
Christoff Bürger
Colin Atkinson
Damien Cassou
Dimitrios Kolovos
Eduardo Rivera
Emilie Balland
Fabian Christ
Florian Mantz
Gerd Gröner
Jeremy Pate
Jeroen Van den Bos
Laura Po
Lennart C. L. Kats
Loek Cleophas
Louis Rose
Maartje de Jonge
Marios Fokaefs
Markus Luckey
Mathieu Morey
Maurizio Vincini
Michael Decker
Michael Schmidt
Michel Reniers
Nicolas Loriant
Nophadol Jekjantuk
Oliver Hopt
Osmar Marchi dos Santos
Paul Brauner
Philipp Schaer
Sebastian Cech
Sebastian Richly
Serguei Roubtsov
Silvia Rota
Simone Roettger
Stefan Jurack
Stefano Montanelli
Suman Roychoudhury
Sven Karol
Ted Kaminski
Tijs van der Storm
Tobias Walter
Tomaz Lukman
Valentina Presutti
Xiaocheng Ge
Yu Sun
Yuan Ren
Yuting Zhao
Zef Hemel

Table of Contents

A Language for Software Variation Research*

Martin Erwig

School of EECS
Oregon State University

Variation occurs in many places in software engineering and takes quite different forms. Software can have different versions, and it can come in different configurations. Software can offer different sets of features, and it can appear in different stages of refactoring without any visible effect in functionality. Traditionally, all these forms of variation have used different representations. While this specialization might have some benefits by facilitating the tailoring to the specific needs of one form of variation, it has also some serious drawbacks. First, different representations prevent or complicate a potential integration of different forms of variation. For example, variation in functionality is currently only poorly supported in most versioning tools by branching. Second, it can be difficult to transfer research results achieved within one representation to other representations. Finally, different representations can lead to duplicated work and a balkanization of variation research efforts.

In this talk I describe the *choice calculus*, a formal representation for software variation that can serve as a common, underlying representation for variation research, playing a similar role that lambda calculus plays in programming language research. I will sketch the syntax and semantics of the choice calculus and present several applications.

At the core of the choice calculus are *choices*, which represent different alternatives that can be selected. Choices are annotated by names, which group choices into *dimensions*. Dimensions provide a structuring and scoping mechanism for choices. Moreover, each dimension introduces the number of alternatives each choice in it must have and tags for selecting those alternatives. The semantics of the choice calculus is defined via repeated elimination of dimensions and their associated choices through the selection of a tag defined by that dimension. The choice calculus obeys a rich set of laws that give rise to a number of normal forms and allow the flexible restructuring of variation representations to adjust to the needs of different applications.

Among the potential applications of the choice calculus are feature modeling, change pattern detection, property preservation, and the development of change IDEs. These are described in the long version of this abstract [1]; more technical details about the choice calculus can found in [2].

References

1. Erwig, M.: A Language for Software Variation. In: ACM SIGPLAN Conf. on Generative Programming and Component Engineering, pp. 3–12 (2010)
2. Erwig, M., Walkingshaw, E.: The Choice Calculus: A Representation for Software Variation. ACM Transactions on Software Engineering and Methodology (to appear, 2011)

* This work is supported by the Air Force Office of Scientific Research under the grant FA9550-09-1-0229 and by the National Science Foundation under the grant CCF-0917092.

Automated Selective Caching for Reference Attribute Grammars

Emma Söderberg and Görel Hedin

Department of Computer Science, Lund University, Sweden
{emma.soderberg,gorel.hedin}@cs.lth.se

Abstract. Reference attribute grammars (RAGs) can be used to express semantics as super-imposed graphs on top of abstract syntax trees (ASTs). A RAG-based AST can be used as the in-memory model providing semantic information for software language tools such as compilers, refactoring tools, and meta-modeling tools. RAG performance is based on dynamic attribute evaluation with caching. Caching all attributes gives optimal performance in the sense that each attribute is evaluated at most once. However, performance can be further improved by a selective caching strategy, avoiding caching overhead where it does not pay off. In this paper we present a profiling-based technique for automatically finding a good cache configuration. The technique has been evaluated on a generated Java compiler, compiling programs from the Jacks test suite and the DaCapo benchmark suite.

1 Introduction

Reference attribute grammars (RAGs) [11] provide a means for describing semantics as super-imposed graphs on top of an abstract syntax tree (AST) using *reference attributes*. Reference attributes are defined by functions and may have values referring to distant nodes in the AST. RAGs have been shown useful for the generation of many different software language tools, including Java compilers [31,9], Java extensions [13,14,22], domain-specific language tools [16,2], refactoring tools [24], and meta-modeling tools [7]. Furthermore, they are being used in an increasing number of meta-compilation systems [12,30,25,18].

RAG evaluation is based on a dynamic algorithm where attributes are evaluated on demand, and their values are cached (memoized) for obtaining optimal performance [15]. Caching all attributes gives optimal performance in the sense that each attribute is evaluated at most once. However, caching has a cost in both compilation time and memory consumption, and caching does not pay off in practice for all attributes. Performance can therefore be improved by *selective caching*, caching only a subset of all attributes, using a *cache configuration*. But determining a good cache configuration is not easy to do manually. It requires a good understanding of how the underlying attribute evaluator works, and a lot of experience is needed to understand how different input programs can affect the caching inside the generated language tool. Ideally, the language engineer should not need to worry about this, but let the system compute the configuration automatically.

In this paper we present a profiling-based approach for automatically computing a cache configuration. The approach has been evaluated experimentally on a generated compiler for Java [9], implemented using JastAdd [12], a meta-compilation system based on RAGs. We have profiled this compiler using programs from Jacks (a compiler test suite for Java) [28] and DaCapo (a benchmark suite for Java) [4]. Our evaluation shows that it is possible to obtain an average compilation speed-up of 20% while only using the profiling results from one application with a fairly low attribute coverage of 67%. The contributions of this paper include the following:

- A profiling-based approach for automatic selective caching of RAGs.
- An implementation of the approach integrated with the JastAdd meta-compilation system.
- An evaluation of the approach, comparing it both to full caching (caching all attributes) and to an expert cache configuration (produced manually).

The rest of this paper is structured as follows. Section 2 gives background on reference attribute grammars and their evaluation, explaining the JastAdd caching scheme in particular. Section 3 introduces the concept of an AIG, an attribute instance graph with call information, used as the basis for the caching analysis. Section 4 introduces our technique for computing a cache configuration. Section 5 presents an experimental evaluation of the approach. Section 6 discusses related work, and Section 7 gives a conclusion along with future work.

2 Reference Attribute Grammars

Reference Attribute Grammars (RAGs) [11], extend Knuth-style attribute grammars [19] by allowing attributes to be references to nodes in the abstract syntax tree (AST). This is a powerful notion because it allows the nodes in an AST to be connected into the graphs needed for compilation. For example, reference attributes can be used to build a type graph connecting subclasses to superclasses [8], or a control-flow graph between statements in a method [20]. Similar extensions to attribute grammars include Poetzsch-Heffter's occurrence algebras [21] and Boyland's remote attribute grammars [6].

In attribute grammars, attributes are defined by equations in such a way that for any attribute instance in any possible AST, there is exactly one equation defining its value. The equations can be viewed as side-effect-free functions which make use of constants and of other attribute values.

In RAGs, it is allowed for an equation to define an attribute by following a reference attribute and accessing its attributes. For example, suppose node n_1 has attributes a and b, where b is a reference to a node n_2, and that n_2 has an attribute c. Then a can be defined by an equation as follows:

$$a = b.c$$

For Knuth-style attribute grammars, dependencies are restricted to attributes in parents or children. But the use of references gives rise to non-local dependencies, i.e., dependencies that are independent of the AST hierarchy: a will be dependent on b and c, where the dependency on b is local, but the dependency on c is non-local: the node n_2 referred to by b could be anywhere in the AST. The resulting attribute dependency graph cannot be computed without actually evaluating the reference attributes, and it is therefore difficult to statically precompute evaluation orders based on the grammar alone. Instead, evaluation of RAGs is done using a simple but general dynamic evaluation approach, originally developed for Knuth-style attribute grammars, see [15]. In this approach, attribute access is replaced by a recursive call which evaluates the equation defining the attribute. To speed up the evaluation, the evaluation results can be cached (memoized) in order to avoid evaluating the equation for a given attribute instance more than once. Caching all attributes results in optimal evaluation in that each attribute instance is evaluated at most once. Because this evaluation scheme does not require any pre-computed analysis of the attribute dependencies, it works also in the presence of reference attributes.

Caching is necessary to get practical compiler performance for other than the tiniest input programs. But caching also implies an overhead. Compared to caching all attributes, *selective caching* may improve performance, both concerning time and memory.

2.1 The JastAdd Caching Scheme

In JastAdd, the dynamic evaluation scheme is implemented in Java, making use of an object-oriented class hierarchy to represent the abstract grammar. Attributes are implemented by method declarations, equations by method implementations, and attribute accesses by method calls. Caching is decided per attribute declaration, and cached attribute values are stored in the AST nodes using two Java fields: one field is a flag keeping track of if the value has been cached yet, and another field holds the value. Figure 1 shows the implementation of the equation $a = b.c$, both in a non-cached and a cached version. It is assumed that a is of type A. The example shows the implementation of a so called *synthesized attribute*, i.e., an attribute defined by an equation in the node itself. The implementation of a so called *inherited attribute*, defined by an equation in an ancestor node, is slightly more involved, but uses the same technique for caching. The implementation in Figure 1 is also simplified as compared to the actual implementation in JastAdd which takes into account, for example, circularity checking. These differences are, however, irrelevant to the caching problem.

This caching scheme gives a low overhead for attribute accesses: a simple test on a flag. On the other hand, the caching pays off only after at least one attribute instance has been accessed at least twice. Depending on the cost of the value computation, more accesses than that might be needed for the scheme to pay off.

JastAdd allows attributes to have parameters. A *parameterized attribute* has an unbounded number of values, one for each possible combination of parameter values. To cache accessed values, the flag and value fields are replaced by a map where the actual parameter combination is looked up, and the cached values are stored. This is a substantially more costly caching scheme, both for accessing attributes and for updating the cache, and more accesses per parameter combination will be needed to make it pay off.

Non-cached version:

```
class Node {
  A a() {
    return b().c();
  }
}
```

Cached version:

```
class Node {
  boolean a_cached = false;
  A a_value;
  A a() {
    if (! a_cached) {
      a_value = b().c();
      a_cached = true;
    }
    return a_value;
  }
}
```

Fig. 1. Caching scheme for non-parameterized attributes

3 Attribute Instance Graphs

In order to decide which attributes that may pay off to cache, we build a graph that captures the attribute dependencies in an AST. This graph can be built by instrumenting the compiler to record all attribute accesses during a compilation. By analyzing such graphs for representative input programs, we would like to identify a number of attributes that are likely to improve the compilation performance if left uncached. We define the *attribute instance graph* (AIG) to be a directed graph with one vertex per attribute instance in the AST. The AIG has an edge (a_1, a_2) if, during the evaluation of a_1, there is a direct call to a_2, i.e., indirect calls via other attributes do not give rise to edges. Each edge is labeled with a *call count* that represents the number of calls. This count will usually be 1, but in an equation like $c = d + d$, the count on the edge (c, d) will be 2, since d is called twice to compute c.

The main program of the compiler is modeled by an artificial vertex `main`, with edges to all the attribute instances it calls. This may be many or few calls, depending on how the main program is written.

To handle parameterized attributes, we represent each accessed combination of parameter values for an attribute instance by a vertex. For example, the evaluation of the equation $d = e(3) + e(4) + e(4)$ will give rise to two vertices for e, one for $e(3)$ and one for $e(4)$. The edges are, as before, labeled by the call counts, so the edge $(d, e(3))$ is labeled by 1, and the edge $(d, e(4))$ by 2, since it is called twice. Figure 2 shows an example AIG for the following equations:

$$\begin{aligned} a &= b.c \\ c &= d + d \\ d &= e(3) + e(4) + e(4) \end{aligned}$$

and where it is assumed that a is called once from the main program.

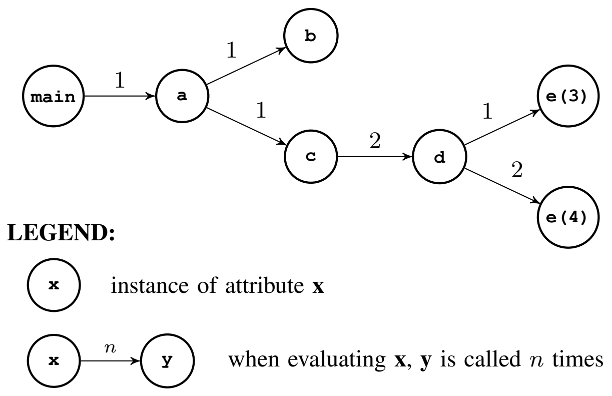


Fig. 2. Example AIG

3.1 An Example Grammar

Figure 3 shows parts of a typical JastAdd grammar for name and type analysis. The abstract grammar rules correspond to a class hierarchy. For example, `Use` (representing a use of an identifier) is a subclass of `Expr`. The first attribution rule:

```
syn Type Expr.type();
```

declares a synthesized attribute of type `Type`, declared in `Expr` and of the name `type`. All nodes of class `Expr` and its subclasses will have an instance of this attribute.

```
abstract Expr;
Use : Expr ::= ...;
Literal : Expr ::= ...;
AddExpr : Expr ::=
    e1:Expr e2:Expr;

Decl ::= Type ... ;

abstract Type;
Integer : Type;
Unknown : Type;

...
```

```
syn Type Expr.type();
syn Type Decl.type() = ...;

eq Literal.type() =
  stdTypes().integer();
eq Use.type() = decl().type();
eq AddExpr.type() =
  (left.type().sameAs(right.type()) ?
  left.type() : stdTypes.unknown();

syn Decl Use.decl() = lookup(...);
inh Decl Use.lookup(String name);
inh Type Expr.stdTypes();

syn boolean Type.sameAs(Type t) = ...;
...
```

Fig. 3. Example JastAdd attribute grammar

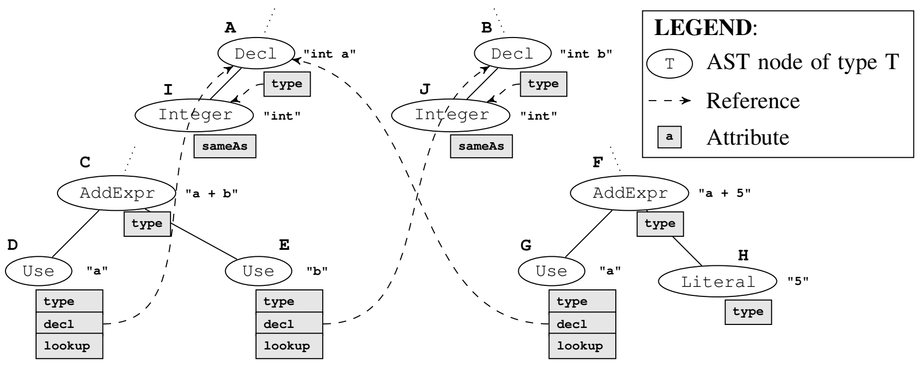


Fig. 4. An example attributed AST

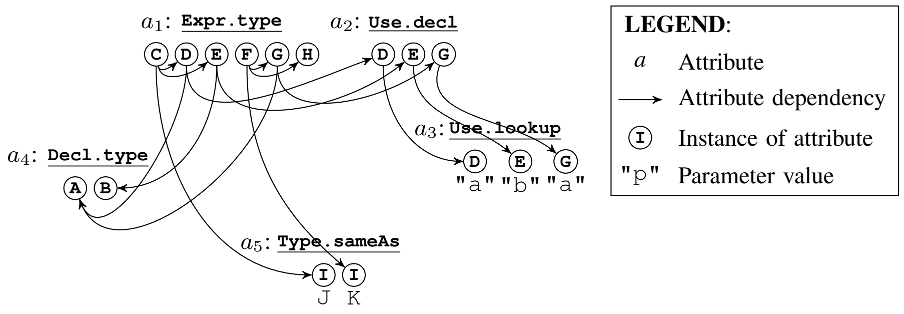


Fig. 5. Parts of the AIG for the example

Different equations are given for it in the different subclasses of `Expr`. For example, the equation

```
eq Use.type() = decl().type();
```

says that for a `Use` node, the value of `type` is defined to be `decl().type()`. The attribute `decl()` is another attribute in the `Use` node, referring to the appropriate declaration node, possibly far away from the `Use` node in the AST. The `decl()` attribute is in turn defined using a parameterized attribute `lookup`, also in the `Use` node. The `lookup` attribute is an inherited attribute, and the equation for it is in an ancestor node of the `Use` node (not shown in the grammar). For more information on name and type analysis in RAGs, see [8].

Figure 4 shows parts of an attributed AST for the grammar in Figure 3. The example program contains two declarations: `"int a"` and `"int b"`, and two add expressions: `"a + b"` and `"a + 5"`. For the `decl` attributes of `Use` nodes, the reference values are shown as arrows pointing to the appropriate `Decl` node. Similarly, the `type` attributes of `Decl` nodes have arrows pointing to the appropriate `Type` node. The nodes have been labeled `A`, `B`, and so on, for future reference.

Figure 5 shows parts of the AIG for this example. In the AIG we have grouped together all instances of a particular attribute declaration, and labeled each attribute instance with the node to which it belongs. For instance, since the node `D` has the three attributes (`type`, `decl`, and `lookup`), there are three vertices labeled `D` in the AIG. For parameterized attribute instances, there is one vertex per actual parameter combination, and their values are shown under the vertex. For instance, the `sameAs` attribute for `I` is called with two different parameters: `J` and `K`, giving rise to two vertices. (`K` is a node representing integer literal types and is not shown in Figure 4.) All call counts in the AIG are 1 and have therefore been omitted.

4 Computing a Cache Configuration

Our goal is to automatically compute a good cache configuration for a RAG specification. A cache configuration is simply the set of attributes configured to be cached. Among the different attribute kinds, there are some that will always be cached, due to properties of the kind. For example, circular attributes [10], which may depend on themselves, and non-terminal attributes (NTAs) [29], which may have ASTs as values. There is no cache decision to make for these attributes, i.e., they are *unconfigurable*. We let PRE denote the set of unconfigurable attributes. Since the attributes in the PRE set are always cached, we exclude them from remaining definitions in this paper. We let ALL denote the remaining set of *configurable* attributes. This ALL set can further be divided into two disjoint sets PARAM and NONPARAM, for parameterized and non-parameterized attributes respectively. For the rest of this paper we will refer to configurable attributes when we write attributes.

As a basis for our computation, we do profiling runs of the compiler on a set of test programs, producing the AIG for each program. These runs are done with all attributes cached, allowing us to use reasonably large test programs, and making it easy to compute the AIG which reflects the theoretically optimal evaluation with each attribute instance evaluated at most once. We will refer to these test programs as the *profiling input* denoted by the set P. Further, a certain profiling input ($p \in$ P) will, depending on its structure, require that a certain number of attributes are evaluated. We call this set of attributes the USED$_p$ set. However, it cannot be assumed that a single profiling input uses all attributes. We define the set of unused attributes for a profiling input p as follows:

$$\text{UNUSED}_p = \text{ALL} \setminus \text{USED}_p \tag{1}$$

4.1 The ONE Set

The calls label on the edges in the AIG reflects the number of attribute calls in a fully cached configuration. To find out if a certain attribute is worth uncaching, we define extra_evals(a_i), i.e., the number of extra evaluations of the attribute instance a_i that will be done if the attribute a is not cached:

$$\text{extra\_evals}(a_i) = \begin{cases} \text{calls}(a_i) - 1, & \text{if } a \in \text{NONPARAM} \\ \sum_{c \in \text{params}(a_i)} (\text{calls}(c) - 1), & \text{if } a \in \text{PARAM} \end{cases} \tag{2}$$

where $\mathrm{params}(a_i)$ is the set of vertices in the AIG representing different parameter combinations for the parameterized attribute instance a_i. The number of extra evaluations is a measure of what is lost by not caching an attribute. The total number of extra evaluations for an attribute a is simply the sum of the extra evaluations of all its instances:

$$\text{extra\_evals}(a) = \sum_{a_i \in \mathrm{I}_{\text{called}}(a)} \text{extra\_evals}(a_i); \tag{3}$$

where $\mathrm{I}_{\text{called}}(a)$ is the set of attribute instances of a that are called at least once. Of particular interest is the set of attributes for which all instances are called at most once. These should be good candidates to leave uncached since they do not incur any extra evaluations for a certain profiling input (p). We call this set the ONE_p set, and for a profiling input p it is constructed as follows:

$$\text{ONE}_p = \{a \in \text{USED}_p | \text{extra\_evals}(a) = 0\} \tag{4}$$

The $\text{USED}_p \setminus \text{ONE}_p$ set contains the remaining attributes in the AIG, i.e., the attributes which may gain from being cached, depending on the cost of their evaluation.

4.2 Selecting a Good Profiling Input

To obtain a good cache configuration, it is desirable to use profiling input that is realistic in its attribute usage, and that has a high *attribute coverage*, i.e., as large a USED_p set as possible. We define the attribute coverage (in percent) for a profiling input, $p \in \mathrm{P}$, as follows:

$$\text{coverage}(p) = (|\text{USED}_p|/|\text{ALL}|) * 100 \tag{5}$$

Furthermore, for tools used in an interactive setting with continuous compilation of potentially erroneous input, it is important to also take incorrect programs into account. To help fulfill these demands, different profiling inputs can be combined. In particular, a compilation test suite may give high attribute coverage and test both correct and erroneous programs. But test suites might contain many small programs that do not use the attributes in a realistic way. In particular, attributes which most likely should be in the $\text{USED}_p - \text{ONE}_p$ set for an average application may end up in the ONE_p sets of the test suite programs because these are small. By combining the test suite with a large real program, better results may be obtained. Still, even with many applications and a full test suite, it may be hard to get full coverage. For example, there may be semantic checks connected to uncommon language constructs and, hence, attributes rarely used.

4.3 Choosing a Cache Configuration

In constructing a good cache configuration we want to consider the USED_p, UNUSED_p, ONE_p and ALL sets. From these sets we can experiment with two interesting configurations:

$$\text{ALLONE}_p = \text{ALL} \setminus \text{ONE}_p \tag{6}$$

$$\text{USEDONE}_p = \text{USED}_p \setminus \text{ONE}_p \tag{7}$$

Presumably, the first configuration, which includes the $\textsc{unused}_p$ set, will provide robustness for cases where the profiling input is insufficient, i.e., the $\textsc{used}_p$ set is too small. In contrast, the second configuration may provide better performance in that it uses less memory for cases where the profiling input is sufficient.

4.4 Combining Cache Configurations

In order to combine the results of several profiling inputs, for example, A, B and C in P, we need to consider each of the resulting sets $\textsc{used}_p$ and $\textsc{one}_p$. One attribute might be used in the compilation of program A but not in the compilation of program B. If an attribute is used in both B and C, it might belong to $\textsc{one}_B$, but not to $\textsc{one}_C$, and so on. We want to know which attributes that end up in a total $\textsc{one}_\mathrm{P}$ set for all profiling inputs ($p \in$ P), i.e., the attributes that are used by at least one profiling input, but that, if they are used by a particular profiling input, they are in its $\textsc{one}_p$ set. More precisely:

$$\textsc{one}_\mathrm{P} = \bigcup_{p \in \mathrm{P}} \textsc{used}_p \setminus \bigcup_{p \in \mathrm{P}} (\textsc{used}_p \setminus \textsc{one}_p) \tag{8}$$

These attributes should be good candidates to be left uncached. By including or excluding the $\textsc{unused}_\mathrm{P}$ set, we can now construct the following combined cache configuration, for a profiling input set P, in analogy to Definition 6 and Definition 7:

$$\textsc{AllOne}_\mathrm{P} = \textsc{all} \setminus \textsc{one}_\mathrm{P} \tag{9}$$

$$\textsc{UsedOne}_\mathrm{P} = \textsc{used}_\mathrm{P} \setminus \textsc{one}_\mathrm{P} \tag{10}$$

5 Evaluation

To evaluate our approach we have applied it to the frontend of the Java compiler JastAddJ [9]. This compiler is specified with RAGs using the JastAdd system. We have profiled the compilation with one or several Java programs as profiling input, and used the resulting AIGs to compute different cache configurations. We have divided our evaluation into the following experiments:

Experiment A: The effects of no caching
Experiment B: The effects of profiling using a compiler test suite
Experiment C: The effects of profiling using a benchmark application
Experiment D: The effects of combining B and C

Throughout our experiments we use the results of caching all attributes and the results of using a manual configuration, composed by an an expert, for comparison.

5.1 Experimental Setup

All measurements were run on a high-performing computer with two Intel Xeon Quad Core @ 3.2 GHz processors, a bus speed of 1.6 GHz and 32 GB of primary memory. The operating system used was Mac OS X 10.6.2 and the Java version was Java 1.6.0._15.

The JastAddJ compiler. The frontend of the JastAddJ compiler (for Java version 1.4 and 1.5) has an $\textsc{all}$ set containing 740 attributes and a $\textsc{pre}$ set containing 47 unconfigurable attributes (14 are circular and 33 are non-terminal attributes). The compiler comes with a configuration MANUAL, with 281 attributes manually selected for caching by the compiler implementor, an expert on RAGs, making an effort to obtain as good compilation speed as possible. The compiler performs within a factor of three as compared to the standard javac compiler, which is good considering that it is generated from a specification. MANUAL is clearly an expert configuration, and it cannot be expected that a better one can be obtained manually.

Measuring of performance. The JastAddJ compiler is implemented in Java (generated from the RAG specification), so measuring its compilation speed comes down to measuring the speed of a Java program. This is notoriously difficult, due to dynamic class loading, just-in-time compilation and optimization, and automatic memory management [4]. To eliminate as many of these factors as possible, we use the multi-iteration approach suggested in [5]. We start by warming up the compiler with a number of non-measured compilations (5), thereby allowing class loading and optimization of all relevant compiler code to take place, in order to reach a steady state. Then we turn off the just-in-time compilation and run a couple of extra unmeasured compilations (2) to drain any JIT work queues. After that we run several (20) measured compilation runs for which we compute 95% confidence intervals. In addition to this, we start each measured run with a forced garbage collection (GC) in order to obtain as similar conditions as possible for each run. Memory usage is measured by checking of available memory in the Java heap after each forced GC call and after each compilation. The memory measurements are also given with a 95% confidence interval. We present a summary of these results in Figure 7, Figure 8, Figure 9 and Figure 10. All results have a confidence interval of less than $\pm 0.03\%$. These intervals have not been included in the figures since they would be barely visible with the resolution we need to use. A complete list of results are available on the web [27].

Profiling and test input. As a basis for profiling input, we use the Jacks test suite [28], the DaCapo benchmark suite [4,26] and a small hello world program. We use 4170 tests from the Jacks suite, checking frontend semantics, and the following applications from the DaCapo suite (lines of code (LOC)):

ANTLR: an LL(k) parser generator (ca 35 000 LOC).
Bloat: a program for optimization and analysis of Java bytecode (ca 41 000 LOC).
Chart: a program for plotting of graphs and rendering of PDF files (ca 12 000 LOC).
FOP: parses XSL-FO files and generates PDF files (ca 136 000 LOC).
HsqlDb: a database application (ca 138 000 LOC).
Jython: a Python interpreter (ca 76 000 LOC).
Lucene: a program for indexing and searching of large text corpuses (ca 87 000 LOC).
PMD: a Java bytecode analyzer for a range of source code problems (ca 55 000 LOC).
Xalan: a program for transformation of XML documents into HTML (ca 172 000 LOC).

In our experiments, we use different combinations of these applications and tests as profiling input. We will denote these profiling input sets as follows:

J: The Jacks test suite profiling input set
D: The DaCapo benchmarks profiling input set
"APP": The benchmark APP of the DaCapo benchmarks.
For example, ANTLR means the ANTLR benchmark.
HELLO: The hello world program

We combine these profiling input sets in various ways, for example, the profiling input set J+ANTLR means we combine the Jacks suite with the benchmark ANTLR. Finally, as test input for performance testing we use the benchmarks from the DaCapo suite and the hello world program.

Cache configurations. We want to compare the results of using the cache configurations defined in Section 4. In addition, the JastAddJ specification comes with a manual cache configuration (MANUAL) which we want to compare to. We also have the option to cache all attributes (ALL), or to cache no attributes (NONE):

MANUAL: This expert configuration is interesting to compare to, as it would be nice if we could obtain similar results with our automated methods.
ALL: The ALL configuration is interesting as it is easily obtainable and robust with respect to performance: there is no risk that a particular attribute will be evaluated very many times for a particular input program, and thereby degrade performance.
NONE: The least possible configuration is interesting as it provides a lower bound on the memory needed during evaluation. However, this configuration will in general be useless in practice, leading to compilation times that increase exponentially with program size.

From each profiling input set P, we compute USED_P, and ONE_P, and construct the configurations USEDONE_P and ALLONE_P (according to Definition 9 and 10):

USEDONE_P: This is an interesting cache configuration as it should give good performance by avoiding caching of unused attributes and attributes used only once by P. The obvious risk with this configuration is that other programs might use attributes unused by P, causing performance degradation. There is also a risk that the attributes in the ONE_P set may belong to another program's $\text{USED} \setminus \text{ONE}$ set, also causing a performance degradation. However, if attributes in ONE_P are only used once in a typical application, they are likely to be used only once in most applications.
ALLONE_P: This configuration is more robust than the USEDONE_P configuration in that also unused attributes are cached, which prevents severe performance degradation for those attributes.

Attribute coverage. Figure 6 gives an overview of the $\text{USED}_P \setminus \text{ONE}_P$, ONE_P and UNUSED_P sets for the profiling inputs from the DaCapo suite. The figure also includes the combined sets for DaCapo (D) and Jacks (J). Not surprisingly, Hello World has the lowest attribute coverage. Still, it covers as much as 29%. The high coverage is due to analysis of standard library classes needed to compile the program. The combined results for the DaCapo suite and two of its applications have better or the same coverage

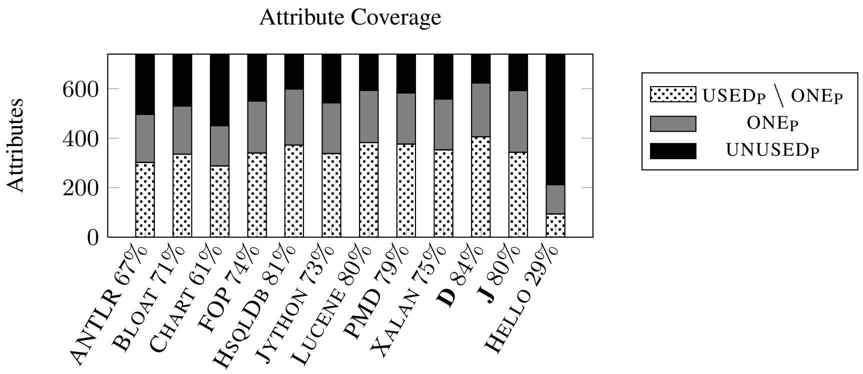


Fig. 6. Attribute coverage for the benchmarks in the DaCapo suite, the combined coverage for all the DaCapo applications (D), the combined coverage for the programs in the Jacks suite (J) and for a hello world program. The attribute coverages are given next to the names of the application/combination.

as the Jacks suite, i.e., the USED$_J$ set of Jacks *does not* enclose the USED$_D$ set of DaCapo neither does it have an empty UNUSED$_J$. These observations are interesting since they might indicate that additional tests could be added to Jacks. We can also note that the attribute coverage is not directly proportional to the size of an application, as shown by PMD and Lucene which both are smaller than Xalan and FOP in regard to LOC. This may not be surprising since the actual attribute coverage is related to the diversity of language constructs in an application rather than to the application size.

5.2 Experiment A: The Effects of No Caching

To compare the behavior of *no caching* with various other configurations, we profiled a simple Hello World program (HELLO) and then tested performance by compiling the same program using the configurations ALL, NONE, MANUAL, USEDONE$_{\text{HELLO}}$ and ALLONE$_{\text{HELLO}}$. The results are shown in Figure 7. It is clear from these results that

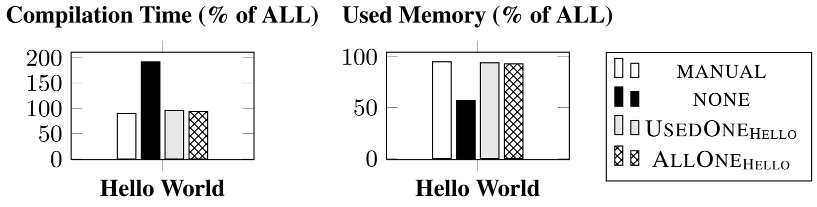


Fig. 7. Results from Experiment A: Compilation of Hello World using static configurations along with configurations obtained using Hello World (HELLO) as profiling input. The average compilation time / memory usage when compilating with the ALL configuration were 50.0 ms / 14.7 kb. The corresponding values for the NONE configuration were 95.9 ms / 8.4 kb.

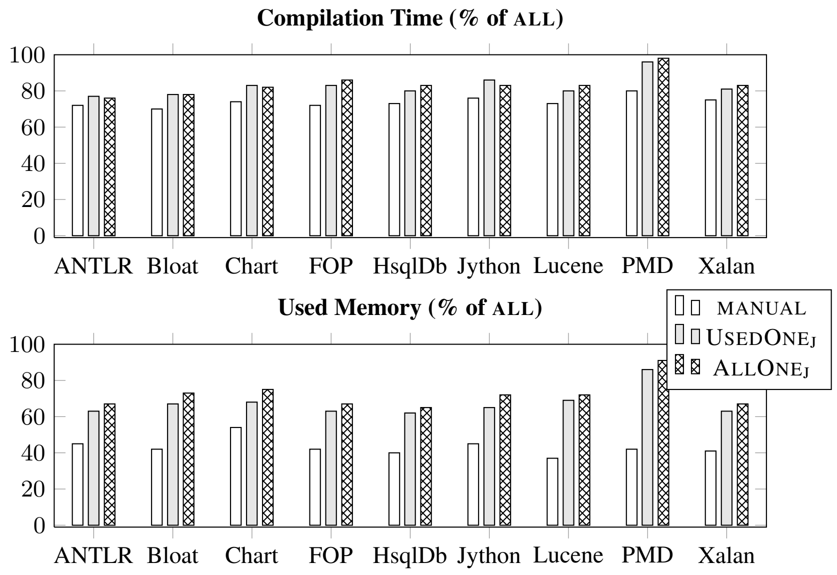


Fig. 8. Results from Experiment B: Compilation of DaCapo benchmarks using configurations from the Jacks suite. All results are given in relation to the compilation time and memory usage of the ALL configuration and results for MANUAL are included for comparison.

the minimal NONE configuration is not a good configuration, not even on this small test program. Even though it provides excellent memory usage, the compilation time is more than twice as slow as any of the other configurations. For a larger application the NONE configuration would be useless.

5.3 Experiment B: The Effects of Profiling Using a Compiler Test Suite

To show the effects of using a compiler test suite we profiled the compilation of the Jacks suite and obtained the two configurations USEDONEJ and ALLONEJ. We then measured performance when compiling the DaCapo benchmarks using these configurations. The results are shown in Figure 8 and are given as percent in relation to the compilation time and memory usage of the ALL configuration[1]. The results for the MANUAL configuration are included for comparison.

Clearly, the MANUAL configuration performs better with regard to both compilation time and memory usage, with an average compilation time / memory usage of 75% / 47% in relation to the ALL configuration. The average results for USEDONEJ is 83% / 67%. The ALLONEJ configuration has the same average compilation time 83% / 72%, but higher average memory usage. It is interesting to note that USEDONEJ is robust enough to handle the compilation of all the DaCapo benchmarks. So it seems that the

[1] The absolute average results for ALL are the following: Antlr (1.462s/0.270Gb), Bloat (1.995s/0.339Gb), Chart (0.928s/0.177Gb), FOP (8.328s/1.362Gb), HsqlDb (6.054s/1.160Gb), Jython (3.257s/0.611Gb), Lucene (4.893s/0.930Gb), PMD (3.921s/0.691Gb), Xalan (6.606s/1.141Gb).

Jacks test suite has a sufficiently large coverage, i.e., we can use the USEDONE$_J$ configuration rather than the ALLONE$_J$ configuration. In doing so, we can use less memory with the same performance and robustness.

5.4 Experiment C: The Effects of Profiling Using a Benchmark Program

To show the effects of using benchmarks we profiled using each of the DaCapo benchmarks obtaining the USEDONE and ALLONE configurations for each benchmark. We also combined the profiling results for all the benchmarks to create the combined configurations USEDONE$_D$ and ALLONE$_D$. We then measured performance when compiling the DaCapo benchmarks using these configurations. A selected set of the results are shown in Figure 9, including the combined results and the best USEDONE and ALLONE configurations from the individual benchmarks. All results are given as percent in relation to the compilation time and memory usage of the ALL configuration. The results for the MANUAL set are also included for comparison. Note that not all results are shown in Figure 9. Two of the excluded configurations USEDONE$_{ANTLR}$ and USEDONE$_{CHART}$ performed worse than full caching (ALL). These results validate the concern that the USEDONE$_p$ configuration would have robustness problems for insufficient profiling input. In this case, neither ANTLR nor Chart were sufficient as profiling

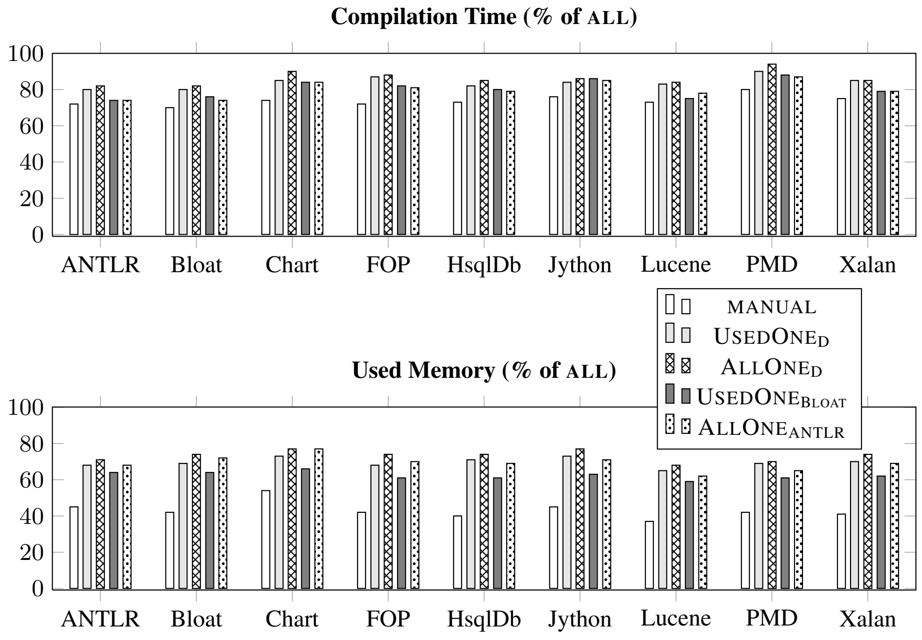


Fig. 9. Results from Experiment C: Compilation of DaCapo benchmarks using configurations from the DaCapo benchmarks. All results are shown as compilation time and memory usage as percent of the results for the ALL configuration and results for MANUAL are included for comparison.

inputs on their own. We can note that these two applications have the least coverage among the applications from the DaCapo suite (67% for ANTLR and 61% for Chart).

The USEDONE configurations. The USEDONE configurations for ANTLR and Chart perform worse than the ALL configuration for several of the applications in the DaCapo benchmarks: FOP, Lucene and PMD. The remaining USEDONE configurations can be sorted with regard to percent of compilation time, calculated as the geometric mean of the DaCapo benchmark program compilation times (each in relation to the ALL configuration), as follows:

80%: $\text{USEDONE}_{\text{BLOAT}}$ (mem. 62%)
82%: $\text{USEDONE}_{\text{FOP}}$ (mem. 63%), $\text{USEDONE}_{\text{XALAN}}$ (mem.66%)
83%: $\text{USEDONE}_{\text{HSQLDB}}$ (mem. 68%), $\text{USEDONE}_{\text{JYTHON}}$ (mem. 65%),
$\text{USEDONE}_{\text{LUCENE}}$ (mem. 68%), $\text{USEDONE}_{\text{PMD}}$ (mem. 66%)
84%: $\text{USEDONE}_{\text{D}}$ (mem. 70%)

These results indicate that a certain coverage is needed in order to obtain a robust USEDONE configuration. It is also interesting to note that the combined $\text{USEDONE}_{\text{D}}$ configuration for DaCapo performs the worst (except for the non-robust configurations). One possible explanation to this performance might be that some attributes ending up in the $\text{USEDONE}_{\text{D}}$ set might be used rarely or not at all in several compilations. Still, these attributes are cached which leads to more memory usage.

The ALLONE configurations. The ALLONE configurations generally perform worse than the USEDONE configurations. This result might be due to the fact that these configurations include unused attributes for robustness. However, this strategy for robustness pays off in that all ALLONE configurations become robust, i.e., they compile all the DaCapo benchmarks faster than the ALL configuration. The ALLONE configurations can be sorted as follows, with regard to percent of compilation time:

80%: $\text{ALLONE}_{\text{ANTLR}}$ (mem. 69%)
82%: $\text{ALLONE}_{\text{BLOAT}}$ (mem. 69%), $\text{ALLONE}_{\text{FOP}}$ (mem. 69%)
83%: $\text{ALLONE}_{\text{CHART}}$ (mem. 71%)
84%: $\text{ALLONE}_{\text{HSQLDB}}$ (mem. 73%), $\text{ALLONE}_{\text{JYTHON}}$ (mem. 70%),
$\text{ALLONE}_{\text{XALAN}}$ (mem. 70%)
85%: $\text{ALLONE}_{\text{LUCENE}}$ (mem. 73%), $\text{ALLONE}_{\text{PMD}}$ (mem. 71%)
86%: ALLONE_{D} (mem. 73%)

These results indicate that a profiled application does not necessarily need to be large, or have the best coverage, for the resulting configuration to provide good performance. The best individual ALLONE configuration is obtained from profiling ANTLR which is remarkable since ANTLR has the next lowest attribute coverage, while the combined ALLONE_{D} configuration for DaCapo performs the worst on average. This result might be due to the fact that the combined configuration caches attributes that might be in the ONE set of an individual application. This fact is also true for several of the individual configurations but apparently the complete combination takes the edge off the configuration.

5.5 Experiment D: The Effects of Combining B and C

To show the effects of profiling using a compiler test suite (B) together with profiling a benchmark program (C) we combine the cache configurations from experiment B and C. We then measured performance when compiling the DaCapo benchmarks using these configurations. A selected set of the results are shown in Figure 10, including the fully combined results and the best UsedOne and AllOne configurations, obtained from combining configurations from ANTLR and Jacks. We can sort the UsedOne configurations, with regard to their percent of compilation time, as follows:

81%: UsedOne$_{\text{J+ANTLR}}$ (mem. 64%), UsedOne$_{\text{J+Bloat}}$ (mem. 67%), UsedOne$_{\text{J+Chart}}$ (mem. 65%)
82%: UsedOne$_{\text{J+FOP}}$ (mem. 68%), UsedOne$_{\text{J+Xalan}}$ (mem. 69%)
83%: UsedOne$_{\text{J+Jython}}$ (mem. 69%)
84%: UsedOne$_{\text{J+HsqlDb}}$ (mem. 70%), UsedOne$_{\text{J+Lucene}}$ (mem. 70%), UsedOne$_{\text{J+PMD}}$ (mem. 70%)
85%: UsedOne$_{\text{J+D}}$ (mem. 71%)

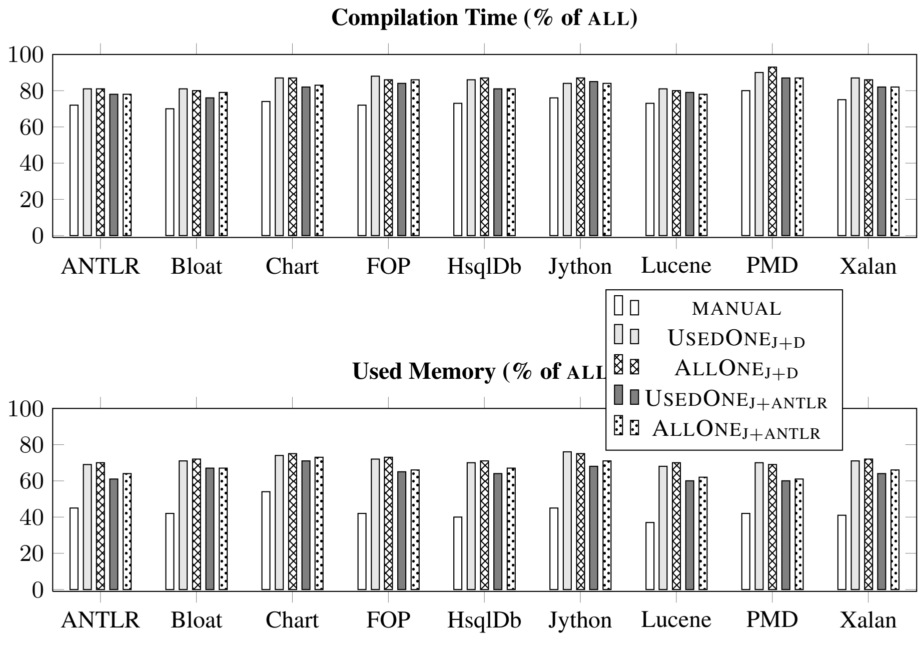


Fig. 10. Results from Experiment D: Compilation of DaCapo benchmarks using combined configurations from the Jacks and DaCapo suites. All results are shown as compilation time and memory usage as percent of the results for the ALL configuration and results for MANUAL are included for comparison.

We can sort the ALLONE configurations in the same fashion:

82%: $\text{ALLONE}_{\text{J+ANTLR}}$ (mem. 66%), $\text{ALLONE}_{\text{J+BLOAT}}$ (mem. 68%)
84%: $\text{ALLONE}_{\text{J+CHART}}$ (mem. 66%)
85%: $\text{ALLONE}_{\text{J+D}}$ (mem. 72%)
88%: $\text{ALLONE}_{\text{J+FOP}}$ (mem. 69%), $\text{ALLONE}_{\text{J+JYTHON}}$ (mem. 69%), $\text{ALLONE}_{\text{J+XALAN}}$ (mem. 70%)
89%: $\text{ALLONE}_{\text{J+LUCENE}}$ (mem. 71%), $\text{ALLONE}_{\text{J+PMD}}$ (mem. 71%)

We note that the influence of the benchmarks improve the average performance of the Jacks configurations (83%) with one or two percent. It is interesting to note that the benchmarks providing the best performance on average for Jacks, independent of configuration, are those with small coverage and few lines of code. These results indicate that it is worth combining a compiler test suite with a normal program, but that the program should not be too large or complicated. This way, we will end up with a configuration that caches attributes that end up in the USEDONE set of any small intricate program, as well as in the USEDONE set of larger programs, but without caching attributes that seem to be less commonly used many times. Further, it should be noted that the memory usage results for the combined $\text{ALLONE}_{\text{J+D}}$ and $\text{USEDONE}_{\text{J+D}}$ present unexpected results when compiling Jython. Presumably, the first configuration should use more memory than the second configuration, but the results show the reverse. The difference is slight but still statistically significant. At this point we have no explanation for this unexpected result.

6 Related Work

There has been a substantial amount of research on optimizing the performance of attribute evaluators and to avoid storing all attribute instances in the AST. Much of this effort is directed towards optimizing static visit-oriented evaluators, where attribute evaluation sequences are computed statically from the dependencies in an attribute grammar. For RAGs, such static analysis is, in general, not possible due to the reference attributes. As an example, Saarinen introduces the notion of *temporary attributes* that are not needed outside a single visit, and shows how these can be stored on a stack rather than in the AST [23]. The attributes we have classified as ONE correspond to such temporary attributes: they are accessed only once, and can be seen as stored in the stack of recursive attribute calls. Other static analyses of attribute grammars are aimed at detecting *attribute lifetimes*, i.e., the time between the computation of an attribute instance until its last use. Attributes whose instances have non-overlapping lifetimes can share a global variable, see, e.g., [17]. Again, such analysis cannot be directly transfered to RAGs due to the use of reference attributes.

Memoization is a technique for storing function results for future use, and is used, for example, in dynamic programming [3]. Our use of cached attributes is a kind of memoization. Acar et al. present a framework for selective memoization in a function-oriented language [1]. However, their approach is in a different direction than ours, intended to help the programmer to use memoized functions more easily and with more control, rather than to find out which functions to cache. There are also other differences between memoization in function-oriented programming, and in our object-oriented evaluator. In function-oriented programming, the functions will often have many and complex arguments that can be difficult or costly to compare, introducing substantial overhead for memoization. In contrast, our implementation is object-oriented, reducing most attribute calls to parameterless functions which are cheap to cache. And for parameterized attributes, the arguments are often references which are cheap to compare.

7 Conclusions and Future Work

We have presented a profiling technique for automatically finding a good caching configuration for compilers generated from RAG specifications. Since the attribute dependencies in RAGs cannot be computed statically, but depend on the evaluation of reference attributes, we have based the technique on profiling of test programs. We have introduced the notion of an attribute dependency graph with call counts, extracted from an actual compilation. Experimental evaluation on a generated Java compiler shows that by profiling on only a single program with an attribute coverage of only 67%, we reach a mean compilation speed-up of 20% and an average decrease in memory usage of 38%, as compared to caching all configurable attributes. This is close to the average compilation speed-up obtained for a manually composed expert configuration (25%). The corresponding average decrease in memory usage for the manual configuration (53%) is still significantly better. Our evaluation shows that we get similar performance improvements for both tested cache configuration approaches. Given these results, we would recommend the ALLONE configuration due to its higher robustness.

We find these results very encouraging and intend to continue this work with more experimental evaluations. In particular, we would like to study the effects of caching, or not caching, parameterized attributes, and to apply the technique to compilers for other languages. Further, we would like to study the effects of analyzing the content of the attribute equations. Most likely there are attributes which only return a constant or similar and, hence, should not benefit from caching independent of the number of calls. Finally, it would be interesting to further study the differences between the cache configurations from the profiler and the manual configuration.

Acknowledgements

We are grateful to Torbjörn Ekman for fruitful discussions and for implementing support in JastAdd for profiling and separate cache configurations. Thanks also to anonymous reviewers for comments on an earlier version of this paper.

References

1. Acar, U.A., Blelloch, G.E., Harper, R.: Selective memoization. In: POPL, pp. 14–25. ACM, New York (2003)
2. Åkesson, J., Ekman, T., Hedin, G.: Implementation of a Modelica compiler using JastAdd attribute grammars. Science of Computer Programming 75(1-2), 21–38 (2010)

3. Bellman, R.: Dynamic Programming. Princeton University Press, Princeton (1957)
4. Blackburn, S.M., Garner, R., Hoffmann, C., Khang, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Moss, B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: java benchmarking development and analysis. In: OOPSLA 2006: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 169–190. ACM, New York (2006)
5. Blackburn, S.M., McKinley, K.S., Garner, R., Hoffmann, C., Khan, A.M., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A.L., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanovic, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: Wake up and smell the coffee: evaluation methodology for the 21st century. Communications of the ACM 51(8), 83–89 (2008)
6. Boyland, J.T.: Remote attribute grammars. Journal of the ACM 52(4), 627–687 (2005)
7. Bürger, C., Karol, S., Wende, C.: Applying attribute grammars for metamodel semantics. In: Proceedings of the International Workshop on Formalization of Modeling Languages. ACM Digital Library (2010)
8. Ekman, T., Hedin, G.: Modular Name Analysis for Java Using JastAdd. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2005. LNCS, vol. 4143, pp. 422–436. Springer, Heidelberg (2006)
9. Ekman, T., Hedin, G.: The Jastadd Extensible Java Compiler. In: 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2007), pp. 1–18. ACM, New York (2007)
10. Farrow, R.: Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In: SIGPLAN 1986: Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction, pp. 85–98. ACM, New York (1986)
11. Hedin, G.: Reference Attributed Grammars. Informatica (Slovenia) 24(3), 301–317 (2000)
12. Hedin, G., Magnusson, E.: JastAdd: an aspect-oriented compiler construction system. Science of Computer Programming 47(1), 37–58 (2003)
13. Huang, S.S., Hormati, A., Bacon, D.F., Rabbah, R.M.: Liquid metal: Object-oriented programming across the hardware/Software boundary. In: Ryan, M. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 76–103. Springer, Heidelberg (2008)
14. Ibrahim, A., Jiao, Y., Tilevich, E., Cook, W.R.: Remote batch invocation for compositional object services. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 595–617. Springer, Heidelberg (2009)
15. Jourdan, M.: An optimal-time recursive evaluator for attribute grammars. In: Paul, M., Robinet, B. (eds.) Programming 1984. LNCS, vol. 167, pp. 167–178. Springer, Heidelberg (1984)
16. Jouve, W., Palix, N., Consel, C., Kadionik, P.: A SIP-based programming framework for advanced telephony applications. In: Schulzrinne, H., State, R., Niccolini, S. (eds.) IPTComm 2008. LNCS, vol. 5310, pp. 1–20. Springer, Heidelberg (2008)
17. Kastens, U.: Lifetime analysis for attributes. Acta Informatica 24(6), 633–651 (1987)
18. Kats, L.C.L., Sloane, A.M., Visser, E.: Decorated Attribute Grammars: Attribute Evaluation Meets Strategic Programming. In: de Moor, O., Schwartzbach, M.I. (eds.) CC 2009. LNCS, vol. 5501, pp. 142–157. Springer, Heidelberg (2009)
19. Knuth, D.E.: Semantics of Context-free Languages. Mathematical Systems Theory 2(2), 127–145 (1968); Correction: Mathematical Systems Theory 5(1), 95–96 (1971)
20. Nilsson-Nyman, E., Ekman, T., Hedin, G., Magnusson, E.: Declarative intraprocedural flow analysis of Java source code. In: Proceedings of the Eight Workshop on Language Description, Tools and Applications (LDTA 2008). Electronic Notes in Theoretical Computer Science, Elsevier B.V., Amsterdam (2008)

21. Poetzsch-Heffter, A.: Prototyping realistic programming languages based on formal specifications. Acta Informatica 34(10), 737–772 (1997)
22. Rajan, H.: Ptolemy: A language with quantified, typed events (2010), `http://www.cs.iastate.edu/~ptolemy/`
23. Saarinen, M.: On constructing efficient evaluators for attribute grammars. In: Ausiello, G., Böhm, C. (eds.) ICALP 1978. LNCS, vol. 62, pp. 382–397. Springer, Heidelberg (1978)
24. Schäfer, M., Ekman, T., de Moor, O.: Sound and Extensible Renaming for Java. In: Kiczales, G. (ed.) 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2008). ACM Press, New York (2008)
25. Sloane, A.M., Kats, L.C.L., Visser, E.: A Pure Object-Oriented Embedding of Attribute Grammars. In: Proceedings of the 9th Workshop on Language Descriptions, Tools and Applications, LDTA 2009 (2009)
26. The DaCapo Project. The DaCapo Benchmarks (2009), `http://dacapobench.org`
27. The JastAdd Caching Trac. Performance results for JastAddJ (2010), `http://svn.cs.lth.se/trac/jastadd-caching`
28. The Mauve Project. Jacks (Jacks is an Automated Compiler Killing Suite) (2009), `http://sources.redhat.com/mauve`
29. Vogt, H., Doaitse Swierstra, S., Kuiper, M.F.: Higher-order attribute grammars. In: PLDI, pp. 131–145 (1989)
30. Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: An extensible attribute grammar system. Science of Computer Programming 75(1-2), 39–54 (2010)
31. Van Wyk, E., Krishnan, L., Bodin, D., Schwerdfeger, A.: Attribute grammar-based language extensions for java. In: Bateni, M. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 575–599. Springer, Heidelberg (2007)

Reference Attribute Grammars for Metamodel Semantics

Christoff Bürger, Sven Karol, Christian Wende, and Uwe Aßmann

Institut für Software- und Multimediatechnik
Technische Universität Dresden
Dresden, Germany
{christoff.buerger,sven.karol,c.wende,uwe.assmann}@tu-dresden.de

Abstract. While current metamodelling languages are well-suited for the structural definition of abstract syntax and metamodelling platforms like the Eclipse Modelling Framework (EMF) provide various means for the specification of a textual or graphical concrete syntax, techniques for the specification of model semantics are not as matured. Therefore, we propose the application of reference attribute grammars (RAGs) to alleviate the lack of support for formal semantics specification in metamodelling. We contribute the conceptual foundations to integrate metamodelling languages and RAGs, and present *JastEMF* — a tool for the specification of EMF metamodel semantics using JastAdd RAGs. The presented approach is exemplified by an integrated metamodelling example. Its advantages, disadvantages and limitations are discussed and related to metamodelling, attribute grammars (AGs) and other approaches for metamodel semantics.

1 Introduction

Metamodelling is a vital activity for Model-Driven Software Development (MDSD). It covers the definition of structures to represent abstract syntax models, the specification of a concrete syntax, and the specification of the meaning of models [1]. While infrastructures like the Eclipse Modelling Framework (EMF) [2] provide means for the specification of abstract syntax and various associated tools for the specification of a textual or graphical concrete syntax, techniques for the specification of model semantics are not as matured [1].

In this paper we propose the application of RAGs [3] — a well-investigated extension of Knuth's classic AGs [4] — to alleviate the lack of support for formal semantics specification in metamodelling. They enable (1) the specification of semantics on tree and graph-based abstract syntax structures with unique spanning trees, (2) completeness and consistency checks of semantic specifications, and (3) the generation of an implementation of semantics specifications.

The contributions of this paper are as follows: The next section discusses common concerns in the specification of metamodels, identifies key capabilities we target with *semantics-integrated metamodelling* and introduces a motivating example. In Section 3 we sketch general foundations for the application of RAGs for metamodel semantics — our semantics-integrated metamodelling approach. In Section 4 we demonstrate the

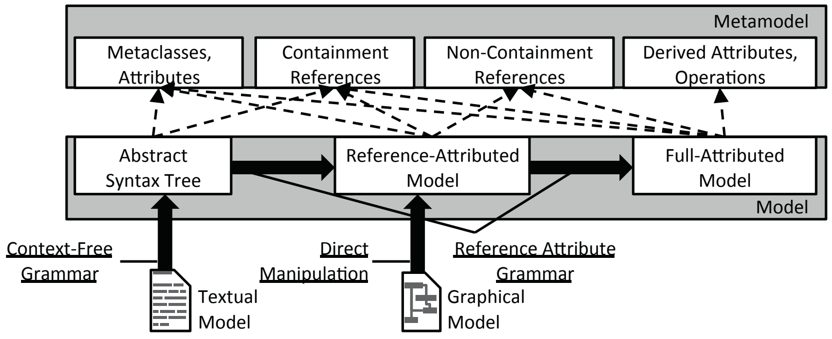


Fig. 1. Transformations in Metamodelling

feasibility of this idea by presenting *JastEMF*, a tool that integrates the Ecore metamodelling language and the JastAdd [5] attribute grammar system. In Section 5 we describe the application of our semantics-integrated metamodelling approach to implement the SiPLE-Statemachines language introduced in Section 2. In Section 6 we evaluate JastEMF against the key capabilities identified for semantics-integrated metamodelling based on our experiences with the SiPLE-Statemachines case study. Finally, we discuss related work in Section 7 and conclude our contribution in Section 8.

2 Motivation for Semantics-Integrated Metamodelling

This section motivates our idea of semantics-integrating metamodelling with RAGs by identifying methodical gaps to achieve common objectives in metamodelling. Afterwards, we introduce a set of key capabilities we target with *semantics-integrated metamodelling* and introduce SiPLE-Statemachines, an exemplary metamodel project that serves as a continuous example throughout this paper.

2.1 Metamodelling: Objectives, Transformations, Specifications

Metamodelling has the objective to specify the implementation of a modelling language. Such implementation should provide means to transform programs (i.e., models) starting from (1) their textual representation to (2) their abstract syntax representation and finally (3) representations of their static and execution semantics [1]. In the metamodelling world, all these representations and transformations are related to the language's metamodel, which typically *declares* the data structures that are used for representing language constructs in abstract syntax models and is the interface for the *specification* of concrete syntax and semantics. As depicted in Figure 1, each transformation's input and output model is built using specific kinds of constructs of the language metamodel. A first transformation — typically specified using regular expressions and context-free grammars — derives an abstract syntax tree (AST) from textual symbols. The data structure required to represent the AST is solely declared by the `Metaclasses`, `Attributes` and `Containment References` in the metamodel. In a second transformation, the structures that are declared as `Non-Containment References` (e.g., connecting variable usage with variable declarations) need to be derived, resulting in a reference-attributed model — i.e., the abstract syntax tree with a superimposed reference graph. Graphical editors often directly operate on such reference-attributed models (cf. Figure 1) and rely on a direct manipulation of non-containment references. A last transformation performs semantics evaluations to derive values for `Derived Attributes` and executes `Operations` declared in the metamodel. Note that the reference-attributed model and the full-attributed model still contain the abstract syntax tree as their *spanning tree*.

All the above mentioned transformations are important artefacts of a language and, thus, motivate a formal definition. However, formal approaches for the specification of static or execution semantics are not yet established in the metamodelling world. As depicted in Figure 1 we aim at closing this gap by the application of RAGs for the specification of metamodel semantics.

2.2 Capabilities of Semantics-Integrated Metamodelling

By applying RAGs to achieve *semantics-integrated metamodelling* we expect to combine the benefits of metamodelling frameworks and attribute grammars. However, to our experience, integration means not only combination of benefits but often also a compromise of the technical and methodical capabilities of the individual approaches. We therefore collected a number of technical key capabilities to be contributed by each individual approach that afterwards will be used to evaluate our integrated solution.

Metamodelling frameworks (e.g., the EMF) are built around a metamodelling language (e.g., Ecore) and typically provide tools for the specification and implementation of modelling languages and their tooling. In particular they provide:

MM 1: Metamodelling Abstraction: Metamodelling language that provides a dedicated abstraction to specify language metamodels.
MM 2: Metamodelling Consistency: Tools to check the structural completeness and consistency of metamodel specifications.
MM 3: Metamodel Implementation Generators: Generators to derive implementations from metamodel specifications.
MM 4: Metamodel/Model Compatibility: A common repository and representation that enables integration of modelling languages and models.
MM 5: Tooling Compatibility: Common platform for tool integration/application:

- **MM 5.1: Model-to-Model Transformation Tools:** E.g., ATL [6] or XTend [7].
- **MM 5.2: Model-to-Text Transformation Tools:** Code generators and model-driven template languages like Mofscript [8] or XPand [7].
- **MM 5.3: Text-to-Model Transformation Tools:** Parser generators for models as EMFText [9], Monticore [10] or XText [11].
- **MM 5.4: Generic Model Editors:** Generic tools to access and edit models as the Generic EMF editor [2] or Exeed [12].
- **MM 5.5: Tooling Generators:** Tooling to specify and generate textual (EMFText, Monticore, XText) or graphical (GMF [2], EuGENia[1]) model editors.

[1] `http://www.eclipse.org/gmt/epsilon/doc/articles/eugenia-gmf-tutorial/`

Attribute grammar systems typically provide means for the specification of languages' abstract syntax and semantics and tools to derive an implementation from such specifications. In particular they provide:

AG 1: Semantics Abstraction: Well-investigated, declarative abstraction for formal semantics specifications.
AG 2: Semantics Consistency: Tooling to check the structural completeness and consistency of semantics specifications.
AG 3: Semantics Generators: Generators to derive an implementation (i.e., evaluator) from semantics specifications.
AG 4: Semantics Modularity: Modular, extensible semantics specifications [13].

2.3 SiPLE-Statemachines: A Typical Modelling Language

To exemplify and evaluate semantics-integrated metamodelling we will use a typical modelling scenario. It is built upon a Simple imperative Programming Language Example (SiPLE) and a statemachine language which are combined to support the modelling of executable statemachines.

SiPLE's main features are boolean, integer, and real arithmetics, nested scopes, nested procedure declarations, recursion, while loops and conditionals. All these features of SiPLE have the intuitive semantics familiar from imperative programming languages. Listing 1.1 shows a basic SiPLE program that asks the user for a number, computes its Fibonacci value and prints it.

Listing 1.1. Fibonacci Numbers in SiPLE

```
Procedure main() Begin
 Procedure fibonacci(Var n:Integer) : Integer Begin
  If n = 0 Or n = 1 Then Return 1; Fi;
  Return fibonacci(n-2) + fibonacci(n-1);
 End;

 Var n:Integer;
 Read n;
 Write fibonacci(n);
End;
```

Statemachines describe the behaviour of systems using a state-based abstraction [14]. In contrast to the textual syntax of SiPLE, they are typically modelled using a graphical notation. The exemplary statemachine depicted in Fig. 2 describes the behaviour of a phone. It uses concepts like states (e.g., `Dialing`), transitions (e.g., `incoming call`), guard conditions and actions. With transitions the phone reacts on particular events from the environment by changing states, e.g., `incoming call` where the phone changes from `Waiting` to `Ringing`. Guards and actions enable a more fine-grained specification of boolean conditions and imperative behaviour, respectively. Therefore, we want to combine the statemachine language and SiPLE to SiPLE-Statemachine. Consider for instance the state `Dialing` where an entry action is used to read a number from the user (`Read number;`). SiPLE action statements and boolean guard expressions can also be associated to transitions. For instance, when the phone is

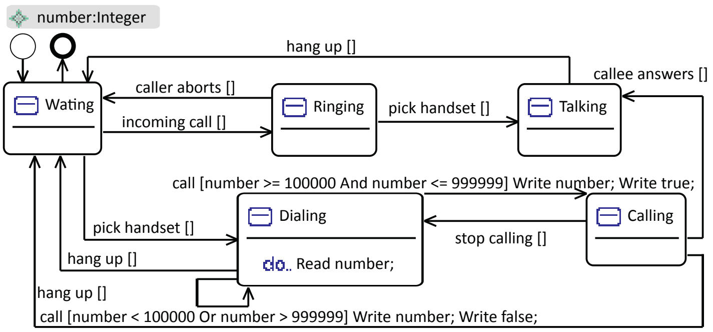


Fig. 2. Phone example modelled in a generated GMF editor for SiPLE-Statemachine

in the `Dialing` state and receives a `call` event, it checks the entered number using a boolean SiPLE expression and changes its state in accordance. While walking the according transition it writes the dialed or rejected number to the standard output (`Write number; Write true;` or `Write number; Write false;`).

In this paper we will demonstrate and evaluate the integration of metamodelling and RAGs by implementing SiPLE-Statemachines. Relying on the key capabilities identified in Section 2.2 we plan to provide:

- A metamodel (MM 1, MM 2, MM 3), textual concrete syntax (MM 5.3, MM 5.5) and attribute-grammar semantics (AG 1, AG 2, AG 3) for SiPLE.
- An interpreter that implements an execution semantics for SiPLE (MM 5).
- The SiPLE-Statemachine metamodel (MM 1, MM 2, MM 3) composed from the two metamodels of SiPLE and Statemachine (MM 4).
- A generated graphical editor for SiPLE-Statemachine (MM 5.5).
- An attribute-grammar semantics for SiPLE-Statemachine (AG 1, AG 2, AG 3, AG 4), e.g., for mapping transition labels to states and to reason about state reachability.
- An execution semantics for SiPLE-Statemachine by implementing a model-to-text transformation (MM 5.2) for statemachines to plain SiPLE code.

3 Foundations of Attribute Grammars for Metamodel Semantics

Because RAGs rely on a specific representation of abstract syntax, their application in metamodelling requires an integration with metamodel constructs. In this section we prepare such integration by investigating RAGs' specific syntax requirements, clarifying what kind of semantics they can specify and which kind of model information these semantics represent, and finally showing that most metamodelling languages indeed satisfy RAGs' syntax requirements.

3.1 Reference Attribute Grammars and Metamodel Semantics

RAGs are used to specify semantics for tree structures that are usually specified using a context-free grammar (CFG). Given a tree the RAG annotates it with values and imposes a graph on it representing the language's semantics. Because we like to use RAGs for metamodel semantics, we must identify metamodel constructs that induce such tree-structure in model instances. We like to use our approach not only for certain metamodels, but rather for any metamodel developed in a metamodelling language. Thus, the separation of metamodel constructs into tree- and graph-inducing structures — into syntax and semantics — should be metamodelling language inherent, i.e. be implied by the meta-metamodel[2]. For this purpose, we investigate common metamodelling languages' concepts and distinguish them into syntactic (tree structure *defining*) and semantic (graph structure *declaring*) ones. Both sets will be disjunct. Given this separation RAGs are indeed appropriate to specify metamodel semantics in the sense that they can be used to specify the transformation of abstract syntax trees to reference attributed graphs and full-attributed graphs (cf. Figure 1). RAGs can be used for any kind of static model semantics like model checking and analysis. Definition 3.1 summarises the foundations of integrating RAGs and metamodels:

Definition 3.1 (Metamodel Semantics): Let Ω be a metamodel and E_Ω the finite set of its elements. Let E_{syn_Ω} and E_{sem_Ω} be disjunct subsets of E_Ω, whereas $E_\Omega = E_{syn_\Omega} \cup E_{sem_\Omega}$. Let E_ω be the set of entities of a model instance $\omega \in \Omega$. Since $\omega \in \Omega$, all entities $e \in E_\omega$ have a type $t_e \in E_\Omega$. Let S_Ω be a function that defines for all $\omega \in \Omega$ for each entity $e \in E_\omega$ with $t_e \in E_{sem_\Omega}$ the value of e. We call S_Ω a metamodel semantic for Ω. Iff E_{syn_Ω} specifies a spanning tree for each $\omega \in \Omega$, S_Ω can be specified with a RAG.

The metamodel semantics S_Ω can depend on any metamodel element $me \in E_\Omega$. They even can depend on themselves, in which case they are only well-defined if there exists a fix-point. Thus, different model instances can only have different semantics, if the semantics depend on syntactic elements $me \in E_{syn_\Omega}$. Colloquially explained, a model's semantics (i.e., all entities $\{e | e \in E_\omega \wedge t_e \in E_{sem_\Omega}\}$) depend on its structure (i.e., all entities $\{e | e \in E_\omega \wedge t_e \in E_{syn_\Omega}\}$).

What remains to show is, which metamodelling concepts belong to E_{syn_Ω} and E_{sem_Ω} and that indeed E_{syn_Ω} specifies a tree structure.

3.2 Common Metamodelling Languages and Abstract Syntax Trees

Most metamodelling languages support (1) *metaclasses* consisting of (2) *non-derived* and (3) *derived attributes* and (4) *operations*. Metaclasses can be related to each other by (5) *containment* and (6) *non-containment relationships*. Non-derived attribute values represent AST terminals in models and, thus, are in E_{syn_Ω}. Containment references model that instances of a metaclass C_1 consist of instances of a metaclass C_2. The contained C_2 instances are an inextricable, structural part of the C_1 instances. The relationship between C_1 and C_2 is a composite and iff an instance $e_2 \in C_2$ is a composite of an instance $e_1 \in C_1$, e_1 cannot be a composite of e_2. Thus, containment relationships specify tree structures. They are in E_{syn_Ω}. Derived attributes and side-effect free operations model the values that can be calculated from other values of a given model. Non-containment relationships model arbitrary references between

[2] Most metamodelling languages are specified in themselves, such that it is appropriate just to talk about metamodels in the following.

metaclasses. Thus, derived attributes, side-effect free operations, and non-containment relationships are in E_{sem_Ω}. Operations with side-effects can model either, extensions of models derived from existing model information or arbitrary model manipulations. Derived model extensions are in E_{sem_Ω}, because they can be considered to be part of the graph imposed by semantics (w.r.t. AGs such derived model extensions are higher-order attributes [15]). Operations that represent arbitrary model manipulations (e.g., to delete model elements or imperatively add new elements) cannot be handled by our definition of metamodel semantics.

3.3 Graphs and (Partial) Reference-Attributed Models

In the domain of modelling, often reference-attributed models (cf. Section 2.1 and Fig. 1) are the starting point for semantic evaluations. A typical scenario are models developed in graphical editors. Of course, it is no problem for a RAG-based metamodel semantic S_Ω if elements of E_{sem_Ω} of a model instance have a predefined value — i.e., if instead of a tree the semantic evaluation starts from a graph with a unique spanning tree.

Throughout semantic evaluation, a RAG evaluator can use such predefined values and simply ignore their specified semantics. If all occurrences of a non-containment reference, for every possible model instance, have a predefined value, the specification of its semantics can even be omitted. Thus, (partial) reference-attributed models do not influence the applicability of our RAG approach for metamodel semantics.

4 JastEMF: An Exemplary Attribute Grammar and Metamodelling Language Integration

In this section, we discuss the integration of an exemplary metamodelling framework (EMF [2]) and RAG system (JastAdd [5]). We shortly introduce both approaches and then discuss the details of their integration in *JastEMF*.

4.1 The Eclipse Modelling Framework

The EMF is a common metamodelling infrastructure for the Eclipse platform providing metamodel development and implementation tools based on the metamodelling language Ecore [2]. EMF contributes tools to edit Ecore metamodel specifications, check their consistency and generate a Java-based implementation of the metamodel specifications. The framework is used for the implementation of a plethora of modelling languages[3], and is an important integration platform for various modelling tools.

For the definition of concrete syntax the EMF is complemented by various tools to specify a concrete syntax in relation to a metamodel. Editor generators that are tightly integrated with EMF like EMFText [9] or XText [11] enable the declarative specification of context-free grammars to define parsers, printers, and advanced textual editors for models. There are also tools to realise a graphical (diagrammatic) model syntax, e.g., the Graphical Modelling Framework (GMF) [2].

[3] `http://www.emn.fr/z-info/atlanmod/index.php/Ecore`

For the specification of semantics, the EMF mainly relies on Java source code that evaluates derived attributes or implements operations declared in the metamodel. Besides the application of the Object Constraint Language (OCL) [16] or model-transformations, we are not aware of formal, mature techniques to specify static and execution semantics in EMF. For a further discussion of approaches for metamodel semantics we refer to Section 7.

4.2 The JastAdd Metacompiler

JastAdd [5,13] is an object-oriented compiler generation system. It allows to generate a Java implementation of a demand-driven semantics evaluator from a given AG. Besides the basic attribute grammar concepts [4], JastAdd supports advanced AG extensions such as reference [3] (RAGs) and circular attributes [17].

JastAdd has two specification languages. One to specify abstract syntax and another to specify an attribution (i.e. semantics). Abstract syntax specifications consist of node type declarations (non-terminals) and their child nodes (arbitrary list of terminals and non-terminals). Language semantics is usually specified within several modules containing attribute definitions and attribute equations that are associated with node types of the abstract syntax.

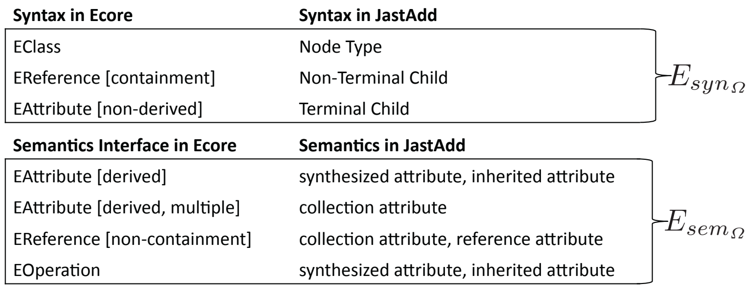
Given a set of AST and attribute specifications the JastAdd compiler generates a Java class for each node type, accessors for the node's children nodes, and methods for each attribute defined for the node type. The code required for attributes' evaluation is generated into their method bodies. Consequently, evaluation of semantics can be triggered by accessing the corresponding methods.

4.3 Integrating EMF and JastAdd

Because both EMF and JastAdd provide code generation for Java, they are well-suited to explore semantics-integrated metamodelling. For their practical integration it is required (1) to merge the Java classes that represent a language's abstract syntax in EMF and in JastAdd and (2) to apply the generated attribute evaluator to compute EMF models' semantics.

Based on the integration foundations presented in Section 3 we derived a mapping of elements in the Ecore metamodel and specification concepts used by JastAdd. The concrete mappings depicted in Figure 3 are grouped in two sets. The first set contains elements related only to model *syntax* (E_{syn_Ω}). The second set contains the elements related to model *semantics* (E_{sem_Ω}). In the second set constructs of the Ecore metamodel are typically used to declare the *semantics interface* of specific elements while the corresponding JastAdd construct specifies the element's semantics. Depending on its actual syntax and semantics, multiple mappings are possible.

In general, derived properties, non-containment references and operations that are side-effect free are considered to be static semantics, whereas their semantics can be specified using synthesized or inherited attributes (reference attributes in the case of a non-containment reference). If the cardinality of a derived property or non-containment reference is greater than one, often collection attributes [18,19] are much more convenient than ordinary attributes, since they permit to collect remotely located AST nodes w.r.t. conditions and reference attributes. However, since JastAdd attributes can have any



Syntax in Ecore	Syntax in JastAdd
EClass	Node Type
EReference [containment]	Non-Terminal Child
EAttribute [non-derived]	Terminal Child

E_{syn_Ω}

Semantics Interface in Ecore	Semantics in JastAdd
EAttribute [derived]	synthesized attribute, inherited attribute
EAttribute [derived, multiple]	collection attribute
EReference [non-containment]	collection attribute, reference attribute
EOperation	synthesized attribute, inherited attribute

E_{sem_Ω}

Fig. 3. Integrating Ecore and JastAdd

valid Java type, it is also possible to specify ordinary attributes that represent collections. Operations with side-effects should not be realized by attributes, but rather by ordinary Java methods specified within JastAdd attribute specifications. Such intertype declarations are woven by JastAdd as known from aspect weaving tools like AspectJ [20].

JastEMF's Integration Process. To realise the integration of EMF and JastAdd, we implemented *JastEMF*[4]. Given an Ecore metamodel with in JastAdd specified semantics — a so called *JastEMF integration project* — JastEMF can be executed to generate an integrated language implementation, i.e. an EMF metamodel implementation with integrated JastAdd semantics. *Integration projects* must provide the following artefacts:

– An Ecore metamodel declaring the language's abstract syntax.
– An Ecore generator model configurating EMF and JastEMF code generation.
– A set of JastAdd attribute specifications defining the language's semantics that satisfy the mappings defined for concepts in E_{sem_Ω} (cf. Fig. 3).

Based on these artefacts, JastEMF's generation process (cf. Figure 4) reuses the generators for JastAdd and EMF and merges the generated Java classes in accordance to the introduced mapping. First, the process uses the EMF Generator Model, which is fed to the `(1) EMF Code Generator` to generate an EMF Metamodel Implementation and the `(2) JastEMF JastAdd Adaptation Specification Generator` to derive a JastAdd AST Specification and a JastAdd Repository Adaptation Specification. The Repository Adaptation Specification contributes attribute specifications that adapt the JastAdd Evaluator Implementation to use the EMF repository instead of its own internal repository. As a second input the process requires the JastAdd Semantics Specifications. The JastAdd AST Specification, the JastAdd Repository Adaptation Specification and the JastAdd Semantics Specifications are used by the `(3) JastAdd Compiler` to generate a JastAdd Evaluator Implementation. To integrate this Evaluator with the Metamodel Implementation it has to be refactored to incorporate metamodel naming conventions and package structures. Therefore, the `(4) JastEMF Refactoring Generator` derives a JDT[5] refactoring script from the metamodel and applies it `(5)`. For an overview of the refactorings applied we refer to [21]. Finally, the Refactored Evaluator Implementation is merged with the EMF Metamodel

[4] `www.jastemf.org`
[5] `http://www.eclipse.org/jdt/`

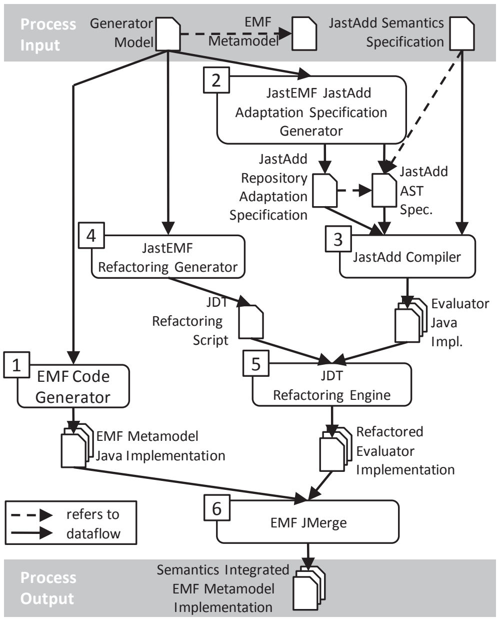

Process Input
Generator Model
EMF Metamodel
JastAdd Semantics Specification
2 JastEMF JastAdd Adaptation Specification Generator
JastAdd Repository Adaptation Specification
JastAdd AST Spec.
3 JastAdd Compiler
4 JastEMF Refactoring Generator
JDT Refactoring Script
Evaluator Java Impl.
1 EMF Code Generator
5 JDT Refactoring Engine
EMF Metamodel Java Implementation
Refactored Evaluator Implementation
refers to
dataflow
6 EMF JMerge
Process Output
Semantics Integrated EMF Metamodel Implementation

Fig. 4. JastEMF's Generation Process

Implementation using `(6) EMF JMerge`. This last step results in a metamodel implementation with tightly integrated semantics, where semantic declarations from the EMF metamodel are combined with their attribute-based specifications defined in the JastAdd semantics.

With JastEMF the complexity of this integration process is completely hidden for developers. Also, developers can work with EMF and JastAdd as usual.

5 SiPLE-Statemachines Case Study

In the following, we present the application of semantics-integrated metamodelling for implementing SiPLE-Statemachine.

Therefore, we discuss: (1) the application of EMF Ecore for specifying and integrating SiPLE and SiPLE-Statemachine abstract syntax, (2) the application of JastAdd for the specification of SiPLE and SiPLE-Statemachine static and execution semantics, and (3) the application of JastEMF to generate an integrated SiPLE-Statemachine implementation.

5.1 Modelling Abstract Syntax with EMF

SiPLE Abstract Syntax. The SiPLE metamodel is presented in Figure 5. A `CompilationUnit` consists of `Declarations`, which can be `VariableDeclarations` declaring a variable's name and type or `ProcedureDeclarations` declaring a procedure. Each procedure has a name, a return type, a list of parameters and a body that

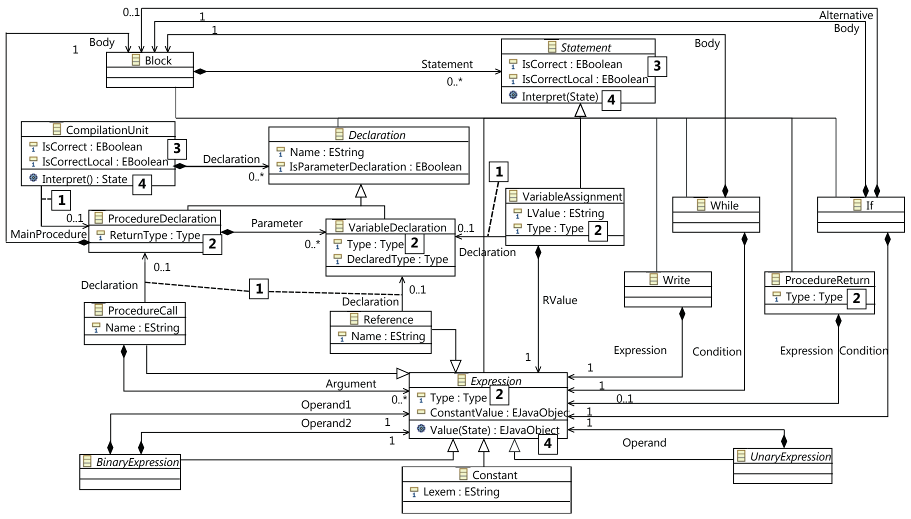


Fig. 5. SiPLE Metamodel

is a `Block`. Each `Block` consists of a `Statement` sequence. In SiPLE, nearly everything is a `Statement`, such as `While` loops, `If` conditionals, `Expressions`, `Declarations` and `VariableAssignments`. Expressions can be `BinaryExpressions` or `UnaryExpressions`, whose concrete sub-classes, e.g., `Addition` and `Not`, are not presented in the figure. Furthermore, there are primitive expressions such as `Constants`, `References` and `ProcedureCalls`.

The metamodel specifies a spanning tree (containment references, non-derived attributes) enriched with semantics interfaces (non-containment references, derived attributes, operations). For a better understanding, we assigned numbers to different parts of the semantics interfaces declared in the metamodel. Parts that are to be computed by name analysis are marked with $\boxed{1}$, e.g., each `VariableAssignment` and each `ProcedureCall` in a well-formed program has a reference pointing to their respective declaration. Parts depending on type analysis are labeled by $\boxed{2}$, e.g., the derived attribute `Type` represents the actual type of an `Expression`. $\boxed{3}$ marked parts belong to the constraint checking realization, which currently consists of the `IsCorrect` and `IsCorrectLocal` attributes. Parts labeled with $\boxed{4}$ (e.g., `Interpret()`) declare the execution semantics interface for both runtime evaluation and constant folding.

SiPLE-Statemachine Abstract Syntax. The SiPLE-Statemachine metamodel is presented in Figure 6. A `StateMachine` consists of a set of `State` and `Transition Declarations`. Each `Transition` has a label representing an event that triggers the `Transition`. Furthermore, guard `Expressions` allow to specify boolean SiPLE expressions as additional conditions for transition triggering and SiPLE `Statements` allow to annotate actions to states and transitions that are executed when a state is entered or a transition is triggered respectively. Each `Transition` refers to a source and a target `State`.

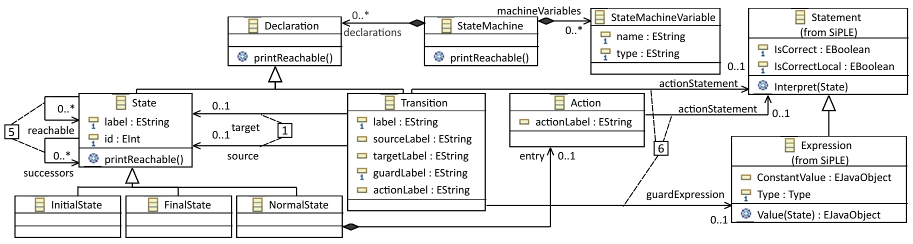


Fig. 6. SiPLE-Statemachine Metamodel

The metamodel's semantics are an extension of the semantics used for the JastAdd statemachine tutorial in [22]. Again, we assigned numbers to parts of the semantics interface. As above, $\boxed{1}$ marks parts belonging to name analysis. In SiPLE-Statemachine name analysis is used to compute the actual source and target states from the corresponding labels of a `Transition` object. Further semantics analysis that computes additional information such as the successor relation and transitive closure of all states reachable from a given state are labeled by $\boxed{5}$. Parts that are used to parse textual action and guard labels to appropriate SiPLE fragments are marked with $\boxed{6}$.

5.2 Specifying Semantics with JastAdd

SiPLE Semantics. There are four semantic concerns that completely specify SiPLE's static and execution semantics. With JastAdd, each concern can be specified as an *aspect*. A *core specification* is used to *declare* each concern's semantics, i.e., to *declare* all attributes (cf. Listing 1.2). The actual attribute *definitions* (the equations) reside in separate JastAdd specifications[6].

Listing 1.2. Excerpt from the SiPLE Core Specification

```
aspect NameAnalysis { // [1] in Figure 5
  // Procedure name space:
  inh Collection<ProcedureDeclaration> ASTNode.LookUpPDecl(String name);
  syn ProcedureDeclaration ProcedureCall.Declaration();
  syn ProcedureDeclaration CompilationUnit.MainProcedure();

  // Variable name space:
  inh Collection<VariableDeclaration> ASTNode.LookUpVDecl(String name);
  syn VariableDeclaration Reference.Declaration();
  syn VariableDeclaration VariableAssignment.Declaration();
}
aspect TypeAnalysis { // [2] in Figure 5
  syn Type VariableDeclaration.Type();
  syn Type VariableAssignment.Type();
  syn Type ProcedureReturn.Type();
  syn Type Expression.Type();
}
```

[6] All specifications can be found at `www.jastemf.org`.

```
aspect ConstraintChecking { // [3] in Figure 5
  syn boolean ASTNode.IsCorrect();
  syn boolean ASTNode.IsCorrectLocal();
}
aspect Interpretation { // [4] in Figure 5
  public abstract Object Expression.Value(State vm) throws InterpretationException;
  public abstract void Statement.Interpret(State vm) throws InterpretationException;
  syn State CompilationUnit.Interpret();
}
```

NameAnalysis. SiPLE uses separate block-structured namespaces for variable and procedure declarations. Although there is a single global scope in each `Compilation-Unit`, each block introduces a new private scope, shadowing declarations in the outside scope. No explicit symbol tables are required to resolve visible declarations — the AST is the symbol table.

TypeAnalysis. SiPLE is a statically, strongly typed language. For each kind of expression its type is computed from the types of its arguments, e.g., if an addition has a `Real` number argument and an `Integer` argument the computed type will be `Real`. Types are statically computed for arithmetic operations, assignments, conditionals (`If`, `While`), procedure calls and procedure returns.

ConstraintChecking. Each language construct of a SiPLE program can be statically checked for *local* correctness, i.e., whether the node representing the construct satisfies all *its* context-sensitive language constraints or not. Of course, these checks are usually just simple constraints specified based on SiPLE's name and type analysis like "an `If` condition's type must be boolean" or "each reference must be declared". If all nodes of a (sub)tree — i.e., a program (fragment) — are local correct, the (sub)tree is correct.

Interpretation. SiPLE's execution semantics is also specified using JastAdd. We applied JastAdd's ability to use Java method bodies for attribute specifications. This allows for a seamless integration of a Java implementation of the operational semantics and the declarative, RAG-based static semantics analysis. The interpretation is triggered with a call to a `CompilationUnit`'s `Interpret()` operation that initialises a state object representing a stack for procedure frames and traverses the program's statements by calling their `Interpret(State)` operation.

SiPLE-Statemachine Semantics. The SiPLE-Statemachine semantics is mainly specified in three JastAdd aspects which are shown in Listing 1.3. The original source comes from [22]. To integrate SiPLE, we additionally introduced a `SiPLEComposition` aspect.

Listing 1.3. Excerpt from the SiPLE-Statemachine Core Specification

```
aspect NameAnalysis { // [1] in Figure 6
    syn lazy State Transition.source();
    syn lazy State Transition.target();
    inh State Declaration.lookup(String label);
    syn State Declaration.localLookup(String label);
}

aspect Reachability{ // [5] in Figure 6
    syn EList State.successors() circular [...];
  syn EList State.reachable() circular [...];
}
```

```
aspect SiPLEComposition { // [6] in Figure 6
    public Statement Action.getActionStatement();
    public Expression Transition.getGuardExpression();
    public Statement Transition.getActionStatement();
}
```

NameAnalysis. In SiPLE-Statemachine, name analysis maps textual labels in transitions to states. Since states are not block-structured, all declarations of the statemachine are traversed and each state label is compared to the looked up label. Note that a graphical editor may set a transition's source and target state directly.

Reachability. The synthesized `successor` attribute computes a state's direct successor relation from the set of declared transitions. In contrast to the original example, we declared the attribute to be an `EList` to achieve better graphical editor support and as circular because of editor notifications issues. Based on the successor relation, the `reachable` attribute computes a state's transitive closure, which can be displayed on demand in a graphical editor we generated for SiPLE-Statemachines.

SipleComposition. This helper aspect uses the SiPLE parser to parse and embed SiPLE `Expressions` and SiPLE `Statements` for guard and action labels. Actually, these parts belong to SiPLE and are integrated into SiPLE-Statemachine (cf. 6). However, since JastAdd does not support packages and always generates AST classes for the given AST specifications instead of reusing classes from existing packages as supported by the EMF, we had to model them as attributes.

5.3 Integration with Further Metamodelling Tools

To prepare the evaluation of JastEMF's integration approach we applied several metamodelling tools for further implementation tasks. As our focus is about the specification of semantics for metamodels using RAGs, we shortly describe their purpose but refer to respective publications for details:

- To generate a parser and an advanced text editor for SiPLE we use EMFText [9]. Amongst others, the generated editor supports syntax highlighting, code completion, outline and a properties view.
- To provide a graphical syntax for SiPLE-Statemachine we used EuGENia[7], a tool that processes specific metaclass annotations to generate based on their information (node or transition, shape, style, color, etc.) an according set of GMF [2] specifications. From such specifications the GMF framework then generates a well-integrated, powerful graphical editor for the SiPLE-Statemachine language (cf. Figure 2).
- To make SiPLE-Statemachine executable, we use an XPand [7] template to generate plain SiPLE code, which can be executed by a call to its `CompilationUnit`'s `Interpret` operation.

[7] `http://www.eclipse.org/gmt/epsilon/doc/articles/eugenia-gmf-tutorial/`

6 Evaluation

In this section we evaluate JastEMF based on our experience with the SiPLE-Statemachines case study and w.r.t. the semantics-integrated metamodelling capabilities presented in section 2.2. Afterwards, we discuss limitations of our approach and further experiences in applying JastEMF that motivate deeper investigation and future work.

6.1 Evaluating the Capabilities of Integrated Metamodelling.

The JastEMF integration process (cf. Section 4.3 Figure 4) is completely steered by a standard Ecore generator model, the according Ecore metamodel and a set of standard JastAdd specifications. For constructing, manipulating, validating and reasoning about the input metamodel and semantic specifications all the tools available for the respective artefact can be reused. Furthermore, the process reuses the EMF and JastAdd tooling for code generation and all applied refactorings and code merges retain the metamodel implementation's API.

Consequently, the key capabilities metamodelling abstraction (**MM 1**), metamodel implementation generators (**MM 3**), semantics abstraction (**AG 1**) and semantics generators (**AG 3**) are provided by JastEMF.

JastEMF also provides metamodelling tooling compatibility (**MM 5**). This is well demonstrated in the SiPLE-Statemachines case study by:

- Using the model-driven template language XPand to generate SiPLE code for statemachines (**MM 5.2**). In general, model-driven template languages heavily benefit from semantics-integrated metamodelling, because computed semantics can be reused within templates.
- Using EMFText to generate a text-to-model parser (**MM 5.3**).
- Using the generic, tree-based EMF model editor shipped with EMF that seamlessly integrates semantics in its properties view (**MM 5.4**). Thus, models can be interactively edited, their semantics browsed and semantic values manually changed, whereas dependend semantics are automatically updated.
- Using EMFText and EuGENia/GMF to generate advanced SiPLE and SiPLE-Statemachine editors with integrated semantics (**MM 5.5**).

Regarding metamodelling and semantics consistency (**MM 2**, **AG 2**) JastEMF's integration approach has to be evaluated from two perspectives: First, the individual consistency of the Ecore metamodel and the JastAdd specifications should and can be checked reusing their respective tooling. Second, in semantics integrated metamodelling the consistency of the mapping between syntax and semantics specifications (cf. Figure 3) has to be considered. As the JastEMF integration process (cf. Figure 4 $\boxed{\mathbf{2}}$) automatically derives a JastAdd AST specification from the Ecore metamodel, JastEMF provides such consistency for concepts in E_{syn_Ω}. However, JastEMF does not yet check the correctness of the semantics mapping (E_{sem_Ω}), which we like to improve in the future by an additional analysis step integrated in JastEMF.

Metamodel and model compatibility (**MM 4**) and semantics modularity (**AG 4**) both relate to extensibility, reuse and modularisation. In the EMF, metamodel and model compatibility helps to integrate *existing* languages, their tooling and models — i.e., to reuse *existing* metamodel implementations. Therefore, Ecore supports importing and referencing metaclasses and their implementation from other metamodels. For such reuse scenarios JastAdd has no appropriate mechanism. AST specifications can be combined from several specifications, but JastAdd always generates a new evaluator implementation and does not support the reuse of existing AST classes and their semantics. Consequently, reuse can only be achieved on the specification, but not implementation level. This limitation could be addressed by extending the JastAdd AST specification language with a packaging and package import concept that conforms to Ecore's metamodel imports[8].

On the other hand, JastAdd's aspect mechanism and weaving capabilities permit semantic extensions of languages by contributing new attributes (and node types) to an existing abstract syntax. As the EMF does not support the external contribution of structural features to existing metaclasses, the original metamodel needs to be changed to incorporate semantic extensions. This is a severe drawback considering incremental metamodel evolution that motivates further research on advancing modularity in future metamodelling approaches.

6.2 Limitations and Further Issues

JastAdd Rewrite Issues. JastAdd is not only a RAG system, but also supports local, constrained rewrites that are executed as soon as a node with an applicable rewrite is accessed. Within rewrites new nodes can be constructed and existing ones rearranged. We observed that in EMF such AST rearrangements in the combination with tree copies can lead to broken ASTs. Therefore, JastEMF does not support JastAdd rewrites.

Semantics of Incomplete Models. In dynamic environments such as the EMF, *syntactically* incomplete models are common throughout editor sessions. However, semantics of syntactically erroneous models are not defined and typically their evaluation fails with an exception. To our experience, most interactive modelling tools do not shield editor users before such exceptions. There is a need for more sensitive consideration of semantics in metamodelling frameworks and associated tooling, such that users are not disturbed by semantic exceptions caused by syntactically erroneous structures.

For future work, we consider the investigation of incremental AGs [23,24], which trace attribute dependencies to reduce the recomputation overhead in the presence of frequent context information changes, to address these issues. Metamodelling tools could use their attribute dependency knowledge to decide whether an attribute is defined or not. If an attribute is not defined — i.e., depends on missing model parts — its evaluation could be delayed to prevent it from throwing an exception.

In summary, we think, that JastEMF's benefits clearly outweigh the remaining technical problems. It demonstrates, that RAGs contribute declarativeness, well-foundness, generativeness and ease of specification for semantics-integrated metamodelling. On the other hand, metamodels and their accompanying frameworks provide convenient

[8] To by-pass the problem, we introduced simple helper properties and attributes in the SiPLE-Statemachines case study. The properties hold specified entry actions, guard expressions and transition actions as ordinary Java strings whereas the attributes initialise SiPLE's parser to transform these strings into appropriate SiPLE ASTs.

means to specify the API of AG generated tools and prepare their integration into software development platforms.

7 Related Work

There are a number of approaches related to and dealing with metamodel semantics. In particular we distinguish related work that (1) can benefit from our approach, like concrete syntax mapping tools and (2) propose alternative solutions, like constraint languages, integrated metamodelling environments, graph-grammars or abstract state machines. In the following we investigate each of them.

Textual concrete syntax mapping tools like EMFText [9] and MontiCore [10] combine existing parser generator technology with metamodelling technology to realize text-to-model parser generators [25]. They enable users to generate powerful text editors including features such as code completion and pretty printing. However, semantics analysis is often neglected or involves proprietary meachanims for implementing a subset of static semantics like name analysis manually. Such tools could immediately profit from our integration of formal semantics in metamodelling.

Constraint languages. like the OCL [16] or XCheck [7] enable the specification of well-formedness constraints on metamodels. The rationale behind OCL is to define invariants that check for context-sensitive well-formedness of models and to compute simple derived values. However we are not aware of any application of OCL for the specification of complete static semantics. In comparison AGs do not focus on constraint definitions, but are widely applied for semantics specification and provide advanced means to efficiently derive the context-sensitive information language constraints usually depend on[9].

Integrated metamodelling environments. provide dedicated languages to specify abstract syntax and semantics but often lack a formal background. Usually semantics have to be specified using a special constraint language and a special operational (i.e. imperative) programming language, both tightly integrated with a metamodelling language and its framework.

A typical representative is Kermeta [27]. A language in Kermeta is developed by specifying its abstract syntax with an Essential MOF (EMOF) metamodel and static semantics with OCL constraints. Execution semantics can be implemented using a third imperative programming language. Abstract syntax, static semantics and execution semantics are developed in modules that can be combined using Kermeta's aspect language. The modularization concept supported by Kermeta's aspect language seems very similar to the aspect concept of JastAdd: They both support the separation of crosscutting semantic concerns. Additionally, Kermeta and JastEMF projects immediately benefit from EMF tooling in Eclipse.

Our main concern about such integrated metamodelling environments is the overhead for developers to learn and apply all their different proprietary languages. We believe that JastAdd's seamless integration with Java has two main benefits: (1) one

[9] E.g., consider the specification of a data-flow analysis as presented in [26].

can rely on Java's standardised and well-known semantics and (2) the smooth learning curve from Java to declarative semantics specification reduces the initial effort for using JastAdd.

Graph-grammars are a convenient approach not only to specify structures — i.e. metamodels' abstract syntax — but also operations on these structures — i.e. metamodel semantics. Given such specifications, **graph rewrite systems** can be used to derive appropriate repository implementations and semantics [28]. The main advantages of the graph-grammar approach are its well-founded theoretical background and its uniform character where syntax and semantics can be specified within a single formalism.

PROGRESS. An important research project, that exploited graph-grammars for tool development and integration, has been the IPSEN project [28]. Its programming language PROGRESS (PROgramming with Graph REwriting Systems) supports the specification of graph schemas (i.e. metamodels), graph queries and graph transformations (i.e. semantics). PROGRESS graph schemas rely on attributed graph grammars to specify node attributes and their derived values. Though, attributed graph-grammars should not be confused with AGs. Attributed graph-grammars have no distinction between inherited and synthesized attributes and consequently lack many convenient AG concepts like broadcasting. In summary, the IPSEN project demonstrated, that graph-grammars are a convenient formalism to specify a broad range of tools and automatically integrate their repositories and semantics.

FUJABA. A more recent graph rewriting tool is FUJABA [29], which integrates Unified Modelling Language (UML) class diagrams and graph rewriting to specify semantics of class operations. It provides *story driven modelling* as a visual language to define rewrite rules, which can be compared to UML activity diagrams. MOFLON [30] adapts FUJABA to support the Meta Object Faclility (MOF) as a modelling language.

In general we think, that graph-rewriting systems are harder to understand than AGs. Given a set of rewrite rules, it is complicated to foresee all possible consequences of their application on start graphs. Rewrite results usually depend on the order of rule applications. To solve this problem, it is necessary to ensure that the rewrite system is confluent, which implies a lot of additional effort, not only for the proof of confluence, but also for the design of appropriate stratification rules. On the other hand, AGs require a basic context-free structure or a spanning tree they are defined on whereas graph rewriting does not rely on such assumptions. Furthermore, RAGs can only add information to an AST but not remove them or even change its structure. However, there are AG concepts such as higher order attributes [15] (*non-terminal* attributes in JastAdd) or JastAdd's local rewrite capabilities which improve in that direction.

Abstract State Machines (ASMs) are a theoretically backed approach to specify execution semantics [14]. For the specification of metamodel semantics they were recently applied in [31] to define sets of minimal modelling languages with well defined ASM semantics — so called *semantic units*. The semantics of an arbitrary modelling language L can now be defined by a mapping of L to such semantic units (*semantic anchoring*). Of course, the transformation to semantic units and context-sensitive well-formedness constraints (i.e., static semantics) still have to be defined using other approaches. Thus, ASMs and our RAG approach complement each other for the purpose to specify metamodels' execution *and* static semantics.

8 Conclusion

In this paper we presented the application of RAGs for metamodel semantics. We sketched necessary foundations — essentially that most metamodelling languages can be decomposed into context-free and context-sensitive language constructs — and presented JastEMF, an example integration of the EMF metamodelling framework and the JastAdd RAG system. Finally, we demonstrated and evaluated the advantages and limitations of our approach by a case study, which is exemplary for both compiler construction (SiPLE) and metamodelling (statemachines). This shows, that for MDSD the well-investigated formalism of RAGs is a valuable approach for specifying metamodel semantics and on the other hand, MDSD introduces interesting application areas and new challenges for RAG tools.

Acknowledgments

This research was funded by the *European Social Fond* in cooperation with the *Free State of Saxony* within the Ph.D. project *Innovative Werkzeugarchitekturen für die Erstellung von IT-Produktlinien im Bereich eingebetteter Software*, the *Deutsche Forschungsgemeinschaft* (DFG, German Research Foundation) within the project *Hyper-Adapt*, and the European Commission within the FP7 project MOST #216691.

We also like to thank Görel Hedin for her support and valuable comments, especially w.r.t. AGs and JastAdd.

References

1. Selic, B.: The Theory and Practice of Modeling Language Design for Model-Based Software Engineering – A Personal Perspective. In: Lämmel, R. (ed.) GTTSE 2009. LNCS, vol. 6491, pp. 290–321. Springer, Heidelberg (2011)
2. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF – Eclipse Modeling Framework, 2nd edn. Addison-Wesley, Reading (2008)
3. Hedin, G.: Reference Attributed Grammars. Informatica (Slovenia) 24(3), 301–317 (2000)
4. Knuth, D.E.: Semantics of Context-Free Languages. Theory of Computing Systems 2(2), 127–145 (1968)
5. Ekman, T., Hedin, G.: The JastAdd system — modular extensible compiler construction. Science of Computer Programming 69(1-3), 14–26 (2007)
6. ATLAS Group: ATLAS Transformation Language (ATL) User Guide (2006), `http://wiki.eclipse.org/ATL/User_Guide`
7. Efftinge, S., et al.: openArchitectureWare User Guide v.4.3.1 (2008), `http://www.openarchitectureware.org/pub/documentation/4.3.1/`
8. Oldevik, J.: MOFScript User Guide v.0.8 (2009), `http://www.eclipse.org/gmt/mofscript/doc/MOFScript-User-Guide-0.8.pdf`

9. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Derivation and Refinement of Textual Syntax for Models. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 114–129. Springer, Heidelberg (2009)
10. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: Modular Development of Textual Domain Specific Languages. In: TOOLS 2008 (Objects, Components, Models and Patterns). LNBIP, vol. 11, pp. 297–315. Springer, Heidelberg (2008)
11. Efftinge, S., Völter, M.: oAW xText: A framework for textual DSLs. In: Workshop on Modeling Symposium at Eclipse Summit (2006)
12. Kolovos, D.S.: Editing EMF models with Exeed (EXtended Emf EDitor) (2007), `http://www.eclipse.org/gmt/epsilon/doc/Exeed.pdf`
13. Ekman, T.: Extensible Compiler Construction. PhD thesis, University of Lund (2006)
14. Börger, E., Stark, R.F.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003)
15. Vogt, H.H., Swierstra, D., Kuiper, M.F.: Higher Order Attribute Grammars. In: PLDI 1989, pp. 131–145. ACM, New York (1989)
16. OMG: Object constraint language, version 2.2 (2010), `http://www.omg.org/spec/OCL/2.2/`
17. Farrow, R.: Automatic Generation of Fixed-Point-Finding Evaluators for Circular, but Well-defined, Attribute Grammars. In: SIGPLAN 1986, pp. 85–98. ACM, New York (1986)
18. Boyland, J.T.: Remote attribute grammars. Journal of the ACM 52(4), 627–687 (2005)
19. Magnusson, E.: Object-Oriented Declarative Program Analysis. PhD thesis, University of Lund (2007)
20. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Lee, S.H. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
21. Bürger, C., Karol, S.: Towards Attribute Grammars for Metamodel Semantics. Technical report, Technische Universität Dresden (2010) ISSN 1430-211X
22. Hedin, G.: Generating Language Tools with JastAdd. In: Lämmel, R. (ed.) GTTSE 2009. LNCS, vol. 6491. Springer, Heidelberg (2011)
23. Reps, T., Teitelbaum, T., Demers, A.: Incremental Context-Dependent Analysis for Language-Based Editors. ACM (TOPLAS) 5(3), 449–477 (1983)
24. Maddox III, W.H.: Incremental Static Semantic Analysis. PhD thesis, University of California at Berkeley (1997)
25. Goldschmidt, T., Becker, S., Uhl, A.: Classification of Concrete Textual Syntax Mapping Approaches. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 169–184. Springer, Heidelberg (2008)
26. Nilsson-Nyman, E., Hedin, G., Magnusson, E., Ekman, T.: Declarative Intraprocedural Flow Analysis of Java Source Code. Electron. Notes Theor. Comput. Sci. 238(5), 155–171 (2009)
27. Muller, P., Fleurey, F., Jézéquel, J.: Weaving Executability into Object-Oriented Metalanguages. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 264–278. Springer, Heidelberg (2005)
28. Nagl, M. (ed.): Building Tightly Integrated Software Development Environments: The IPSEN Approach. LNCS, vol. 1170. Springer, Heidelberg (1996)
29. Nickel, U., Niere, J., Zündorf, A.: The FUJABA environment. In: ICSE 2000, pp. 742–745. ACM, New York (2000)
30. Amelunxen, C., Königs, A., Rötschke, T., Schürr, A.: MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 361–375. Springer, Heidelberg (2006)
31. Chen, K., Sztipanovits, J., Abdelwalhed, S., Jackson, E.K.: Semantic Anchoring with Model Transformations. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 115–129. Springer, Heidelberg (2005)

Modelling GLL Parser Implementations

Adrian Johnstone and Elizabeth Scott

Royal Holloway, University of London, Egham, Surrey, TW20 0EX, UK
{a.johnstone,e.scott}@rhul.ac.uk

Abstract. We describe the development of space-efficient implementations of GLL parsers, and the process by which we refine a set-theoretic model of the algorithm into a practical parser generator that creates practical parsers. GLL parsers are recursive descent-like, in that the structure of the parser's code closely mirrors the grammar rules, and so grammars (and their parsers) may be debugged by tracing the running parser in a debugger. While GLL *recognisers* are straightforward to describe, full GLL parsers present technical traps and challenges for the unwary. In particular, naïve implementations based closely on the theoretical description of GLL can result in data structures that are not practical for grammars for real programming language grammars such as ANSI-C. We develop an equivalent formulation of the algorithm as a high-level set-theoretic model supported by table-based indices, in order to then explore a set of alternative implementations which trade space for time in ways which preserve the cubic bound.

Keywords: GLL parsing, general context-free parsing, implementation spaces, time-space tradeoffs.

1 The Interaction between Theory and Engineering

In Computer Science, the theoretician is mostly concerned with the correctness and asymptotic performance of algorithms whereas the software engineer demands 'adequate' time complexity on typical data coupled to memory requirements that do not cause excessive swapping. The theoretician's concerns are independent of implementation but the engineer's concerns are dominated by it and so the two communities do not always communicate well. As a result, our discipline has not yet achieved the comfortable symbiosis displayed by, for example, theoretical and experimental physicists.

The dominant characteristic of theoretically-oriented presentations of algorithms is *under specification.* It is fundamental practice for a theoretician to specify only as much as is required to prove the correctness of the results because this gives those results the widest possible generality, and thus applicability.

For the software engineer, under specification can be daunting: they must choose data structures that preserve the asymptotic performance demonstrated by the theoretical algorithm, and sometimes the performance expectations are only implicit in the theoretician's presentation. For instance, theoretical algorithms will often use sets as a fundamental data type. To achieve the lowest asymptotic bounds on performance the algorithm may need sets that have constant lookup time (which suggests an array based implementation) or sets whose contents may be iterated over in time proportional to their cardinality (which suggests a linked list style of organisation). The engineer may in fact be less concerned by the asymptotic performance than the average case performance on typical cases, and so a hash-table based approach might be appropriate. These implementation issues may critically determine the take up of a new technique because in reworking the algorithm to accommodate different data representations, the implementer may introduce effects that make the algorithm incorrect, slow or impractically memory intensive in subtle cases.

This paper is motivated by our direct experience of the difficulties encountered when migrating a theoretically attractive algorithm to a practical implementation even within our own research group, and then the further difficulties of communicating the results and rationale for that engineering process to collaborators and users.

The main focus of this paper is a modelling case study of our GLL generalised parsing algorithm [6] which yields cubic time parsers for all context free grammars. Elsewhere we have presented proofs of correctness and asymptotic bounds on performance along with preliminary results that show excellent average case performance on programming language grammars. It is clear, however, that the theoretical presentations have proved somewhat difficult for software engineers, who may find the notation opaque or some of the notions alien, and who may miss some of the critical assumptions concerning data structures which are required to have constant lookup time. Direct implementation of these structures consumes cubic memory and thus more subtle alternatives are required. In this paper we shall explicitly address the motivation for our choice of high level data structures, and explain how we migrate a naïve version to a production version by successive refinement. Our goal is to describe at the meta-level the process by which we refine algorithm implementations.

We view this as a modelling process. Much of the model-driven engineering literature is concerned with programming in the large, that is the composition of complete systems from specifications at a level of abstraction well away from the implementation platform, potentially allowing significant interworking and reuse of disparate programming resources. This paper is focused on programming in the small. We use a specification language that avoids implementation details of the main data structures, and then use application specific measures to refine the specification into an implementation with optimal space-time tradeoff. We do this in a way that lends itself to automation, holding out the prospect of (semi-)automatic exploration of the implementation space. It is worth investing this effort because we are optimising a *meta*-program: our GLL parser generator generates GLL parsers, and every parser user will benefit from optimisations that we identify.

Our models are written in LC, a small object-oriented language with an extremely simple type system based on enumerations and tables. Our goal is to develop a notation that is comfortable for theoretical work from which implementations may be directly generated, and which also allows tight specification of memory idioms so that the generated implementations can be tuned using the same techniques that we presently implement manually, but with much reduced clerical overhead.

We begin with a discursive overview of GLL and then describe some aspects of the LC language. We give an example GLL parser written in LC and explain its operation. We then show how memory consumption may be significantly reduced without incurring heavy performance penalties. We finish by discussing the potential to semi-automatically explore the space of refined implementations in search of a good time-space trade off for particular kinds of inputs.

2 General Context Free Parsing and the GLL Algorithm

Parsing is possibly one of the most well studied problems in computer science because of the very broad applicability of parsers, and because formal language theory offers deep insights onto the nature of the computational process. Translators such as compilers take a program written in a source (high level) language and output a program with the same meaning written in a target (machine) language. The syntax of the source language is typically specified using a context free grammar and the meaning, or semantic specification, of a language is typically built on the syntax. Thus the first stage of compilation is, given an input program, to establish its syntactic structure. This process is well understood and there exist linear algorithms for large classes of grammars and cubic algorithms that are applicable to all context free grammars.

Formally, a *context free grammar* (CFG) consists of a set $\mathbf{N}$ of non-terminal symbols, a set $\mathbf{T}$ of terminal symbols, an element $S \in \mathbf{N}$ called the start symbol, and a set of grammar rules of the form $A ::= \alpha$ where $A \in \mathbf{N}$ and α is a string in $(\mathbf{T} \cup \mathbf{N})^*$. The symbol ϵ denotes the empty string, and in our implementations we will use # to denote ϵ. We often compose rules with the same left hand sides into a single rule using the alternation symbol, $A ::= \alpha_1 \mid \ldots \mid \alpha_t$. We refer to the strings α_j as the *alternates of* A.

We use a grammar Γ to define a language which is a set of strings of terminals. We do this by starting with the start symbol and repeatedly replacing a nonterminal with one of its alternates until a string containing no nonterminals is obtained. A *derivation step* is an expansion $\gamma A \beta \Rightarrow \gamma \alpha \beta$ where $\gamma, \beta \in (\mathbf{T} \cup \mathbf{N})^*$ and $A ::= \alpha$ is a grammar rule. A *derivation* of τ from σ is a sequence $\sigma \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \ldots \Rightarrow \beta_{n-1} \Rightarrow \tau$, also written $\sigma \overset{*}{\Rightarrow} \tau$ or, if $n > 0$, $\sigma \overset{+}{\Rightarrow} \tau$. The language defined by a CFG Γ is the set of all strings u of terminals which can be derived from the start symbol, S, of Γ. Parsing is the process of determining, given a string u, some or all of the derivations $S \overset{*}{\Rightarrow} u$.

Of the linear parsing techniques perhaps the most widely used is the LR-table driven stack based parser [3]. For the class of grammars which admit LR-parsers the technique is straightforward to implement. However, the class of grammars does not include any grammars for 'real' programming languages and as a result implementations 'modify' the technique to extend its applicability. It can be hard to reason about the correctness or the subtle behaviour of the resulting implementation.

An alternative is to use a general technique such as Earley[2], CYK[10] or GLR[8], [5]. In worst case Earley algorithms are cubic, CYK requires the grammar to be rewritten in 2-form and standard GLR algorithms are unbounded polynomial order, although the typical performance of GLR algorithms approaches linear and a cubic version has been developed [7]. These general algorithms are used in the natural language community but have had relatively limited take up within mainstream computer science. This is, at least to some extent, because they are hard for many implementers to understand and their implementation needs to be done with a great deal of care to get acceptable space and runtime performance. For example, ASF+SDF [9] uses Farshi's version of GLR [4] which achieves correctness in a rather brute force way and hence acquires a performance cost, and Bison includes a GLR mode [1] which does not employ the full details of the GLR approach and hence cannot parse an input of length 20 for some highly ambiguous grammars.

Recently we have introduced the general GLL parsing technique [6], which is based on the linear recursive descent paradigm. Recursive descent (RD) is an attractive technique because the parser's source program bears a close relationship to the grammar of the language and hence is easy to reason about, for instance by tracing the code in a debugger. However, the class of grammars to which RD can be applied is very limited and many extensions have been implemented which use either full or limited backtracking. Full backtracking can result in exponential runtime and space requirements on some grammars and limited backtracking will fail to parse some grammars correctly. GLL models full backtracking by maintaining multiple process threads. The worst-case cubic complexity is achieved by using a Tomita-style *graph structured stack* (GSS) to handle the function call stacks and left recursion (a fundamental problem for RD parsers) is handled with loops in this graph. (A non-terminal A is *left recursive* if there is a string μ such that $A \overset{+}{\Rightarrow} A\mu$, and a grammar is left recursive if it has a left recursive nonterminal.)

The GLL technique is based on the idea of traversing the grammar, Γ, using an input string, u, and we have two pointers one into Γ and one into u. We define a *grammar slot* to be a position immediately before or after any symbol in any alternate. These slots closely resemble LR(0) items and we use similar notation, $X ::= x_1 \dots x_i \cdot x_{i+1} \dots x_q$ denotes the slot before the symbol x_{i+1}. The grammar pointer points to a grammar slot. The input pointer points to a position immediately before a symbol in the input string. For $u = x_1 \dots x_p$, i is the position immediately before x_{i+1} and p is the position immediately before the end-of-string symbol, which we denote by \$.

A grammar is traversed by moving the grammar pointer through the grammar. At each stage the grammar pointer will be a slot of the form $X ::= \alpha \cdot x\beta$ or $X ::= \alpha\cdot$ and the input pointer will be an input position, i. There are then four possible cases: (i) If $x = a_{i+1}$ the grammar pointer is moved to the slot $X ::= \alpha x \cdot \beta$ and the input pointer is moved to $i + 1$. (ii) If x is a nonterminal

A then the 'return' slot, $X ::= \alpha x \cdot \beta$ is pushed onto a stack, the grammar pointer is moved to some slot of the form $A ::= \cdot\gamma$ and the input pointer is unchanged. (iii) If the grammar pointer is $X ::= \alpha\cdot$ and the stack is nonempty, the slot $Y ::= \delta X \cdot \mu$ which will be on the top of the stack is popped and this becomes the grammar pointer. (iv) Otherwise if grammar pointer is of the form $S ::= \tau\cdot$ and the input pointer is at the end of the input a successful traversal is recorded, else the traversal is terminated in failure. Initially the grammar pointer is positioned at a slot $S ::= \cdot\alpha$, where S is the start symbol, and the input pointer is 0.

Of course we have not said how the slot $A ::= \cdot\gamma$ in case (ii) is selected. In the most general case this choice is fully nondeterministic and there can be infinitely many different traversals for a given input string. We can reduce the nondeterminism significantly using what we call *selector sets*. For a string α and start symbol S we define $\text{FIRST}_{\mathbf{T}}(\alpha) = \{t|\alpha\overset{*}{\Rightarrow}t\alpha'\}$, and $\text{FIRST}(\alpha) = \text{FIRST}_{\mathbf{T}}(\alpha) \cup \{\epsilon\}$ if $\alpha\overset{*}{\Rightarrow}\epsilon$ and $\text{FIRST}(\alpha) = \text{FIRST}_{\mathbf{T}}(\alpha)$ otherwise. We also define $\text{FOLLOW}(\alpha) = \{t|S\overset{*}{\Rightarrow}\tau\alpha t\mu\}$ if $\alpha \neq S$ and $\text{FOLLOW}(S) = \{t|S\overset{*}{\Rightarrow}\tau St\mu\} \cup \{\$\}$.

Then for any slot $X ::= \alpha \cdot \beta$ we define $select(X ::= \alpha \cdot \beta)$ to be the union of the sets $\text{FIRST}(\beta x)$, for each $x \in \text{FOLLOW}(X)$, and we can modify the traversal case (ii) above to say (ii) If x is a nonterminal A and there is a grammar slot $A ::= \cdot\gamma$ where $a_i \in select(\gamma)$ then the 'return' slot, $X ::= \alpha x \cdot \beta$ is pushed onto a stack, the grammar pointer is moved to the slot $A ::= \cdot\gamma$ and the input pointer is unchanged. The initial grammar pointer is also set to a slot $S ::= \cdot\alpha$ where $a_0 \in select(A ::= \cdot\alpha)$.

Whilst the use of selector sets can significantly reduce the number of possible choices at step (ii), in general there will still be more than one qualifying slot $A ::= \cdot\gamma$ and, in some cases, infinitely many traversals. GLL is a technique designed to cope with this in worst-case cubic time and space.

At step (ii), instead of continuing the traversal each possible continuation path is recorded in a *context descriptor* and ultimately pursued. We would expect a descriptor to contain a slot, an input position and a stack. Then, at step (ii), for each slot $A ::= \cdot\gamma$ such that $a_i \in select(A ::= \cdot\gamma)$ a descriptor is created with that slot, the current stack onto which the return slot is pushed and input position i. A descriptor (L, s, i) is 'processed' by restarting the traversal with the grammar pointer at the slot L, s as the stack and input pointer at i. There are potentially infinitely many descriptors for a given input string because there are potentially infinitely many stacks. The solution, introduced by Tomita for his initial version of the GLR technique, is to combine all the stacks into a single graph structure, merging the lower portions of stacks where they are identical and recombining stack tops when possible. At the heart of the GLL technique are functions for building this graph of merged stacks, which we call the GSS. Instead of a full stack, descriptors then contain a node of the GSS which corresponds to the top of its associated stack(s), thus one descriptor can record several partial traversals provided they restart at the same grammar and input positions.

So far we have addressed only the recognition of a string; we want to capture the syntactic structure to pass on to later stages in the translation process. A common method for displaying syntactic structure is to use a *derivation tree*: an ordered tree whose root is labelled with the start symbol, leaf nodes are labelled with a terminal or ϵ and interior nodes are labelled with a non-terminal, A say, and have a sequence of children corresponding to the symbols in an alternate of A. This is a derivation tree for $a_1 \dots a_n$ if the leaf nodes are labelled, from left to right, with the symbols $a_1, \dots, a_n$ or ϵ. The problem is that for ambiguous grammars one string may have very many different syntactic structures and so any efficient parsing technique must use an efficient method for representing these. The method used by GLR parsers is to build a *shared packed parse forest* (SPPF) from the set of derivation trees. In an SPPF, nodes from different derivation trees which have the same tree below them are shared and nodes which correspond to different derivations of the same substring from the same non-terminal are combined by creating a packed node for each family of children. The size of such an SPPF is worst-case unbounded polynomial, thus any parsing technique which produces this type of output will have at least unbounded polynomial space and time requirements. The GLL technique uses SPPF building functions that construct a binarised form of SPPF which contains additional *intermediate* nodes. These nodes group the children of packed nodes in pairs from the left so that the out degree of a packed node is at most two. This is sufficient to guarantee the SPPF is worst-case cubic size. In detail, a binarised SPPF has three types of nodes: symbol nodes, with labels of the form (x, j, i) where x is a terminal, nonterminal or ϵ and $0 \leq j \leq i \leq n$; intermediate nodes, with labels of the form (t, j, i); and packed nodes, with labels for the form (t, k), where $0 \leq k \leq n$ and t is a grammar slot. We shall call (j, i) the *extent* (j, i are the left and right extents respectively) of the SPPF symbol or intermediate node and k the *pivot* of the packed node.

3 The LC Specification Language

LC is a small object oriented language which provides only a single primitive data type (the enumeration) and a single data structuring mechanism (the table). LC is designed to allow high-level descriptions of set-theory based algorithms whilst also allowing quite fine grained specification of the implementation in terms of the way the algorithm's objects are to be represented in memory. In this respect, LC is an unusual language with elements of both high level specification languages and very low level assembler-like languages. At present, LC is a (mostly) paper notation which we use here to describe data structure refinements. Our intention is that an LC processor will be constructed which can generate executable programs written in C++, Java and so on as well as LaTeX renderings of our algorithms in the style of [6]. (We note in passing that LC's syntax is hard to parse with traditional deterministic parsers, so in fact LC itself needs a GLL or other general parser). In this section we describe the type system of LC along with a few examples of sugared operations and control flow sufficient to understand the description of the GLL parser below.

Lexical conventions. An LC program is a sequence of *tags* and reserved symbols, notionally separated by white space. A tag is analogous to an identifier in a conventional programming language except that any printable character other than the nine reserved characters `( ) [ ] , | : "` and comments delimited by `(* *)` may appear in a tag. Hence $\boxed{\texttt{adrian}}$, $\boxed{\texttt{10}}$ and $\boxed{\texttt{-3.5}}$ are all valid tags. Where no ambiguity results, the whitespace between tags may be omitted. Character strings are delimited by `"` and may include most of the ANSI-C escape sequences.

Primitive types. The LC primitive data type generator is the enumeration which maps a sequence of symbols onto the natural numbers and thus into internal machine integers. The enumeration is a `|`-delimited list with a prepended tag which is the type name. A boolean type might be defined as `Bool ( false | true )`. Literals of this type are written `Bool::false` and `Bool::true`. Where no ambiguity results, the tag may be written simply as the tag `true` or `false`. Methods may be defined by enumerating the appropriate finite map from the set of input parameters to the output values, so to define a boolean type that contains a logical AND operation we write:

```
Bool ( false | true
       Bool &(Bool, Bool) := ( (false, false),
                                (false, true)   )  )
```

Where no ambiguity results, methods may be invoked using infix notation.

Every primitive type has an extra value `()` read as *empty*. Newly declared variables are initialised to `()` and variables may be 'emptied' by assigning `()` to them.

We can declare some integer types as

```
Int3 ( -4 | -3 | -2 | -1 | 0 | 1 | 2 | 3 )
Unsigned3 ( 0 | 1 | 2 | 3 | 5 | 6 | 7 )
```

We shall assume the existence of a type `Int` which has been defined in this way and which contains sufficient values to allow our computations to proceed without overflow and the usual arithmetic operations.

For each enumeration, the first element maps to 0, the second to 1 and so on. We use $|x|$ to denote the cardinality of set x and $||x||$ to represent the memory space required to represent an element of x. If $|T|$ is the number of explicitly declared enumeration elements then there are $|T| + 1$ elements in the type (allowing for the extra value $\boxed{\texttt{()}}$) and so a value of type T occupies at least $||t|| = \lceil \log_2(|T| + 1) \rceil$ binary digits in memory.

Enumerations may be composed: `BothBoolAndUnsigned3 (Bool | Unsigned3)` is a shorthand for

```
BothBoolAndUnsigned3 ( false | true | 0 | 1 | 2 | 3 | 5 | 6 | 7 )
```

It is an error to compose enumerations which share tag values.

Tables. Each variable in LC is a table with zero or more dimensions each of which has an index set defined by an LC enumeration. So, `Bool x(Int3)(Bool)` declares a two dimensional array of 16 boolean variables, called `x(-3)(false)`, `x(-3)(true)`, `x(-2)(false)`, ..., `x(3)(true)`. We may write `Bool x(Int3, Bool)` as a shorthand declaration.

Representing types with LC tables. Since LC's type system is so simple, it is reasonable to wonder whether it is sufficient. It clearly is complete in fundamental engineering terms because nearly all modern computers use a single virtual address space of binary memory locations corresponding to a one dimensional LC table. All data structures ultimately are mapped to this representation. Hoare's classic survey of datastructuring techniques listed five mechanisms by which primitive types are combined: Cartesian product, discriminated union, map, powerset and recursive data type. LC provides each of these: a comma-delimited list of type and field names, for example `CartProd (Int3 i, Bool b)`, denotes a Cartesian product and corresponds to a record structure in Pascal or a `struct` in C named `CartProd` with two fields named `i` and `b`; a `|`-delimited list of type and field names, for example `DiscUnion (Int3 i | Bool b)`, denotes a discriminated union and plays a similar role to a C `union`; maps are directly represented by tables of functions; powersets are represented by tables indexed by the type of the powerset whose cells are either empty or contain the element of the unusual enumeration type `Set (isMember)`; and recursive types by (impractically) large tables: for instance the edges of a graph of nodes containing a `CartProd` field is specified as `Set g(CartProd, CartProd)`. The process by which such extensive tables is implemented is described in a later section.

Assignments. In LC, assignment is central. Simple assignment is written `x := 2`. Structurally type compatible assignments may be done is a single statement as in `(Int3 x, Bool y, Unsigned3 z) := (3, true, 3)`. We provide some higher level assignment operations which are used as hints by the datastructure refinement stage:

> `x addTo s` is shorthand for `s(x) := isMember`
> `x deleteFrom s` is shorthand for `s(x) := ()`
> `y selectDelete s` is shorthand for: nondeterministically select an index `i` of an occupied cell in `s` then execute `y := s(i) s(i) := ()`.
> `y selectNewestDelete s` is like `selectDelete` except that the most recently assigned cell is guaranteed to be selected (leading to stack-like behaviour)
> `y selectOldestDelete s` is like `selectDelete` except that the occupied cell whose contents has been unchanged the longest is guaranteed to be selected, leading to queue-like behaviour.

Control flow. An LC label may appear before expressions or statements, labels are denoted by a tag followed by a colon. Labels may be assigned to variables and passed as parameters; their type is `CodeLabel` and there is an implicit definition of an enumeration comprising all of the labels in a program in the order in which they are declared. Control may be transferred to a literal label or the contents of a `CodeLabel` variable with the `goto` statement. LC also includes the usual `if` and `while` statements. LC provides some syntactically sugared predicates such as `x in s` which is shorthand for `s(x) != ()`, i.e. that the cell in `s` indexed by `x` is non-empty.

The `for` statement provides a variety of higher level iteration idioms.

> `for x in T` and `for x in Y` each execute once for each member of the enumeration in type T or non-empty member of table `Y` respectively; on each iteration `x` will have a different element but there is no guarantee of the order in which elements are used.
> `for x over T` and `for x over Y` each execute once for each member of the enumeration in type T or non-empty member of table `Y` respectively, with the elements being used in the order in which they are declared in `T` or, respectively, `Y`. In the case of a table, where the index is a tuple, the rightmost element varies most rapidly.

Text strings and output. LC strings are delimited by `"` and accept most ANSI-C escape sequences. An LC program can produce textual output via a method `write()` which, by analogy with ANSI-C's `printf()` function takes a string with embedded place holders %. No type need be supplied, since LC values carry their class with them, but most ANSI-C formatting conventions are supported.

4 An Example GLL Parser

In this section we discuss an LC GLL parser for the grammar

$$S ::= A\ S\ b \mid \epsilon \qquad\qquad A ::= a$$

In the listing below, data types are declared in lines 1–19 and variables in lines 20–30. The GLL parser body is in lines 42–71 and the support routines (which are grammar independent) are appended in lines 73–111. The dispatcher, which dictates the order in which contexts are computed is at lines 35–40.

There are primitive types `GSSLabel`, `SPPFLabel` and `ContextLabel` whose elements are certain grammar slots together with, in the case of `SPPFLabel`, the grammar terminals and nonterminals, and `#` (epsilon). There are explicit maps `contextLabel`, `codeLabel`, and `sppfLabel` from `GSSLabel` to `contextLabel` and from `contextLabel` to `CodeLabel` and `sppfLabel`. We also define the selector sets for each grammar slot, and *abort* sets which are the complements of the selector sets. The maps `isSlot1NotEnd` and `isSlotEnd` take a `ContexLabel` and return a Boolean. The former returns true if the corresponding slot is of the form $X ::= x \cdot \alpha$ where x is a terminal or nonterminal and $\alpha \neq \epsilon$, and the latter returns true if the corresponding slot is of the form $X ::= \gamma\cdot$. The map `lhsSlot` takes a ContextLabel and returns the left hand side nonterminal of the corresponding slot. For readability we have left the explicit definitions of all these types out of the listing.

The methods `findSPPFSymbolNode`, `findSPPFPackNode` and `findGGSNode` return a node with the specified input attributes, making one if it does not already exist. Their definitions have also been omitted so as not to pre-empt the data structure implementation discussion presented in the later sections of this paper.

```
Set (isMember)
N ( A | S )
T ( a | b )
Lexeme (T | EndOfString)
GSSLabel (...)
SPPFLabel (...)
ContextLabel (...)
CodeLabel codeLabel(ContextLabel) := (...)
ContextLabel contextLabel(GSSLabel) := (...)
SPPFLabel sppfLabel(ContextLabel) := (...)
Set abortSet... := (...)
Set selectorSet... := (...)
GSSNode (GSSLabel s, Int i)
GSS (Set GSSEdge (GSSNode src, SPPFSymbolNode w, GSSNode dst))
SPPFSymbolNode (SPPFLabel s, Int leftExtent, Int rightExtent)
SPPFPackNode (SPPFLabel s, Int pivot)
SPPF (SPPFSymbolNode symbolNode, SPPFPackNode packNode,
      SPPFSymbolNode left, SPPFSymbolNode right)
Context (Int i, ContextLabel s, GSSNode u, SPPFSymbolNode w)
GSS gss
SPPF sppf
Context c_C                (* Current context *)
Int c_I                    (* Current input pointer *)
GSSNode c_U                (* Current GSS node *)
SPPFSymbolNode c_N         (* Current SPPF node *)
SPPFSymbolNode c_R         (* Current right sibling SPPF node*)
GSSnode u_0                (* GSS base node *)
Set U(Context)             (* Set of contexts encountered so far *)
Set R(Context)             (* Set of contexts awaiting execution *)
Set P(GSSNode g SPPFSymbolNode p) (* Set of potentially unfinished pops *)
gll_S(
  SPPF parse(Lexeme I(Int))(
    goto L_S

    L_Dispatch:
    c_C selectDelete R
    if c_C = () return sppf
    c_I := c_C(i)
    c_N := c_C(w)
    goto codeLabel(c_C(s))

    L_A:
      if I(i) in selectorSet_A_1 addContext(A_1, c_U, c_I, ())
      goto L_Dispatch

    L_A_1:
      c_R := (a, c_I, c_I + 1) c_N := getSPPFNodeP(A_1_1, c_N, c_R) c_I := c_I + 1
      pop
      goto L_Dispatch

    L_S:
      if I(i) in selectorSet_S_1 addContext(S_1, c_U, c_I, ())
      if I(i) in selectorSet_S_2 addContext(S_2, c_U, c_I, ())
      goto L_Dispatch

    L_S_1:
      c_U := updateGSS(S_1_1) goto L_A
    L_S_1_1:
      if I(c_I) in abortSetS_1_2 goto L_Dispatch
      c_U := updateGSS(S_1_2) goto L_S
    L_S_1_2:
      if I(c_I) in abortSetS_1_3 goto L_Dispatch
      c_R := (b, c_I, c_I + 1) c_N := getSPPFNodeP(S_1_3, c_N, c_R) c_I := c_I + 1
      pop
      goto L_Dispatch

    L_S_2:
      c_R := (#, c_I, c_I) c_N := getSPPFNodeP(S_2_1, c_N, c_R)
      pop
      goto L_Dispatch
  )

  Void addContext(ContextLabel s, GSSNode u, Int i, SPPFSymbolNode w)
    if (s, u, w) not in U(i) (
      (s, u, w) addto U(i)
      (s, u, w) addto R(i)
  )

  Void pop() (
    if c_U = u_0 return
    (c_U, c_N) addto P
    for (w, u) in gss(c_U)
      addContext(contextLabel(c_U(s)), u, c_I,
                 getSPPFNodeP(contextLabel(c_U(s)), w, c_N))
  )

  GSSNode updateGSS(ContextLabel s) (
    w := c_N
    v := findGSSNode(s, c_I)
    if (v, w, c_U) not in gss (
      (v , w, c_U) addto gss
      for z in P(v) addContext(s, c_U, z(rightExtent), getSPPFNodeP(s, w, z))
    )
    return v
  )

  SPPFSymbolNode getSPPFNodeP(ContextLabel s, SPPFSymbolNode z, SPPFSymbolNode w) (
    if isSlot1NotEnd(s) return w
    SPPFLabel t
    if isEndSlot(s) t := lhsSlot(s) else t := sppfLabel(s)
    ( , k, i)  := w
    if z != SPPFSymbolNode::()  (
      y := findSPPFSymbolNode(t, z(leftExtent), i)
      findSPPFPackNode(y, s, k, z, w)
    )
    else (
      y := findSPPFSymbolNode(t, k, i)
      findSPPFPackNode(y, s, k, (), w)
    )
    return y
  )
)
```

5 The Impact of Language Size

Programming languages can vary by factor of seven or more in the cardinality of key enumerations for GLL parsers. Table 1 shows the size of the GSSLabel, SPPFLabel and ContextLabel enumerations for a range of languages. In each case we have used the most authoritative available grammar: language standards for ANSI-C, Pascal and C++; the original report for Oberon and the VS COBOL II grammar recovered from IBM documentation by Ralf Lämmel and Chris Verhoef.

Table 1. GLL measures for programming languages

Grammar	Enumeration extents			Ring	Sets		
	GSS	SPPF	context	length	Defined	Unique	Saving
Oberon 1990	240	645	481	3	442	204	54%
C ANSI X3.159-1989	277	665	505	5	507	176	65%
Pascal: ISO/IEC 7185:1990	393	918	755	3	626	234	63%
Java JLS1.0 1996	384	949	755	4	668	227	66%
C++ ISO/IEC 14882:1998	722	1619	1287	4	1351	294	78%
COBOL (SDF)	1767	4530	3702	5	3502	863	75%

We also show the grammar's *ring length*. This is the longest sequence of terminals present in any alternate in the grammar. It turns out that the dimensions of some tables in a GLL parser may be reduced in size from O(the length of the input string) to O(ring length).

Finally, we show the effect of *set merging*. The number of defined sets is the number of unique sets referenced by a GLL parser: that is the selector sets and the abort sets. It turns out that many of these sets have the same contents, so we can alias names together and only store one table of `isMember` for two or more set names.

6 Process Management in GLL

The description of the GLL technique in Section 2 is essentially declarative. Here we focus on possible implementations of the control-flow in GLL parsing. The heart of GLL parsing from an operational point of view is the task scheduler. GLL contexts (line 11) comprise a `CodeLabel L`, at which to resume execution, and the input pointer, GSS and SPPF nodes that were current at the time the context is created. Each of these specifies an instance of the parser process. Whenever a GLL parser encounters potential multiple control flow paths it creates a process context for each path and then terminates the current process. This happens in two places: (i) whilst processing a nonterminal when the selected productions are added as new processes and (ii) when rules which called a nonterminal are restarted after a pop action (in either the `pop` or `updateGSS` functions).

Now, what is to stop the number of processes growing without limit? The key observation is that our parsing process is *context free*, and this means that all instances of a nonterminal matching a particular substring are equivalent, or to put it another way, if a nonterminal A has already been matched against a particular substring α then we do not have to rerun the parse function. Instead, we merely need to locate the relevant piece of SPPF and connect the SPPF node currently under construction to it. As result, each GLL context created within a run of the parser need execute only once. To ensure this, we maintain a set of contexts that have been seen on a parse `U` along with a set `R` which holds the subset of `U` which currently awaits processing.

Whenever a process terminates, either because it reached the end of a production or because the current input lexeme is invalid, control returns to `L_Dispatch` where a new process is scheduled for execution. In our example, processes are removed non-deterministically (line 36). If we force stack removal by changing the `selectDelete` operator to `selectNewestDelete` then we can simulate the behaviour of a traditional depth-first RD parser except that GLL accepts left recursive grammars. This can be very useful for debugging grammars. It turns out that if we force queue removal using `selectOldestDelete` then we can make significant memory savings, and reduce the maximum size of `R`.

7 Modelling GLL Data Structure Refinement

So far, our LC programs have been based on the notion of a flexible tables which can change their size and even their dimensionality as required. From the theoretician's point of view, we assume that the tables are 'large enough'. Table elements may be accessed in unit time, and on the occasions that we need to iterate over the contents of a table in time proportional to the number of used elements, we assume that a parallel linked list has been constructed in tandem with the table. This model (which we call Big-Fast-Simple or BFS) is sufficient to reason about the asymptotic space and time performance of an algorithm, but is likely to be over-simplistic for real problems, and indeed that is the case for GLL. It turns out that directly implementing GLL data structures such as the SPPF and GSS as arrays is practical for small examples, and is in fact very useful for experimenting with pathological highly ambiguous grammars because we get maximum speed that way, but for realistic inputs to parsers for even small languages such as ANSI-C the direct implementation requires huge memory.

We shall now describe the procedure we use to optimise the space required by our data structures with compromising asymptotic behaviour.

7.1 Address Calculation and Pointer Hopping

We might attempt to implement an LC table as a straightforward multi-dimensional array, or as a sparse array made up of a tree of lists, or as some sort of hybrid. An LC table, like any other kind of data structure, is a container for other data structures and primitive data types; any instance of a type is ultimately just a collection of filled cells along with access paths to each cell, expressed in LC's case as tuples of indexing expressions.

There are essentially only two mechanisms from which to construct access paths: pointer hopping (i.e. linked data structures based on cells which contain the names of other cells) and direct address calculation in which the access path is a computation over the index expressions and some constants. The distinguishing feature, then, is that with address calculation, a data cell's address is a function of its index expression only, but with linking a data cell's address is a function of its index expression and the contents of at least one other cell in the data type.

Multi-dimensional arrays are the most common example of data structures accessed solely by address calculation, but we include hash tables in this category too. Space precludes consideration of hash tables in this paper although we shall introduce our notation for refining LC tables to hash table implementations.

We can characterise the two styles of implementation in terms of their impact on (i) space, (ii) access time for a particular index value, and (iii) iteration time, i.e. the time taken to access all elements.

Consider a one dimensional table x of type Y indexed by type T in which $d \leq |T|$ elements are utilised, the others being set to `()`. So, for example, this LC declaration `Y x(T) := ( (), u1, u1 )` creates a table in which d is 2 since only the second and third elements are in use.

- For a linked implementation, the space required is $O(d)$, the time to access a particular element indexed by $t \in T$ is $O(d)$ and the time taken to iterate over all elements is also $O(d)$.
- For an implementation based on address calculation, the space required is $O(|T|)$, a particular element indexed by $t \in T$ can be accessed in constant time and the time taken to iterate over all elements is $O(|T|)$.

We *refine* an LC table definition by annotating the dimensions to specify either linking (indicated by parentheses) or address calculation indicated by square brackets. We call these possibilities the *dimension modes*.

For a two dimensional table `U x(T,S)` indexed by types $T(1|2|3|4)$ and $S(a|b|c|d)$ we might choose

1. `U x( (T), (S) )` a fully linked representation (using a list of lists),
2. `U x( [T,   S] )` a two dimensional address calculation,
3. `U x( [T], [S] )` two one-dimensional (vector) address calculations
4. `U x( [T], (S) )` a vector of linked lists, or
5. `U x( (T), [S] )` a linked lists of vectors.

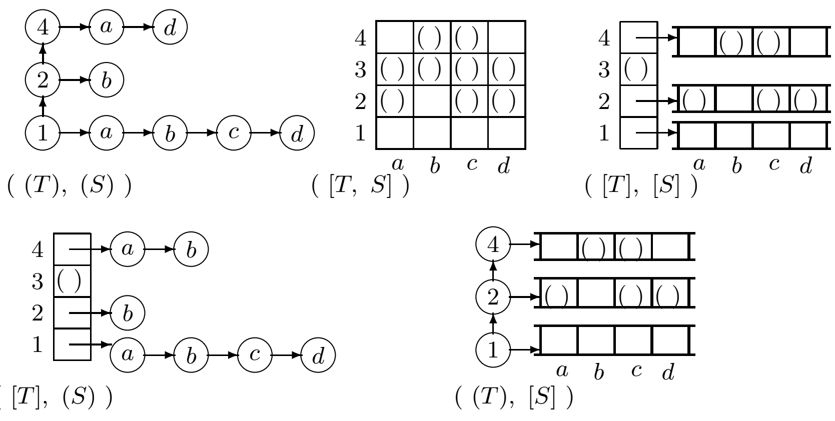


Fig. 1. Table refinement for 2-D structures

These cases are illustrated in Figure 1 for the case of an 2D array indexed by unsigned two-bit integers containing the following ordered pairs: $(1, a)$,$(1, b)$,$(1, c)$, $(1, d)$, $(2, b)$, $(4, a)$ and $(4, d)$.

For multi-dimensional tables implemented as anything other than a full array, the dimension modes of a table are not the only things that affect performance since both the size of a table and its access time for particular elements can be dependent on the *ordering* of the dimensions. We should emphasise that this is purely an implementation matter: the semantics of a table do not change if one permutes the indices in its declaration as long as the indices in accesses to that table are changed to match. This means that we have an opportunity to improve performance through refinement without affecting the analysis of the algorithm as specified in its original, unrefined, form.

Consider the set of ordered pairs $(1, a), (3, a), (4, a)$ and $(4, b)$. The leftmost dimension uses three distinct values, but the rightmost dimension uses only two. If we use the list-of-lists organisation, indexing as `((T), (S))` we can have two structures, one listing the leftmost dimension first and the other the rightmost:

$$\begin{tikzcd}
4 \arrow[r] & a \arrow[r] & b \\
3 \arrow[u] \arrow[r] & a & \\
1 \arrow[u] \arrow[r] & a &
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
b \arrow[r] & 4 & & \\
a \arrow[u] \arrow[r] & 1 \arrow[r] & 3 \arrow[r] & 4
\end{tikzcd}$$

The leftmost-first table requires more space than the rightmost. The driver for this is the *utilisation count* for a dimension of a table. If we can order a table so that, on average, the dimensions with the lowest utilisation counts are used first, then we will on average reduce the size of the table. This effect, in which using the dimension with the highest utilisation count first increases the overall size of the structure can have a very significant effect on memory consumption. Consider the case of a table `x([Unsigned16], [Unsigned16])`. This will comprise vectors of length 2^{16} elements. Let us imagine an extreme case in which only one row of this table is in use, i.e. that we are using cells `x(0,0)`, `x(1,0)`, ... `x(65535, 0)`. If we use the leftmost dimension as the first (column) index, then we need 65 536 row vectors within which only one element is in use. If, on the other hand, we use the rightmost dimension for the column vector then we need only one row vector, all elements of which are in use. We can see that the space allocation for a table indexed as `([T], [T])` can vary between $2|T|$ and $|T|^2$.

Access time can be affected too: if we use a `([Int], (Int))` indexing style then the arrangement with the shortest average row list is fastest. This militates in favour of placing the dimension with the highest utilisation rate first. These two effects are occasionally in tension with each other, but for sparse tables it turns out that the best organisation is to move all of the `[]` dimensions to the left, and then sort the `[]` left-to-right by reducing utilisation counts and then to sort the `()` dimensions left-to-right by reducing utilisation count.

8 The Modelling Process

Having looked at notation for, and effects of, different organisations we shall now look at a real example drawn from the GLL algorithm.

Consider the declaration of the GSS:

```
GSSNode (GSSLabel s, Int i)
SPPFSymbolNode (SPPFLabel s, Int leftExtent, Int rightExtent)
GSS (Set GSSEdge (GSSNode src, SPPFSymbolNode w, GSSNode dst))
GSS gss
```

We begin by flattening the declarations into a signature for table `gss` by substitution:

```
Set gss ((*src*) GSSLabel, Int,
         (*w*)   SPPFLabel, Int, Int,
         (*dst*) GSSLabel, Int
        )
```

This is discouraging: on the face of it the signature for `gss` demands a seven dimensional table, of which four dimensions are indexed by `Int`. We can reduce the extent of those dimensions to `Natural(n)` (a natural number in the range 1 to n) since we know that extents and indices are bounded by the length of the parser's input string. Extents for the other dimensions may be found in Table 1: for ANSI-C there are 277 elements in the `GSSLabel` enumeration and 665 in the `SPPFLabel` enumeration. It is not unreasonable to expect a C compiler to process a string of 10,000 (10^4) tokens: our GSS table would then require at least $10^{16} \times 277^2 \times 665 \approx 5.1 \times 10^{22}$ bits, which is clearly absurd.

Now, the signature for a data structure may contain dependent indices. For instance, in the GSS we can show that the left extent of a GSS edge label `w(leftExtent)` is the same as the index of the destination GSS node, and the rightExtent is the same as the index of the source node. These *repeated dimensions* can be removed, at which point our signature is reduced to

```
Set gss ((*src*) GSSLabel, Int,
         (*w*)   SPPFLabel,
         (*dst*) GSSLabel, Int
        )
```

We must now identify dimensions that are candidates for implementation with address calculation. If we have sufficient runtime profile information, we may choose to ignore asymptotic behaviour and use hash tables tuned to the behaviour of our parser on real examples. In this paper, we restrict ourselves to array-style address mapping only.

For dimensions that require constant time lookup we *must* use address mapping. For other dimensions, we *may* use address mapping if the improvement in performance merits the extra space required.

To find out if an indexing operation must be done in constant time, we must analyse the behaviour of our `gll()` function. The outer loop is governed by the removal of contexts from `R` at line 36. From our analysis in [6], we know that there are quadratically many unique contexts in worst case so the outer loop is $O(n^2)$. We also know that there are $O(n)$ GSS nodes, and thus each node has $O(n)$ out-edges (since the number of edge labels is the number of SPPF labels which is constant).

The parser function itself has no further loops: after each context is extracted from `R` we execute linear code and then jump back to `L_Dispatch`. Only the `pop()` and `updateGSS()` functions have inner loops: in `pop()` we iterate over the $O(n)$ out-edges of a GSS node performing calls to `addContext()` and `getSPPFNode()` and in `updateGSS()` we iterate over the $O(n)$ edges in the unfinished pop set, `P`, again performing calls to `addContext()` and `getSPPFNode()`.

Clearly, `addContext()` and `getSPPFNode()` are executed $O(n^3)$ times, and so must themselves execute in constant time. `addContext()` involves only set tests and insertions which will execute in constant time if we use address calculation. `getSPPFNode()` looks up an SPPF symbol node, and then examines its child pack nodes: these operations must execute in constant time and they turn out to be the most space demanding parts of the implementation.

We are presently concerned only with the GSS implementation so we return to function `updateGSS()`. Lines 81-91 implement the updating of the GSS structure: at line 89 we look for a particular GSS node which will be the source node for an edge, and in lines 90 and 91 we conditionally add an edge to the destination node. This operation is done $O(n^2)$ times, so the whole update can take linear time without undermining the asymptotic performance. We choose to implement the initial lookup (line 89) in constant time and allow linear time for the edge update. By this reasoning, we reach a GSS implementation of

```
Set gss ((*src*) (GSSLabel), [Int],
         (*w*)   (SPPFLabel),
         (*dst*) (GSSLabel), (Int)
        )
```

Finally, we consider ordering of dimensions. We need an estimate of the utilisation counts for each dimension. For long strings, utilisation counts of the `Int` dimensions will be greater than the others, because the extent of the `GSSLabel` and `SPPFLabel` enumerations is constant and quite small (277 and 665 for ANSI-C). It is hard to reason about the utilisation rates for the GSS and SPPF labels: experimentation is required (and some initial results are given below). We note, though, that there are more SPPF labels than GSS labels.

We also need to take account of the two-stage access to the GSS table. We first need to find a particular source node, and then subsequently we use the other indices to check for an edge to the destination node. This means that the indices used in the first-stage query must be grouped together at the left.

On this basis, a good candidate for implementation is

```
Set gss ((*src(i)*) [Int], (*src(s)*) (GSSLabel),
         (*dst(i)*) (Int),
         (*w(s)*)   (SPPFLabel),
         (*dst(s)*) (GSSLabel)
        )
```

This refinement process has given us a compact representation where we have done a lot to save space without destroying the asymptotic behaviour implied by the original unrefined specification. Depending on the results of experimentation, we may wish to relax the constraints a little so as to increase performance at the expense of space. For instance, what would be the impact of changing to this implementation?

```
Set gss ((*src(i)*) [Int], (*src(s)*) [GSSLabel],
         (*dst(i)*) (Int),
         (*w(s)*)   (SPPFLabel),
         (*dst(s)*) (GSSLabel)
        )
```

This proposes an array of arrays to access the source GSS node label rather than an array of lists. We can only investigate these kinds of engineering tradeoff in the context of particular grammars, and particular sets of input strings. By profiling the behaviour of our parser on typical applications we can extract real utilisation counts.

We ran a GLL parser for ANSI-C on the source code for `bool`, a Quine-McCluskey minimiser for Boolean equations, and measured the number of GSS labels used at each input index. The first 50 indices yielded these utilisation factors:

11, 18, 13, 10, 0, 1, 61, 11, 47, 0, 0, 19, 23, 0, 10, 3, 18, 0, 10, 18, 0, 44, 0, 0,
4, 7, 0, 45, 0, 10, 0, 44, 0, 0, 10, 0, 44, 0, 0, 46, 0, 14, 2, 0, 6, 16, 3, 45, 0, 10

so, there are 11 GSS nodes with labels of the form $(0, l_g \in \texttt{GSSLabel})$, 18 of the form $(1, l_g)$ and so on. The total number of GSS nodes here is 623. The mean number of GSS labels used per index position is 12.46, but 38% of the indices have no GSS nodes at all. This might seem initially surprising, but recall that a GSS node is only created when an input position has an associated grammar slot which is immediately before a nonterminal.

Now, linked list table nodes require three integer words of memory (one for the index, and two for the pointers). If these statistics are typical, for every 100 index positions we would expect 38 to be unpopulated, which means that there would be 62 second level vectors of length 277 (the `GSSLabel` extent for ANSI-C), to a total of 17 174 words. In the linked version, we would expect 1246 nodes altogether, and that would require 3738 words, so the `([], [], ...)` representation requires 4.6 times as much memory as the `( (), (), ...)` version, which may not be too onerous. The performance advantages are clear: the mean list length would be 12.46, leading to an expected lookup time 6–10 times slower than directly indexing the vectors even without taking into account cache effects. There are also several lists which would be greater than 40 elements long, leading to substantial loss of performance.

9 Conclusion: Prospects for Automatic Refinement

In this paper we have given some details of our implementation of the GLL technique using an approach that separates high level reasoning about algorithm complexity from details of implementation, and a modelling process that allows us to produce compact implementations which achieve the theoretically predicted performance. We summarise our procedure as follows.

1. Flatten declarations to signatures by substitution.
2. Establish upper bounds on dimensions.
3. Remove dimensions that may be mapped from other dimensions and create maps.
4. Remove repeated dimensions.
5. Identify dimensions that govern the time asymptote — these are the *critical* dimensions.
6. For each dimension, compute, measure or guess the average number of elements of the index type T that are used in typical applications: this is the dimension's Likely Utilisation Count (LUC). The ratio ($|T|$/LUC) is called the dimension's load factor (LF).
7. Implement critical dimensions as one dimensional tables.
8. Implement dimensions whose LF is greater than 33% as one dimensional tables.
9. Group dimensions by query level, with outer queries to the left of inner queries.
10. Within query groups, arrange dimensions so that `[]` indices are always to the left of `()` indices.
11. Within query groups and index modes, sort dimensions from left to right so that load factors increase from left to right.

For a given data structure, the available index modes and orderings define a space of potential implementations. As we have shown, the basic procedure minimises space, but nearby points in the implementation space may have better performance, and as long as the application fits into available memory, most users would like to have the faster version.

In the future we propose that semi-automatic systems be constructed to automatically explore these spaces in much the same way that hardware-software co-design systems have successfully attacked the exploration of implementation spaces for hardware oriented specifications. We have in mind the annotation of critical dimensions by the theoretician, allied to a profiling system which will collect statistics from sets of test inputs so as to measure or estimate load factors. We believe that a notation similar to LC's will be suitable for such an optimiser.

References

1. Gnu Bison home page (2003), `http://www.gnu.org/software/bison`
2. Earley, J.: An efficient context-free parsing algorithm. Communications of the ACM 13(2), 94–102 (1970)
3. Knuth, D.E.: On the translation of languages from left to right. Information and Control 8(6), 607–639 (1965)
4. Nozohoor-Farshi, R.: GLR parsing for ϵ-grammars. In: Tomita, M. (ed.) Generalized LR Parsing, pp. 60–75. Kluwer Academic Publishers, The Netherlands (1991)
5. Scott, E., Johnstone, A.: Generalised bottom up parsers with reduced stack activity. The Computer Journal 48(5), 565–587 (2005)
6. Scott, E., Johnstone, A.: GLL parsing. Electronic Notes in Theoretical Computer Science (2009)
7. Scott, E., Johnstone, A., Economopoulos, G.: A cubic Tomita style GLR parsing algorithm. Acta Informatica 44, 427–461 (2007)
8. Tomita, M.: Efficient parsing for natural language. Kluwer Academic Publishers, Boston (1986)
9. van den Brand, M.G.J., Heering, J., Klint, P., Olivier, P.A.: Compiling language definitions: the ASF+SDF compiler. ACM Transactions on Programming Languages and Systems 24(4), 334–368 (2002)
10. Younger, D.H.: Recognition of context-free languages in time n^3. Inform. Control 10(2), 189–208 (1967)

Metamodel Usage Analysis for Identifying Metamodel Improvements

Markus Herrmannsdoerfer, Daniel Ratiu, and Maximilian Koegel

Institut für Informatik, Technische Universität München
Boltzmannstr. 3, 85748 Garching b. München, Germany
{herrmama,ratiu,koegel}@in.tum.de

Abstract. Modeling languages raise the abstraction level at which software is built by providing a set of constructs tailored to the needs of their users. Metamodels define their constructs and thereby reflect the expectations of the language developers about the use of the language. In practice, language users often do not use the constructs provided by a metamodel as expected by language developers. In this paper, we advocate that insights about how constructs are used can offer language developers useful information for improving the metamodel. We define a set of usage and improvement patterns to characterize the use of the metamodel by the built models. We present our experience with the analysis of the usage of seven metamodels (EMF, GMF, UNICASE) and a large corpus of models. Our empirical investigation shows that we identify mismatches between the expected and actual use of a language that are useful for metamodel improvements.

1 Introduction

Modeling languages promise to increase productivity of software development by offering their users a restricted set of language constructs that directly address their needs. Many times, modeling languages address a well-defined category of users, since these languages are domain-specific, represent technology niches, or are built in-house. Thereby, the needs of the users are the most important driving force for the existence and evolution of the modeling language. Usable languages have a small set of intuitive constructs, are easy to use, hard to misuse, and easy to learn by novices [8]. The constructs of modeling languages are typically defined by metamodels which reflect the expectations of the language developers about the future use of the language.

The question whether these expectations are fulfilled and whether the language users use the language in the same manner as expected has a central importance for language developers [9,15]. Too general modeling languages that have a too wide scope hamper the productivity of their users, are more difficult to learn, and are prone to errors. To answer this question, we need direct feedback from the language users. Without feedback about the actual use, the developers can only guess whether the language meets the expectation of its users. The traditional means by which developers get feedback about their languages (e. g.

community forums), are limited to certain kinds of information like e. g. mostly bug reports and usage questions.

In the case when the language developers have access to a relevant set of models built with the language, they can take advantage of this information and learn about how the language is used by investigating its utterances. Once the information about the actual use of the language is available, the language can be improved along two main directions: firstly, restricting the language and eliminating unnecessary constructs, and secondly, by adding new language constructs that language users need. In this paper, we advocate that the analysis of built models can reflect essential information about the use of the language that is otherwise inaccessible. By using our method, we aim to close the loop between language developers and users, since the information about the actual use can serve for language enhancements or even for the elimination of defects.

This paper contains three main contributions to the state of the art:

1. We present a general method by which language developers can obtain feedback about how the modeling language is actually used by analyzing the models conforming to its metamodel.
2. We describe a set of patterns that characterize the usage of the metamodel and which can be used to improve the language by eliminating obsolete or superfluous constructs, adding missing metamodel constraints that were not set by language developers, or by adding new language constructs that better reflect the needs of the users.
3. We present our experience with applying these analyses to seven metamodels, and we show that even in the case of well-known metamodels like EMF we can identify different defects that give rise to metamodel improvements.

Outline. Sec. 2 describes the metamodeling formalism that we considered for defining our analyses. In Sec. 3, we introduce our method as well as a set of usage analyses to identify metamodel improvements. Sec. 4 presents the results of applying the analyses on a corpus of well-known metamodels and their models. In Sec. 5, we discuss related work, before we conclude the paper in Sec. 6.

2 Metamodeling Formalism

Metamodel. To describe metamodels, we use the simplified E-MOF [14] formalism illustrated in Fig. 1 as a class diagram. A metamodel defines a set of *Type*s which can be either primitive (*PrimitiveType*) or complex (*Class*). Primitive types are either *DataType*s—like Boolean, Integer and String—or *Enumeration*s of *Literals*. Classes consist of a set of *features*. They can have *super types* to inherit features and might be *abstract*. The *name* of a feature needs to be unique among all features of a class including the inherited ones. A *Feature* has a multiplicity (*lower bound* and *upper bound*) and maybe *derived*—i. e. its value is derived from the values of other features. A feature is either an *Attribute* or a *Reference*: an attribute is a feature with a primitive *type*, whereas a reference is a

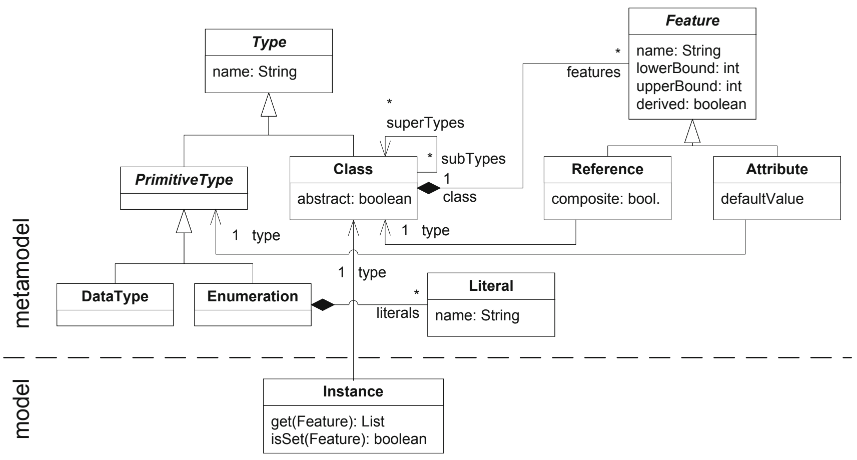


Fig. 1. Metamodeling formalism

feature with a complex *type*. An attribute might also define a *default value* that serves as an initial value for instances. A reference might be *composite*, meaning that an instance can only have one parent via a composite reference.

The classes in Fig. 1 can be interpreted as sets—e. g. *Class* denotes the set of classes defined by a metamodel, and the references can be interpreted as navigation functions on the metamodel—e. g. $c.subTypes$ returns the sub types of $c \in Class$. Additionally, * denotes the transitive closure on navigation functions—e. g. $c.subTypes^*$ returns all sub types of $c \in Class$ including c itself, and $PV(t)$ denotes the possible values of a primitive type $t \in PrimitiveType$.

Model. A model consists of a set of *Instance*s each of which have a class from the metamodel as *type*. To define our analyses in Sec. 3.2, we require an instance to provide two methods. First, the method $i.get(f)$ returns the value of a feature $f \in Feature$ for a certain instance $i \in Instance$. The value is returned as a list even in the case of single-valued features to simplify the formulas. Second, the method $i.isSet(f)$ returns true if the value of a feature $f \in Feature$ is set for a certain instance $i \in Instance$. A feature is set if and only if the value of the feature is different from the empty list and different from the *default value*, in case the feature is an attribute.

3 Metamodel Usage Analysis

In this section, we present our approach to identify metamodel improvements by analyzing how the metamodel is used by the built models. Sec. 3.1 introduces templates to define usage analyses. Sec. 3.2 lists a number of analyses defined using these templates. Sec. 3.3 presents the implementation of the approach.

3.1 Templates for Defining Usage Analyses

Our ultimate goal is to recommend metamodel changes to improve the usability of the language. The metamodel can be changed by applying refactorings, constructors, or destructors [20]. *Refactorings* restructure the metamodel, but do not change the expressiveness of the language defined by the metamodel. By contrast, *destructors* and *constructors* decrease or increase the expressiveness of the language, respectively. By analyzing only the metamodel, we can recommend only refactorings of the metamodel. If we also take the models built with a metamodel into account, we can also recommend destructors and constructors. Destructors can be identified by finding metamodel constructs that are not used in models, and that can thereby be safely removed without affecting the existing models. Constructors can be identified by finding metamodel constructs that are used to encode constructs currently not available in the language. By enriching the metamodel with the needed constructs, we support the users to employ the language in a more direct manner.

Collecting usage data. Before we can identify metamodel improvements, we need to collect usage data from the models built with the metamodel. This usage data collection has to fulfill three requirements: (1) We need to collect data from a significant number of models built with the metamodel. In the best case, we should collect usage data from every built model. If this is not possible, we should analyze a significant number of models to be sure that the results of our analyses are relevant and can be generalized for the actual use of the language. Generally, the higher the ratio of the existing models that are analyzed, the more relevant our analyses. (2) We need to collect the appropriate amount of data necessary for identifying metamodel improvements. If we collect too much data, we might violate the intellectual property of the owners of the analyzed models. If we collect too few data, we might not be able to extract meaningful information from the usage data. (3) The usage data from individual models needs to be collected in a way that it can be composed without losing information.

To specify the collection of usage data, we employ the following template:

Context: the kind of metamodel element for which the usage data is collected. The context can be used as a pattern to apply the usage data collection to metamodel elements of the kind.

Confidence: the number of model elements from which the usage data is collected. The higher this number, the more confidence we can have in the collected data. In the following, we say that we are not confident if this number is zero, i. e. we do not have usage data that can be analyzed.

Specification: a function to specify how the usage data is collected. There may be different result types for the data that is collected. In the following, we use numbers and functions that map elements to numbers.

Analyzing usage data. To identify metamodel improvements, we need to analyze the usage data collected from the models. The analysis is based on an expectation that we have for the usage data. If the expectation about the usage of a metamodel construct is not fulfilled, the construct is a candidate for improvement. To specify expectations and the identification of metamodel improvements from these expectations, we employ the following template:

Expectation: a boolean formula to specify the expectation that the usage data needs to fulfill. If the formula evaluates to true, then the expectation is fulfilled, otherwise we can propose an improvement. Certain expectations can be automatically derived from the metamodel, e. g. we expect that a non-abstract class is used in models. Other expectations can only be defined manually by the developer of the metamodel, e. g. that certain classes are more often used than other classes. In the following, we focus mostly on expectations that can be automatically derived from the metamodel, as they can be applied to any metamodel without additional information.

Improvement: the metamodel changes that can be recommended if the expectation is not fulfilled. The improvement is specified as operations that can be applied to the metamodel. As described above, the improvements can restrict the language by removing existing constructs, or enlarge the language by adding new constructs. If we have collected data from all models built with a metamodel, then we can also be sure that the restrictions can be safely applied, i. e. without breaking the existing models.

3.2 Towards a Catalog of Usage Analyses

In this section, we present a catalog of analyses of the usage of metamodels. Each subsection presents a category of analyses, each analysis being essentially a question about how the metamodel is actually used. We use the templates defined in the previous section to define the analyses in a uniform manner. This catalog of analyses is by no means complete, but rather represents a set of basic analyses. We only define analyses that are used in Sec. 4 as part of the study.

Class Usage Analysis. If metamodels are seen as basis for the definition of the syntax of languages, a non-abstract class represents a construct of the language. Thereby, the measure in which a language construct is used can be investigated by analyzing the number of instances $CU(c)$ of the non-abstract class c defined by the metamodel:

Context: $c \in Class$, $\neg c.abstract$
Confidence: $\|Instance\|$
Specification: $CU(c) := \|\{i \in Instance | i.type = c\}\|$

Q1) Which classes are not used? We expect that the number of instances for a non-abstract class is greater than zero. Classes with no instance represent language constructs that are not needed in practice, or the fact that language users did not know or understand how to use these constructs:

Expectation: $CU(c) > 0$
Improvement: delete the class, make the class abstract

Classes with no instances that have subclasses can be made abstract, otherwise, classes without subclasses might be superfluous and thereby are candidates to be deleted from the metamodel. Both metamodel changes reduce the number of constructs available to the user, making the language easier to use and learn. Furthermore, deleting a class results in a smaller metamodel implementation which is easier to maintain by the language developers. Non-abstract classes that the developers forgot to make abstract can be seen as metamodel bugs.

Q2) What are the most widely used classes? We expect that the more central a class of the metamodel is, the higher the number of its instances. If a class is more widely used than we expect, this might hint at a misuse of the class by the users or the need for additional constructs:

Expectation: the more central the construct, the higher its use frequency
Improvement: the classes that are used more frequently than expected are potential sources for language extensions

This analysis can only be performed manually, since the expectation cannot be derived automatically from the metamodel in a straightforward manner.

Feature Usage Analysis. If metamodels are seen as basis for the definition of a language, features are typically used to define how the constructs of a modeling language can be combined (references) and parameterized (attributes). As derived features cannot be set by users, we investigate only the use of non-derived features:

Context: $f \in Feature, \neg f.derived$
Confidence: $\|FI(f)\|$, $FI(f) := \{i \in Instance | i.type \in f.class.subTypes^*\}$
Specification: $FU(f) := \|\{i \in FI(f) | i.isSet(f)\}\|$

We can only be confident for the cases when there exist instances $FI(f)$ of classes in which the feature f could possibly be set, i. e. in all sub classes of the class in which the feature is defined.

Q3) Which features are not used? We expect that the number of times a non-derived feature is set is greater than zero. Otherwise, we can make it derived or even delete it from the metamodel:

Expectation: $FU(f) > 0$
Improvement: delete the feature, make the feature derived

If we delete a feature from the metamodel or make it derived, it can no longer be set by the users, thus simplifying the usage of the modeling language. Features that are not derived but need to be made derived can be seen as a bug in the metamodel, since the value set by the language user is ignored by the language interpreters.

Feature Multiplicity Analysis. Multiplicities are typically used to define how many constructs can be referred to from another construct. Again, we are only interested in non-derived features, and we can only be confident for a feature, in case there are instances in which the feature could possibly be set:

Context: $f \in \mathit{Feature}, \neg f.\mathit{derived}$
Confidence: $\|FU(f)\|$
Specification: $FM(f) : \mathbb{N} \to \mathbb{N}$, $FM(f, n) := \|\{i \in FU(f) | \|i.\mathit{get}(f)\| = n\}\|$

Q4) Which features are not used to their full multiplicity? We would expect that the distribution of used multiplicities covers the possible multiplicities of a feature. More specifically, we are interested in the following two cases: First, if the lower bound of the feature is 0, there should be instances with no value for the feature—otherwise, we might be able to increase the lower bound:

Expectation: $f.\mathit{lowerBound} = 0 \Rightarrow FM(f, 0) > 0$
Improvement: increase the lower bound

A lower bound greater than 0 explicitly states that the feature should be set, thus avoiding possible errors when using the metamodel. Second, if the upper bound of the feature is greater 1, there should be instances with more than one value for the feature – otherwise, we might be able to decrease the upper bound:

Expectation: $f.\mathit{upperBound} > 1 \Rightarrow \max_{n \in \mathbb{N}} FM(f, n) > 1$
Improvement: decrease the upper bound

Decreasing the upper bound reduces the number of possible combinations of constructs and thereby simplifies the usage of the language.

Attribute Value Analysis. The type of an attribute defines the values that an instance can use. The measure in which the possible values are covered can be investigated by determining how often a certain value is used. Again, we are only interested in non-derived attributes, and we can only be confident for an attribute, if there are instances in which the attribute could possibly be set:

Context: $a \in \mathit{Attribute}$, $\neg a.\mathit{derived}$
Confidence: $\|FI(a)\|$
Specification: $AVU(a) : PV(a.\mathit{type}) \to \mathbb{N}$,
$AVU(a, v) := \|\{i \in FI(a) | v \in i.\mathit{get}(a)\}\|$

Q5) Which attributes are not used in terms of their values? We expect that all the possible values of an attribute are used. In case of attributes that have a finite number of possible values (e. g. Boolean, Enumeration), we require them to be all used. In case of attributes with a (practically) infinite domain (e. g. Integer, String), we require that more than 10 different values are used. Otherwise, we might be able to specialize the type of the attribute:

Expectation: $(\|PV(a.type)\| < \infty \Rightarrow \|PV(a.type)\| = \|VU(a)\|) \wedge (\|PV(a.type)\| = \infty \Rightarrow \|VU(a)\| > 10)$, where $VU(a) = \{v \in PV(a.type) | AVU(a, v) > 0\}$
Improvement: specialize the attribute type

More restricted attributes can give users better guidance about how to fill its values in models, thus increasing usability. Additionally, such attributes are easier to implement for developers, since the implementation has to cover less cases.

Q6) Which attributes do not have the most used value as default value? In many cases, the language developers set as default value of attributes the values that they think are most often used. In these cases, the value that is actually most widely used should be set as default value of the attribute:

Expectation: $a.upperBound = 1 \Rightarrow a.defaultValue = [mvu \in PV(a.type) : \max_{v \in PV(a.type)} AVU(a, v) = AVU(a, mvu)]$
Improvement: change the default value

If this is not the case, and users use other values, the new ones can be set as default.

Q7) Which attributes have often used values? We expect that no attribute value is used too often. Otherwise, we might be able to make the value a first-class construct of the metamodel, e. g. create a subclass for the value. A value is used too often if its usage share is at least 10%:

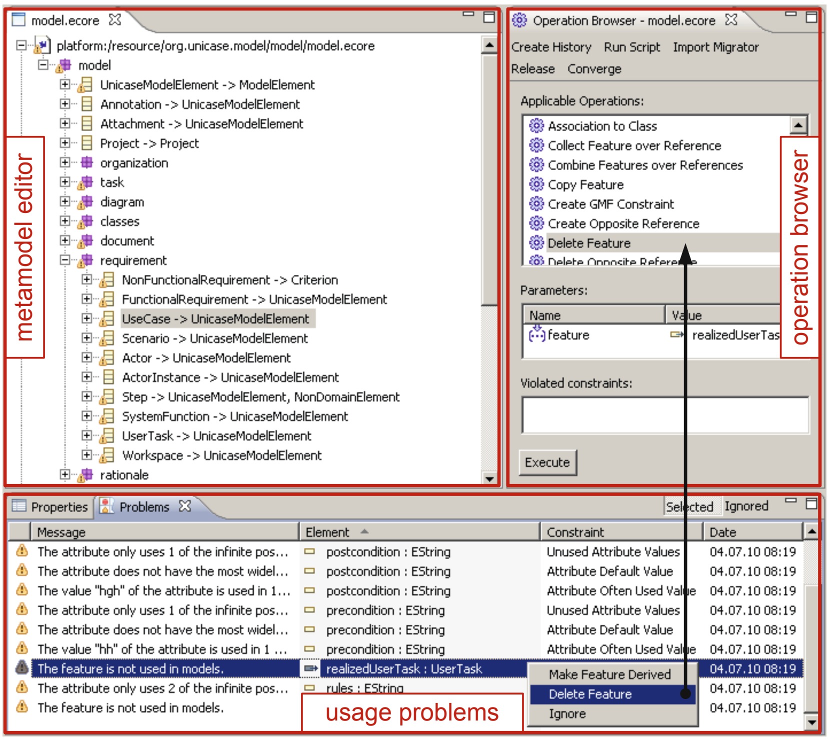
Expectation: $\|PV(a.type)\| = \infty \Rightarrow \forall v \in PV(a.type) : AVU(a, v) < 10\% \cdot FU(a)$
Improvement: lift the value to a first-class construct

Lifting the value to an explicit construct, makes the construct easier to use for users as well as easier to implement for developers.

3.3 Prototypical Implementation

We have implemented the approach based on the Eclipse Modeling Framework (EMF) [18] which is one of the most widely used modeling frameworks. The collecting of usage data is implemented as a batch tool that traverses all the model elements. The results are stored in a model that conforms to a simple metamodel for usage data. The batch tool could be easily integrated into the modeling tool itself and automatically send the data to a server where it can be accessed by the language developers. Since the usage data is required to be composeable, it can be easily aggregated.

The usage data can be loaded into the metamodel editor which proposes improvements based on the usage data. The expectations are implemented as constraints that can access the usage data. Fig. 2 shows how violations of these expectations are presented to the user in the *metamodel editor*. Overlay icons indicate the metamodel elements to which the violations apply, and a view provides



```
platform:/resource/org.unicase.model/model/model.ecore
  model
    UnicaseModelElement -> ModelElement
    Annotation -> UnicaseModelElement
    Attachment -> UnicaseModelElement
    Project -> Project
    organization
    task
    diagram
    classes
    document
    requirement
      NonFunctionalRequirement -> Criterion
      FunctionalRequirement -> UnicaseModelElement
      UseCase -> UnicaseModelElement
      Scenario -> UnicaseModelElement
      Actor -> UnicaseModelElement
      ActorInstance -> UnicaseModelElement
      Step -> UnicaseModelElement, NonDomainElement
      SystemFunction -> UnicaseModelElement
      UserTask -> UnicaseModelElement
      Workspace -> UnicaseModelElement
    rationale
```

Association to Class
Collect Feature over Reference
Combine Features over References
Copy Feature
Create GMF Constraint
Create Opposite Reference
Delete Feature
Delete Opposite Reference

Name	Value
feature	realizedUserTas

Message	Element	Constraint	Date
The attribute only uses 1 of the infinite pos...	postcondition : EString	Unused Attribute Values	04.07.10 08:19
The attribute does not have the most widel...	postcondition : EString	Attribute Default Value	04.07.10 08:19
The value "hgh" of the attribute is used in 1...	postcondition : EString	Attribute Often Used Value	04.07.10 08:19
The attribute only uses 1 of the infinite pos...	precondition : EString	Unused Attribute Values	04.07.10 08:19
The attribute does not have the most widel...	precondition : EString	Attribute Default Value	04.07.10 08:19
The value "hh" of the attribute is used in 1 ...	precondition : EString	Attribute Often Used Value	04.07.10 08:19
The feature is not used in models.	realizedUserTask : UserTask		04.07.10 08:19
The attribute only uses 2 of the infinite pos...	rules : EString		04.07.10 08:19
The feature is not used in models.			04.07.10 08:19

Fig. 2. Proposing metamodel improvements

a list of all *usage problems*. The constraints have been extended to be able to propose operations for metamodel improvements. The operations are implemented using our tool COPE [7] whose purpose is to automate the model migration in response to metamodel evolution. The proposed operations are shown in the context menu and can be executed via COPE's *operation browser*.

4 Empirical Study

We have performed an empirical study to validate whether the identified metamodel improvements would really lead to changes of the metamodel. Sec. 4.1 presents the method that we have applied, and Sec. 4.2 the metamodels and models that we have analyzed. The results of the empirical study are explained in Sec. 4.3 and discussed in Sec. 4.4.

4.1 Study Method

To perform our analyses, we performed the following steps:

1. *Mine models*: We obtained as many models as possible that conform to a certain metamodel. In the case of an in-house metamodel, we asked the

metamodel developers to provide us with the models known to them. For the published metamodels, we iterated through several open repositories (CVS and SVN) and downloaded all models conforming to these metamodels. As far as possible, we removed duplicated models.

2. *Perform usage analysis*: We applied the approach presented in Sec. 3 on the mined models. For each of the analyzed metamodels, this results in a set of usage problems.
3. *Interpret usage problems*: To determine whether the usage problems really help us to identify possible metamodel improvements, we tried to find explanations for the problems. In order to do so, we investigated the documentation of the metamodels as well as the interpreters of the modeling languages, and, if possible, we interviewed the metamodel developers.

4.2 Study Objects

Metamodels. To perform our experiments, we have chosen 7 metamodels whose usage we have analyzed. Table 1 shows the number of elements of these metamodels. Two metamodels are part of the Eclipse Modeling Framework (EMF)[1] which is used to develop the abstract syntax of a modeling language: The ecore metamodel defines the abstract syntax from which an API for model access and a structural editor can be generated; and the genmodel allows to customize the code generation. Four metamodels are part of the Graphical Modeling Framework (GMF)[2] which can be used to develop the diagrammatic, concrete syntax of a modeling language: the graphdef model defines the graphical elements like nodes and edges in the diagram; the tooldef model defines the tools available to author a diagram; the mappings model maps the nodes and edges from the graphdef model and the tools from the tooldef model onto the metamodel elements from the ecore model; and the mappings model is transformed into a gmfgen model which can be altered to customize the generation of a diagram editor. Finally, the last metamodel (unicase) is part of the tool UNICASE[3] which can be used for UML modeling, project planning and change management.

Table 1. A quantitative overview over the analyzed metamodels

#	ecore	genmodel	graphdef	tooldef	mappings	gmfgen	unicase
Class	20	14	72	26	36	137	77
Attribute	33	110	78	16	22	302	88
Reference	48	34	57	12	68	160	161

Models. For each metamodel, we have mined models from different repositories. Table 2 shows the repositories as well as the number of files and elements which

[1] See EMF web site: `http://www.eclipse.org/emf`
[2] See GMF web site: `http://www.eclipse.org/gmf`
[3] See UNICASE web site: `http://unicase.org`

Table 2. A quantitative overview over the analyzed models

repository	ecore		genmodel		graphdef		tooldef		mappings		gmfgen	
	files	elem.s	files	elem.s	files	elem.s	files	elem.s	files	elem.s	files	elem.s
AUTOSAR	18	384,685	18	16,189	11	1,835	11	436	11	538	13	2,373
Eclipse	1,834	250,107	818	69,361	105	6,026	58	1,769	72	5,040	52	11,043
GForge	106	26,736	94	41,997	12	806	10	241	11	480	11	1,680
Google	50	9,266	59	3,786	69	7,627	74	2,421	76	4,028	78	18,710
Atlantic Zoo	278	68,116	–	–	–	–	–	–	–	–	–	–
altogether	2,286	738,910	989	131,333	197	16,294	153	4,867	170	10,086	154	33,806

have been obtained from them. Models that conform to the first 6 metamodels have been obtained from the AUTOSAR development partnership[4], from the Eclipse[5] and GForge[6] open source repositories, by querying the Google Code Search[7] and from the Atlantic Zoo[8]. The Atlantic Zoo only contains ecore models, while the other repositories contain models for all EMF and GMF metamodels. For the unicase metamodel, its developers provided us with 3 files consisting of 8,213 model elements.

4.3 Study Results

In this section, we present the study results separately for each question mentioned in Sec. 3.2. To facilitate understanding the explanations for the identified usage problems, we clustered them according to high-level explanations.

Q1) Which classes are not used? Table 3 quantitatively illustrates for each metamodel the number of used and not used classes in the overall number of non-abstract classes. The second last row shows the number of classes that are used, whereas the other rows classify the unused classes according to the explanations why they are not used. As presented in Sec. 4.1, we derived these explanations by manually analyzing the documentation and implementation of the metamodels or interviewing the developers.

Classes that are obsolete, not implemented, or that logically belong to another metamodel can be removed. A class *is obsolete* if it is not intended to be used in the future. For example, in unicase, the 4 classes to define UML stereotypes are no longer required. A class *is not implemented* if it is not used by the interpreters of the modeling language. For example, the tooldef metamodel defines 7 classes to specify menus and toolbars from which, according to [4], currently no code can be generated. A class *should be moved* to another metamodel if it logically belongs to the other metamodel. For example, in tooldef, the class GenericStyle-Selector should be moved to mappings which contains also a composite reference

[4] See AUTOSAR web site: `http://www.autosar.org`
[5] See Eclipse Repository web site: `http://dev.eclipse.org/viewcvs/index.cgi`
[6] See GForge web site: `http://gforge.enseeiht.fr`
[7] See Google Code Search web site: `http://www.google.com/codesearch`
[8] See Atlantic Zoo web site: `http://www.emn.fr/z-info/atlanmod/index.php/Ecore`

Table 3. Usage of classes

Explanation	ecore	genmodel	graphdef	tooldef	mappings	gmfgen	unicase
is obsolete							4
is not implemented				7			1
should be moved				1			2
should be abstract	1						
should be transient	1					2	1
is too new			3			6	5
should be used			1		1	6	9
is used	13	11	51	11	24	83	42
altogether	15	11	55	19	25	97	64

Table 4. Usage of ecore classes

#	Class	Number	Share
1	EStringToStringMapEntry	328,920	44.51%
2	EGenericType	141,043	19.09%
3	EAnnotation	102,863	13.92%
4	EReference	53,177	7.20%
5	EClass	41,506	5.62%
6	EAttribute	36,357	4.92%
7	EEnumLiteral	10,643	1.44%
8	EOperation	7,158	0.97%
9	EParameter	7,060	0.96%
10	EPackage	4,530	0.61%
11	EDataType	2,747	0.37%
12	EEnum	2,513	0.34%
13	ETypeParameter	226	0.03%
14	EObject	0	0.00%
15	EFactory	0	0.00%

targeting this class. Another example is unicase where 2 classes should be moved to a different metamodel of another organization that also uses the framework underlying UNICASE.

Our manual investigations revealed that other classes that are not used should be abstract or transient. A class *should be abstract* if it is not intended to be instantiated, and is used only to define common features inherited by its subclasses. For instance, in ecore, EObject—the implicit common superclass of all classes—should be made abstract. A class *should be transient* if its instances are not expected to be made persistent—such a class does not represent a language construct. However, the employed metamodeling formalism currently does not support to specify that a class is transient. For instance, in ecore, EFactory—a helper class to create instances of the specified metamodel—should be transient.

Finally, there are non-used classes which do not require any change, since they are either *too new* or *should be used*—i. e. we could not find any plausible explanation why they are not used. A class is *too new* if it was recently added and thus is not yet instantiated in models. For the GMF metamodels graphdef and gmfgen, we identified 9 new classes, while for the unicase metamodel, we found 5 classes that are too new to be used.

Q2) What are the most widely used classes in ecore? Due to space constraints, we focus on the ecore metamodel to answer this question. Table 4 shows its classes and their corresponding number of instances. Interestingly, the number of instances of the ecore classes has an exponential distribution, a phaenomenon observed also in case of the other metamodels. Hence, each metamodel contains a few constructs which are very central to its users. In the case of ecore, we expect that classes, references and attributes are the central constructs. However, the most widely used ecore class is—with more than 44,5% of the analyzed instances—EStringToStringMapEntry which defines key-value-pairs for EAnnotations making up 13,9% of the instances. The fact that the annotation mechanism which is used to extend the ecore metamodel is more widely used than the first-class constructs, suggests the need for increasing ecore's expressiveness. As we show in Q7, some often encountered annotations could be lifted to first-class

Table 5. Usage of features

Explanation	ecore	genmodel	graphdef	tooldef	mappings	gmfgen	unicase
is obsolete						9	11
is not implemented		3		2	2	5	
should be moved							11
should be derived							2
is too new			6			16	6
should be used	1	7	2			55	13
is not confident			6	6	1	23	35
is used	51	125	118	20	84	333	163
altogether	52	135	132	28	87	441	241

Table 6. Usage of multiplicity by features

Explanation	ecore	genmodel	graphdef	tooldef	mappings	gmfgen	unicase
should increase ↩ the lower bound		2	1	2	1	21	5
is not implemented				3	1		
should have a ↩ derived default		5	1			48	3
is too new			5				2
should be used		5	4		4	12	3
is not confident			6	6	1	23	35
is used	52	123	115	17	80	337	193
altogether	52	135	132	28	87	441	241

constructs. Another ecore class that is used very often is EGenericType. This is surprising, since we would expect that generic types are very rarely used in metamodels. The investigation of how this class is exactly used, revealed the fact that only 1,8% (2,476) of EGenericType's instances do not define default values for their features and thereby these instances really represent generic types. In the other 98%, the EGenericType is only used as an indirection to the non-generic type, i. e. in a degenerated manner.

Q3) Which features are not used? Table 5 shows for each metamodel the number of used and unused features in the overall number of non-derived features. In the table, we observe a correlation between the number of unused features and the overall number of features. In most cases, the more features a metamodel defines, the less features are actually used. The only exception to this conjecture is the usage of the tooldef metamodel, in which most features are however not implemented as explained below. The table also classifies the unused features according to explanations why they are not used.

Features that are obsolete, not implemented, or logically belong to another metamodel can be removed from the metamodel. A feature *is obsolete* if it is not intended to be used in the future. If none of the analyzed models uses that feature, it is a good candidate to be removed. We identified the 9 obsolete features of gmfgen by investigating the available migrator. Surprisingly, the migrator removes their values, but the developers forgot to remove the features from gmfgen. In the case of the unicase metamodel, 11 unused features are actually obsolete. A feature is classified as *not implemented* if it is not used by the interpreters of the modeling language. We have identified 12 not implemented features of the EMF metamodel genmodel and the GMF metamodels tooldef, mappings and gmfgen by checking whether they are accessed by the code generators. A feature *should be moved* to another metamodel if it logically belongs to the other metamodel. The features of the unicase metamodel that have to be moved target the classes that should be moved to another metamodel as found by question Q1 in Sec. 3.2.

Another set of non-used features can be changed into derived features, since their values are calculated based on other features. As they are anyway overwritten in the metamodel implementation, setting them explicitly is a mistake, making the language easy to misuse. For example, for the unicase metamodel, we identified 2 features that *should be derived.*

Finally, there are non-used features which do not require changes at all, since they are either too new or our investigation could not identify why the feature is not used. Again, a feature *is too new* to be instantiated, if it was recently added. Like for classes, the 28 too new features were identified by investigating the metamodel histories. The only feature of the ecore metamodel which is not set in any model but *should be used* allows to define generic exceptions for operations. Apparently, exceptions rarely need to have type parameters defined or assigned. For both genmodel and gmfgen, more than 5% of the features *should be used* but are not used. The manual investigation revealed that these are customizations of the code generation that are not used at all. We are *not confident* about the usage of a feature if there are no instances in which the feature could be set. This category thus indicates how many features cannot be used, because the classes in which they are defined are not instantiated.

Q4) Which features are not used to their full multiplicity? Table 6 shows as *is used* the number of used features fulfilling these expectations. Again, the violations are classified according to different explanations that we derived after manual investigation of the metamodel documentation and implementation.

A not completely used feature which can be restricted either *should increase the lower bound* or *is not implemented* yet. Even though we found features whose usage did not completely use the upper bound, we could not find an explanation for any of them. However, we found that some features *should increase lower bound* from 0 to 1. For the EMF and GMF metamodels, we found such features by analyzing whether the code generator does not check whether the feature is set, thereby producing an error if the feature is not set. For the unicase metamodel, the too low lower bounds date back from the days when its developers used an object-to-relational mapping to store data. When doing so, a lower bound of 1 was transformed into a database constraint which required to set the feature already when creating a model element. To avoid this restriction, the lower bounds were decreased to 0, when in effect, they should have been 1. As they no longer store the models in a database, the lower bounds could easily be increased. Again, a feature *is not implemented* if the interpreter does not use the feature. In the tooldef metamodel, we found 3 such features which are not interpreted by the code generator, making them also superfluous.

A not completely used feature which requires to extend the metamodeling formalism *should have a derived default.* A feature *should have a derived default* if it has lower bound 0, but in case it is not set, a default value is derived. This technique is mostly used by the code generation metamodels genmodel and gmfgen to be able to customize a value which is otherwise derived from other features. It is implemented by overwriting the API to access the models which is generated from the metamodel. However, the used metamodeling formalism does not provide a means to specify that a feature has a derived default.

A not completely used feature which does not require changes either *is too new* or *should be used.*

Table 7. Usage of complete values by attributes

Explanation	ecore	genmodel	graphdef	tooldef	mappings	gmfgen	unicase
should be specialized		1				3	3
is too new				1		3	
should be used	1	14	14		3	18	4
is not confident		12	44	7	4	164	37
is used	25	82	19	8	14	102	39
altogether	26	109	77	16	21	290	83

Table 8. Usage of default values of attributes

Explanation	ecore	genmodel	graphdef	tooldef	mappings	gmfgen	unicase
should be changed		2	5			3	
should be set		4	13	1		6	
should not be changed	2	1	1		1		2
should have no default	6	16	7	1	3	38	9
is too new				2			
is not confident		10	19	7	4	93	37
is used	18	76	32	5	13	150	35
altogether	26	109	77	16	21	290	83

Q5) Which attributes are not used in terms of their values? Table 7 illustrates for each metamodel the number of attributes whose values are completely used or not. The table also classifies the not completely used attributes according to different explanations.

A not completely used attribute that can be changed *should be specialized* by restricting its type. In the unicase metamodel, three attributes which use String as domain for UML association multiplicities and UML attribute types can be specialized. We found 4 more such attributes in the genmodel and gmfgen metamodels.

Finally, there are not completely used attributes which do not require changes at all, since they are either too new, should be used, or we do not have enough models to be confident about the result. We are only confident if the attribute is set sufficiently often to cover all its values (finite) or 10 values (infinite). In all metamodels, most of the findings fall into one of these categories.

Q6) Which attributes do not have the most used value as default value? Table 8 illustrates for each metamodel the number of attributes whose value is the same as the default value (*is used*) and those that have different values. Note that in many cases the language developers successfully anticipated the most often used values of attributes by setting the appropriate default value. The table also classifies the deviations according to different explanations.

In case the default value is intended to represent the most widely used value of an attribute, and we found that the users use other default values, the default value *should be changed*. In this way, the new value better anticipates the actual use, and thereby the effort of language users to change the attribute value is necessary in less cases. We found 8 attributes whose default value needs to be updated in the metamodels genmodel, graphdef and gmfgen. An attribute has a default value that *should be set* if it does not currently have a default value, but the usage analysis identifies a recurrent use of certain values. By setting a meaningful default value, the language users are helped. In nearly each metamodel, we found attributes whose default value needs to be set.

Table 9. Usage of values often used by attributes

Explanation	ecore	genmodel	graphdef	tooldef	mappings	gmfgen	unicase
should be lifted	3	1	3			3	2
should be reused		1					4
should not be changed	6	18	22	1	3	41	3
is not implemented		1					
is too new				1			
is not confident		11	19	7		93	37
is used	17	77	33	7	18	153	37
altogether	26	109	77	16	21	290	83

Table 10. Most widely used annotations in ecore

#	Source	Number	Share
1	http://www.eclipse.org/emf/↩ 2002/GenModel	33,056	32.15%
2	http:///org/eclipse/emf/↩ ecore/util/ExtendedMetaData	26,039	25.32%
3	TaggedValues	16,304	15.86%
4	MetaData	10,169	9.89%
5	Stereotype	4,628	4.50%
6	subsets	1,724	1.68%
	...		

The unexpected default value of an attribute which does not require change either *should have no default*, *should not be changed*, *is too new* or *is not confident*. An attribute *should have no default* value if we cannot define a constant default value for the attribute. In each metamodel, we are able to find such attributes which are usually of type String or Integer. A default value *should not be changed* if we could not find a plausible explanation for setting or changing the default value. Two attributes from the unicase metamodel whose default value should not be changed denote whether a job is done. Most of the attribute values are true—denoting a completed job, but in the beginning the job should not be done. We are *not confident* about an attribute if it is not set at least 10 times.

Q7) Which attributes have often used values? Table 9 shows the number of attributes which have often used values or not. The table also classifies the attribute with recurring values according to explanations.

A recurring value which requires metamodel changes either *should be lifted* or *should be reused*. A value *should be lifted* if the value should be represented by a new class in the metamodel. In ecore, we find often used values for the source of an annotation as well as for the key of an annotation entry. Table 10 illustrates the 6 most widely used values of the annotation source. The GenModel annotations customize the code generation, even though there is a separate generator model. Thereby, some of these annotations can be lifted to first-class constructs of the genmodel metamodel. The ExtendedMetaData extends ecore to accommodate additional constructs of XML Schemas which can be imported. This shows the lack of expressiveness of ecore in comparison to XML Schema. The next three sources represent stereotypes and result from the import of metamodels from UML class diagrams. The high number of these cases are evidence that many ecore models originate from UML class diagrams. The last source extends ecore which is an implementation of E-MOF with a subset relation between features which is only available in C-MOF. This shows that for many use cases the expressiveness of E-MOF is not enough. Furthermore, the key of an annotation entry is used in 10% of the cases to specify the documentation of a metamodel element. However, it would be better to make the documentation an explicit attribute of EModelElement—the base class for each metamodel element in ecore.

A value *should be reused* if—instead of recurrently using the value—we refer to a single definition of the value as instance of a new class. In unicase, instead of defining types of UML attributes as Strings, it would be better to group them into separate instances of reusable data types.

An attribute with recurring values which does not require change either *should not be changed*, *is not implemented*, *is too new* or we are *not confident*. For a lot of attributes, we have not found a plausible explanation, and thus conservatively assumed that they *should not be changed*.

4.4 Discussion

Lessons learned. Based on the results of our analyses, we learned a number of lessons about the usage of metamodels by existing models.

Metamodels can be more restrictive. In all investigated cases, the set of models covers only a subset of the set of all possible models that can be built with a metamodel. We discovered that not all classes are instantiated (Q1), not all features are used (Q3), and the range of the cardinality of many features does not reflect their definition from the metamodel (Q4). In all these cases, the metamodels can be improved by restricting the number of admissible models.

Metamodels can contain latent defects. During our experiments, we discovered in several cases defects of the metamodels—e. g. classes that should not be instantiated and are not declared abstract (Q1), classes and features that are not implemented or that are overwritten by the generator (Q1, Q3, Q4, Q6). Moreover, in other cases (Q2), we discovered misuses of metamodel constructs.

Metamodels can be extended. Each of the analyzed metamodels offers their users the possibility to add new information (e. g. as annotations, or sets of key-value pairs). The analysis of the manners in which the metamodels are actually extended reveal that the users recurrently used several annotation patterns. These patterns reveal the need to extend the metamodel with new explicit constructs that capture their intended use (Q2, Q7). Moreover, the specification of some attributes can be extended with the definition of default values (Q6).

Metamodeling formalism can be improved. Our metamodeling formalism is very similar to ecore. The results indicate that in certain cases the metamodeling formalism is not expressive enough to capture certain constraints. For instance, we would have required to mark a class as transient (Q1) and to state that a feature has a default value derived from other features (Q4). Consequently, we have also identified improvements concerning the metamodeling formalism.

Limitations. We are aware of the following limitations concerning our results.

Validity of explanations. We presented a set of explanations for the deviations between expectation and usage. In the case of UNICASE, we had direct access to the language developers and hence could ask them directly about their explanations for the usage problems. In the case of the other metamodels, we interpreted the analysis results based only on the documentation and implementation. Consequently, some of our explanations can be mistaken.

Relevance and number of analyzed models. We analyzed only a subset of the entire number of existing models. This fact can make our results rather questionable. In the case of UNICASE, we asked the developers to provide us with a representative sample of existing models. In the case of the other metamodels, we mined as many models as possible from both public and private (AUTOSAR) repositories to obtain a representative sample of existing models.

5 Related Work

Investigating language utterances. In the case of general-purpose languages, investigating their use is especially demanding, as a huge number of programs exist. There are some landmark works that investigate the use of general-purpose languages [3,17,11]. Gil et al. present a catalog of Java micro patterns which capture common programming practice on the class-level [3]. By automatically searching these patterns in a number of projects, they show that 75% of the classes are modeled after these patterns. Singer et al. present a catalog of Java nano patterns which capture common programming practice on the method-level [17]. There is also work on investigating the usage of domain-specific languages. Lämmel et al. analyze the usage of XML schemas by applying a number of metrics to a large corpus [10]. Lämmel and Pek present their experience with analyzing the usage of the W3C's P3P language [11]. Our empirical results—that many language constructs are more often used than others—are consistent with all these results. We use a similar method for investigating the language utterances, but our work is focused more on identifying improvements for languages. Tolvanen proposes a similar approach for the metamodeling formalism provided by MetaCase [19]. However, even if the sets of analyses are overlapping, our set contains analyses not addressed by Tolvanen's approach (Q2, Q5 and Q7), and provides a validation through a large-scale empirical study that usage analyses do help to identify metamodel improvements.

Language improvements. Once the information about the language use is available, it can serve for language improvements. Atkinson and Kuehne present techniques for language improvements like restricting the language to the used subset or adding new constructs that reflect directly the needs of the language users [1]. Sen et al. present an algorithm that prunes a metamodel [16]: it takes a metamodel as input as well as a subset of its classes and features, and outputs a restricted metamodel that contains only the desired classes and features. By doing this, we obtain a restricted metamodel that contains only necessary constructs. Our work on mining the metamodel usage can serve as input for the pruning algorithm that would generate a language more appropriate to the expectations of its users. Once recurrent patterns are identified, they can serve as source for new language constructs [2]. The results in this paper demonstrate that the analysis of built programs is a feasible way to identify language improvements. Lange et al. show—by performing an experiment—that modeling conventions have a positive impact on the quality of UML models [12]. Henderson-Sellers and Gonzalez-Perez report on different uses and misuses of the stereotype mechanism provided by UML [6]. By analyzing the built models, we might be able to identify certain types of conventions as well as misuses of language extension mechanisms.

Tool usage. There is work on analyzing the language use by recording and analyzing the interactions with the language interpreter. Li et al. [13] present a study on the usage of the Alloy analyzer tool. From the way how the analyzer tool was used, they identified a number of techniques to improve analysis performance. Hage and Keeken [5] present the framework Neon to analyze usage of a programming environment. Neon records data about every compile performed by a programmer and provides a query to analyze the data. To evaluate Neon, the authors executed analyses showing how student programmers improve over time in using the language. The purpose of our approach is not to improve the tools for using the language, but to improve the language itself.

6 Conclusions and Future Work

This paper is part of a more general research direction that we investigate, namely how to analyze the use of modeling languages in order to assess their quality. The focus of this paper is to derive possible improvements of the metamodel by analyzing the language use. We are convinced that the analysis of models built with a modeling language is interesting for any language developer. We showed that—even in the case of mature languages—the analysis of models can reveal issues with the modeling language. Due to the promising results, we plan to further improve the approach presented in this paper. First, we intend to refine the presented analyses by fine-tuning them according to the results from the empirical study. Second, we plan to add new analyses which are currently missing in the catalog. Third, we intend to apply the presented approach as part of a method for the evolutionary development of modeling languages.

For languages used by different organizations, the analysis of models is often impossible due to the intellectual property that the models carry. We envision as a future direction of research the definition of techniques that anonymize the content of the model and send the language developers only a limited information needed for analyzing the language use.

Acknowledgments. This work was partly funded by the German Federal Ministry of Education and Research (BMBF), grants “SPES2020, 01IS08045A” and “Quamoco, 01IS08023B”.

References

1. Atkinson, C., Kühne, T.: A tour of language customization concepts. Advances in Computers 70, 105–161 (2007)
2. Bosch, J.: Design patterns as language constructs. JOOP - Journal of Object-Oriented Programming 11(2), 18–32 (1998)
3. Gil, J., Maman, I.: Micro patterns in Java code. In: OOPSLA 2005: Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 97–116. ACM, New York (2005)
4. Gronback, R.C.: Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. Addison-Wesley, Reading (2009)
5. Hage, J., van Keeken, P.: Neon: A library for language usage analysis. In: Gašević, D., Lämmel, R., Van Wyk, E. (eds.) SLE 2008. LNCS, vol. 5452, pp. 35–53. Springer, Heidelberg (2009)
6. Henderson-Sellers, B., Gonzalez-Perez, C.: Uses and abuses of the stereotype mechanism in UML 1.x and 2.0. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 16–26. Springer, Heidelberg (2006)
7. Herrmannsdoerfer, M., Benz, S., Juergens, E.: COPE - automating coupled evolution of metamodels and models. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 52–76. Springer, Heidelberg (2009)
8. Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schindler, M., Völkl, S.: Design guidelines for domain specific languages. In: The 9th OOPSLA Workshop on Domain-Specific Modeling (2009)
9. Kelly, S., Pohjonen, R.: Worst practices for domain-specific modeling. IEEE Software 26(4), 22–29 (2009)
10. Lämmel, R., Kitsis, S., Remy, D.: Analysis of XML schema usage. In: Conference Proceedings XML 2005 (November 2005)
11. Lämmel, R., Pek, E.: Vivisection of a non-executable, domain-specific language; understanding (the usage of) the P3P language. In: ICPC 2010: 18th International Conference on Program Comprehension. IEEE, Los Alamitos (2010)
12. Lange, C.F.J., DuBois, B., Chaudron, M.R.V., Demeyer, S.: An experimental investigation of UML modeling conventions. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 27–41. Springer, Heidelberg (2006)
13. Li, X., Shannon, D., Walker, J., Khurshid, S., Marinov, D.: Analyzing the uses of a software modeling tool. In: LDTA 2006: 6th Workshop on Language Descriptions, Tools, and Applications, pp. 3–18. Elsevier, Amsterdam (2006)
14. Object Management Group: Meta Object Facility (MOF) core specification version 2.0 (2006), `http://www.omg.org/spec/MOF/2.0/`
15. Paige, R.F., Ostroff, J.S., Brooke, P.J.: Principles for modeling language design. Information and Software Technology 42(10), 665–675 (2000)
16. Sen, S., Moha, N., Baudry, B., Jézéquel, J.-M.: Meta-model pruning. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 32–46. Springer, Heidelberg (2009)
17. Singer, J., Brown, G., Lujan, M., Pocock, A., Yiapanis, P.: Fundamental nano-patterns to characterize and classify java methods. In: LDTA 2009: 9th Workshop on Language Descriptions, Tools and Applications, pp. 204–218 (2009)
18. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0. Addison-Wesley, Reading (2009)
19. Tolvanen, J.P.: Incremental Method Engineering with Modeling Tools: Theoretical Principles and Empirical Evidence. Ph.D. thesis, University of Jyväskylä (1998)
20. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In: Bateni, M. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 600–624. Springer, Heidelberg (2007)

Domain-Specific Modelling Languages with Algebraic Graph Transformations on RDF

Benjamin Braatz and Christoph Brandt

SECAN-Lab, Université du Luxembourg
`benjamin.braatz@uni.lu`, `christoph.brandt@uni.lu`
`http://wiki.uni.lu/secan-lab/`

Abstract. Domain-specific modelling langugages (DSMLs), which are tailored to the requirements of their users, can significantly increase the acceptance of formal (or at least semi-formal) modelling in scenarios where informal drawings and natural language descriptions are predominant today. We show in this paper how the Resource Description Framework (RDF), which is a standard for the fundamental data structures of the Semantic Web, and algebraic graph transformations on these data structures can be used to realise the abstract syntax of such DSMLs. We examine a small DSML for IT infrastructures as an application scenario. From this scenario, we derive distributed modelling, evolution of language definitions, migration of legacy models and integration of modelling languages as key requirements for a DSML framework. RDF and transformation rules are then used to provide a solution, which meets these requirements, where all kinds of modifications—from simple editing steps via model migration to language integration—are realised by the single, uniform formalism of algebraic graph transformation.

1 Introduction

Our motivation for the work presented in this paper is to obtain a framework for the definition and management of families of domain-specific modelling languages (DSMLs). We use the term DSMLs to denote small, flexible, visual and textual languages that are tailored to the needs of their users in a certain application domain. Such languages typically cannot be defined once and for all, but they are in a constant state of flux, since requirements of the stakeholders may become apparent or new requirements may emerge during the life-time of the language. Moreover, the framework should allow families of integrated DSMLs, i. e., several DSMLs—each created for a specific task or a specific group of users—are synchronised on their common overlapping aspects.

In Sect. 2, we present an application scenario for DSMLs, from which we derive seven guiding problems for our framework, where key requirements are evolution and integration of DSMLs.

We use the Resource Decription Framework (RDF), defined in [1], for representing the abstract syntax of DSMLs. This representation is introduced in Sect. 3. RDF is used as the fundamental, generic data structure for the Semantic Web. It is, therefore, well-suited for the distributed management of models in large organisations. With this choice, we expect to reduce the effort that is required for the creation of large integrated models from the knowledge of local users, while also allowing decentralised workflows.

In Sect. 4, we introduce algebraic graph transformations on RDF graphs. Algebraic graph transformations for RDF were proposed and developed in [2,3]. They are used in this paper to provide a single, uniform formalism for all kinds of modifications on models—editing, migration and integration. One of its key advantages is that it can be fully implemented.

In Sect. 5, we show how DSMLs can evolve in our framework and how graph transformations can be used to achieve the migration of models according to such a evolution. In Sect. 6, we present the integration of the languages from the application scenario, again using graph transformations.

Finally, we give comparisons to related work in Sect. 7 and some concluding remarks in Sect. 8.

2 Application Scenario: IT Infrastructure DSML

In this section, we present an application scenario for our proposed DSML framework. It is concerned with a DSML for IT infrastructures, as it may be used by the experts in the field. The language is rather small and simple, but a similar language for the real world would in principle work on the same level of abstraction, tailored to the needs of the users and resembling the informal languages they use today.

In Fig. 1, we see a model of an IT landscape and some editing steps modifying this model. The model shows some local area networks (LANs), the Internet as a wide area network (WAN) and the connections between them, where some of them are protected by firewalls. The local network LAN 1 is connected to the Internet only through a demilitarised zone (DMZ), where both connections are protected. During the editing steps a local network for backup systems is introduced, which is supposed to be reachable through two unprotected connections, which are also added.

Our first problem arises from the requirement that we want to replace the models that are today created by single stakeholders or small teams with integrated models that are created by all relevant stakeholders together in order to provide a common ground for communication.

Problem 1 (Distributed Management of Models). We want to be able to manage the models in a distributed system, such that all stakeholders can work on (parts of) the same model in parallel.

The next two problems are motivated by the requirement that, instead of just using informal drawings, we want to provide techniques for modelling languages with an abstract syntax that is precisely and formally defined.

Problem 2 (Definition of DSMLs). We need a technique to define the language, i. e., which types of language element exist and which configurations of these elements constitute legal models.

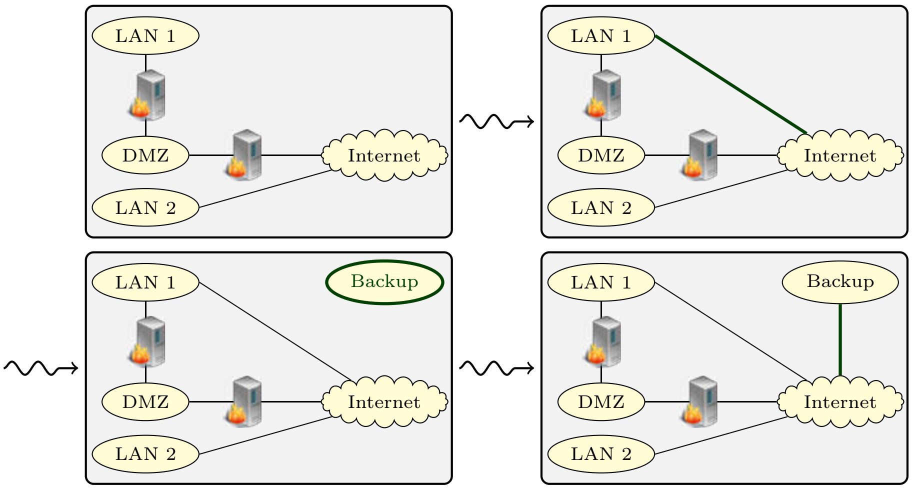


Fig. 1. DSML for IT landscapes—syntax-directed editing

Problem 3 (Syntax-Directed Editing). We want to have a technique to edit models in a syntax-directed manner, i. e., each editing step should be guaranteed to preserve the property of being a legal model according to the language definitions from Problem 2.

The final model of Fig. 1 has the problems that the unprotected connection from LAN 1 to the Internet bypasses the security measures of the DMZ and the Backup network is totally unprotected. In Fig. 2, we eliminate these problems by protecting these connections with a firewall.

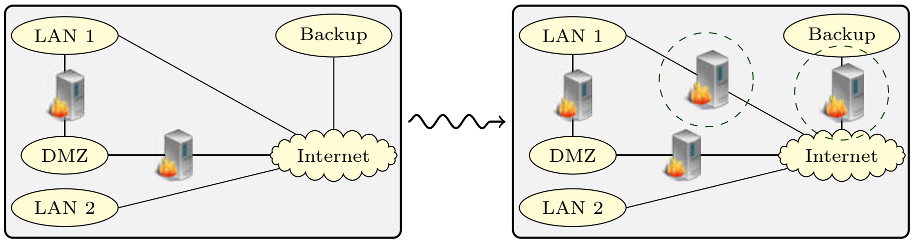


Fig. 2. Protection of connections by firewalls—complex modifications

We want to be able to define such refactorings and other complex modifications of models on an abstract level and allow users to execute them as single editing steps. Such complex modification definitions could also be useful to aggregate best practices, design patterns and knowledge about the construction of models in general.

Problem 4 (Complex Modifications of Models). It should be possible to abstractly define complex modifications and easily instantiate and execute them in concrete situations. The modifications should guarantee to respect the language definitions from Problem 2.

DSMLs change frequently. For example, we could introduce a distinction between public and restricted LANs to our example DSML. This allows to document and visualise the reason why the networks LAN 1 and Backup are protected, but LAN 2 is allowed to have an unprotected connection to the Internet. This is done in Fig. 3, where restricted LANs are visualised by double borders, while the public LAN 2 is shown with a dashed double border.

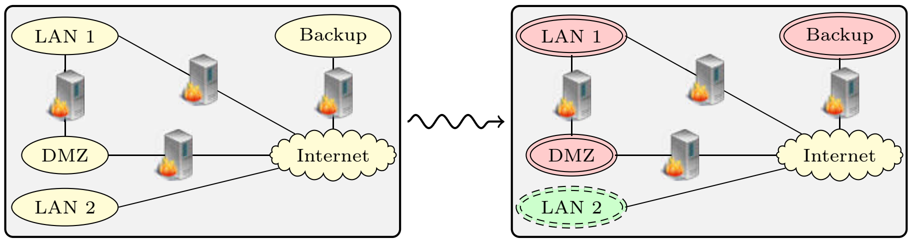


Fig. 3. Public LANs and restricted LANs—language evolution and model migration

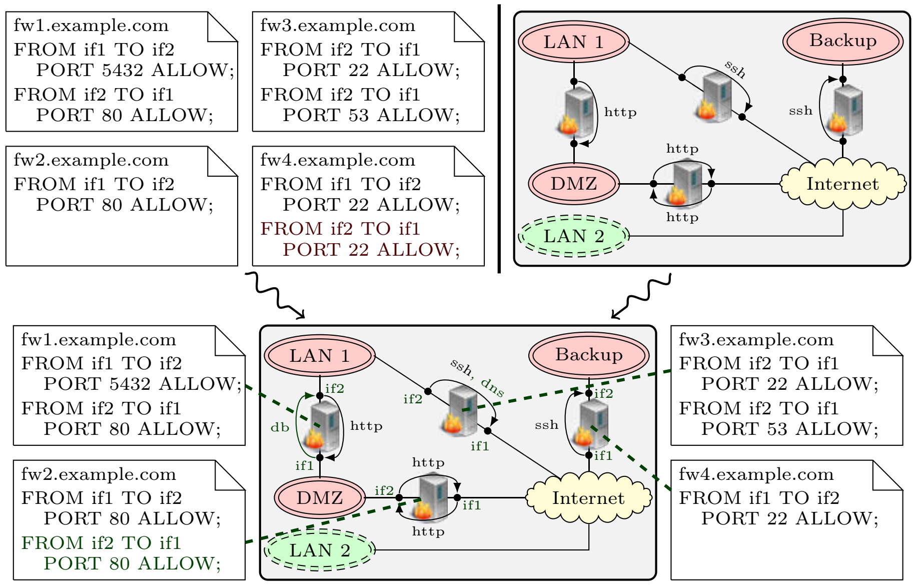
The requirement of allowing language evolution leads to two problems—the evolution to a new language definition itself and the migration of legacy models according to the old definition.

Problem 5 (Evolution of DSMLs). It should be possible with minimal effort to adapt a language according to needs of users.

Problem 6 (Migration of Models). We need a technique to migrate legacy models from a previous language defintion to the evolved definition.

If we have several DSMLs that overlap in certain aspects then we want to be able to integrate them. In Fig. 4, we see an example in our application scenario, where the landscape is enhanced with a notation for the protocols allowed by the firewalls and a second (textual) DSML for the configurations of firewalls is introduced. The integration consists of identifying which configuration snippets belong to which firewall in the landscape and removing and adding configuration lines and protocols in the landscape diagram, such that they match each other.

Problem 7 (Integration of DSMLs). We want to have a technique to integrate multiple DSMLs that overlap in certain aspects.



```
fw1.example.com
FROM if1 TO if2
   PORT 5432 ALLOW;
FROM if2 TO if1
   PORT 80 ALLOW;
```

```
fw2.example.com
FROM if1 TO if2
   PORT 80 ALLOW;
```

```
fw3.example.com
FROM if2 TO if1
   PORT 22 ALLOW;
FROM if2 TO if1
   PORT 53 ALLOW;
```

```
fw4.example.com
FROM if1 TO if2
   PORT 22 ALLOW;
FROM if2 TO if1
   PORT 22 ALLOW;
```

```
fw1.example.com
FROM if1 TO if2
   PORT 5432 ALLOW;
FROM if2 TO if1
   PORT 80 ALLOW;
```

```
fw2.example.com
FROM if1 TO if2
   PORT 80 ALLOW;
FROM if2 TO if1
   PORT 80 ALLOW;
```

```
fw3.example.com
FROM if2 TO if1
   PORT 22 ALLOW;
FROM if2 TO if1
   PORT 53 ALLOW;
```

```
fw4.example.com
FROM if1 TO if2
   PORT 22 ALLOW;
```

Fig. 4. IT landscape and firewall configurations—language integration

3 RDF Graphs: Abstract Syntax for DSMLs

The Resource Description Framework (RDF), defined in [1], provides the fundamental data structures for the Semantic Web. It is used to state facts about resources that are either identified by Uniform Resource Identifiers (URIs) or given directly as literal values. The facts are given by subject–predicate–object triples, where the predicate is also given by a URI. A set of facts is an RDF graph, where the subjects and objects are the nodes and the facts the edges of the graph, labelled with the corresponding predicate (which may also appear as a node). The idea is that everyone can publish such graphs to assert certain facts and these graphs can easily be joined to collect information from heterogeneous sources.

In Fig. 5, we show how such a graph is used to represent part of an IT landscape model. The local net LAN 1 and the Internet are represented by URIs “mod:LAN1” and “mod:INet”, respectively, where “mod:” is a suitable namespace, e. g., “http://models.example.com/”. The names of the networks, which are shown as inscriptions in the concrete visual representation, are given by literal values. The predicate “rdf:type” (abbreviated as “a”) is used to connect nodes with their types. The types as well as the predicates are defined in another namespace “itml:”, which, e. g., might be “http://schema.example.com/”. Technically, it would be possible to use the same namespace for both, the language elements and the model instances, but seperation of namespaces eases the distributed handling of schemas and (multiple) models by different groups of users. The connection between LAN 1 and the Internet is represented by a blank node

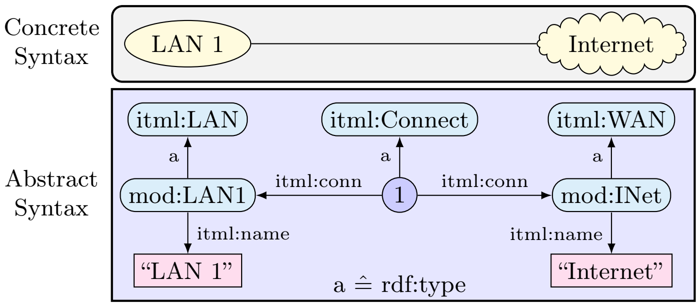


Fig. 5. RDF graph representing the abstract syntax of a DSML model

"1". Blank nodes do not have a global identity and, hence, other graphs cannot state additional facts about entities identified by blank nodes.

The following formal definition of an RDF graph closely corresponds to the one given in [1]. The main difference is that we incorporate a set of blank nodes into the graph, since they are supposed to be local to the corresponding graph. On the other hand, the sets of possible URIs and literals are globally given and occurrences in different graphs are meant to identify the same entity. An implementation should resemble this formal definition and allow arbitrary URIs and literals, while it may internally manage a cache of URIs and literals that are currently used.

Definition 1 (RDF Graph). *An* RDF graph $G = (G_{\mathrm{Bl}}, G_{\mathrm{Tr}})$ *consists of a set* G_{Bl} *of* blank nodes *and a set* $G_{\mathrm{Tr}} \subseteq G_{\mathrm{Nd}} \times \mathrm{URI} \times G_{\mathrm{Nd}}$ *of* triples *(also called* statements *or* edges*), where* $G_{\mathrm{Nd}} := \mathrm{URI} + \mathrm{Lit} + G_{\mathrm{Bl}}$ *is the derived set of* nodes *of the graph and* URI *and* Lit *are globally given sets of all possible Uniform Resource Identifiers (URIs) and literals.*[1] *The constituents* $s \in G_{\mathrm{Nd}}$*,* $p \in \mathrm{URI}$ *and* $o \in G_{\mathrm{Nd}}$ *of a triple* $(s, p, o) \in G_{\mathrm{Tr}}$ *are called* subject*,* predicate *and* object*, respectively.*

Solution 1 (Distribution of RDF Graphs). The use of RDF provides a solution of Problem 1 of Sect. 2, since RDF is designed for the Semantic Web and, hence, provides excellent features for managing and relating distributed models. More specifically, the use of URIs as nodes enables us to distribute the creation of language definitions and the maintenance of models, where the use of URIs and namespaces facilitates seperation of concerns, while still allowing to relate the models and language definitions through the use of URI references.

In RDF, *vocabularies* are used to define types that are supposed to be used as objects of rdf:type triples. The vocabulary for our IT landscape language is given in Fig. 6, where we also include visualisation hints in the vis: namespace in order

[1] The operations $\times$ and $+$ denote the cartesian product and the disjoint union of sets, respectively, where the disjointness ensures that we can determine if an element of G_{Nd} is a URI, a literal or a blank node.

to allow the creation of generic visual clients. Vocabularies may also be called *schemas* or *ontologies*, where we choose the term vocabulary, since it is the most neutral one. In contrast to a *meta model*, a vocabulary has no possibilities to constrain the structure of instances.

In [4], RDFS is introduced – a vocabulary for describing vocabularies. We use some of the terms from RDFS, namely rdfs:subClassOf to define type hierarchies and rdfs:domain and rdfs:range to define the domain and range of properties. RDFS is supposed to be used together with tools that can draw inferences from these information, e. g., adding additional super-types to an element or the domain of a property to an element that is the subject of a corresponding triple, but we do not rely on this in this paper and just use the RDFS terms descriptively.

The Web Ontology Language (OWL 2), defined in [5], introduces possibilities for constraints, similar to the ones used in meta modelling. With these, structures can not only be inferred, but also forbidden in order to meet the requirements of a schema. In our approach, we will, however, not use OWL 2, but grammars based on graph transformation rules, which are introduced in the next section.

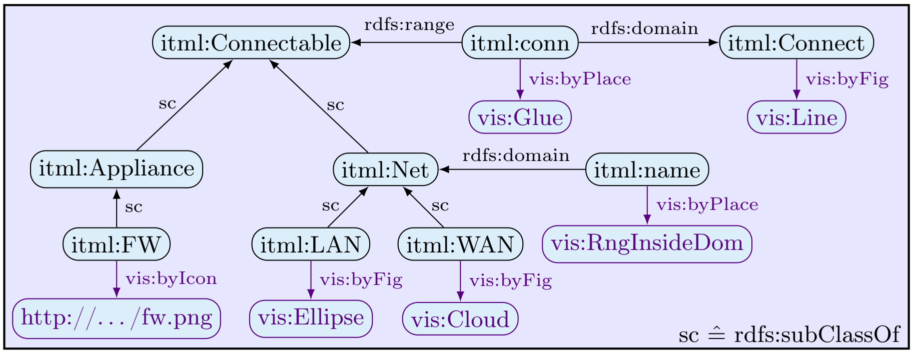


Fig. 6. Vocabulary for IT landscapes with visualisation hints

Compared to the definition of languages by meta modelling, e. g., using the Meta Object Facility (MOF), defined in [6], the approach chosen here is very lean and light-weight. RDF graphs are supposed to be defined in a distributed manner on the Semantic Web with the help of heterogeneous vocabularies. In contrast to that, MOF enforces that every model strictly conforms to a meta model, which, therefore, has to anticipate all needs of the users. Thus, the features of RDF ideally reflect the flexibility requirements of DSMLs.

In [7], proposals for a mapping from MOF to RDF are requested. Such a mapping would allow the representation of MOF meta models and corresponding models as RDF graphs and, hence, facilitate the application of the methods introduced in the present paper to MOF meta models and models.

4 RDF Graph Transformation: Grammars and Editing

We use rule-based, algebraic graph transformations to describe all kinds of changes on RDF graphs. A comprehensive overview over the theory of algebraic graph transformation can be found in [8]. The adaption of this theory to RDF was proposed in [2] and continued in [3].

We choose algebraic graph transformation, since it allows to treat all kinds of transformations from language definition by grammars via model migration to language integration with a single, uniform formalism. Being a formal technique, it also allows to reason formally about the effects of transformations to show, e. g., that certain derived transformation rules respect a given grammar or that transformations are independent of each other and can, hence, be swapped without affecting the result of the combined transformation.

Transformations are defined by transformation rules. An example of such a rule is shown in Fig. 7. This rule removes a connection between two networks, represented by variables x and y and adds a new connection from network x to network z, where z is not allowed to be of type itml:WAN and it is not allowed to introduce a self connection, i. e., to assign x and z to the same net.

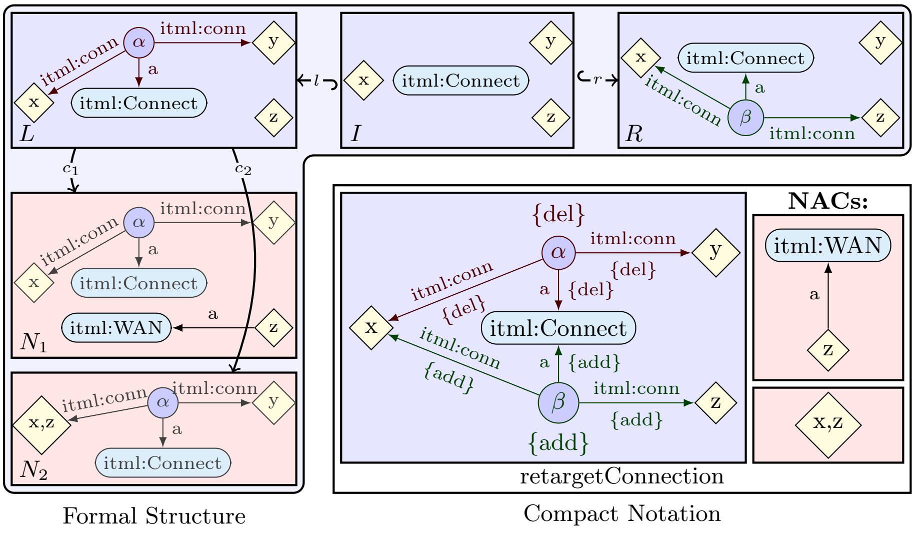


Fig. 7. Example transformation rule retargetConnection

The rule consists of several RDF patterns, which are graphs with additional variables, and RDF pattern homomorphisms, which are structure preserving maps connecting the patterns, where the homomorphisms l and r are injective, i. e., one-to-one (visualised by the hook at the tail of the arrows). The difference between the left-hand side L and the interface I are the elements of the pattern that are supposed to be deleted and the difference between I and the right-hand side R are the elements that are supposed to be added by the rule. Moreover, there is a set of negative application conditions (NACs), which are extensions of L and specify situations in which the rule is not applicable, where the NACs are in an implicit conjunction, i. e., all have to be satisfied and none of the situations is allowed. In the lower right, we show a compact notation for the rule, where irrelevant parts of the NACs are omitted and L, I and R are shown in a single diagram with the additional elements of L marked by "{del}" and the additional elements of R by "{add}".

We now give the formal definition for RDF patterns and RDF pattern homomorphisms. Observe that there are two different kinds of variables, one that can be assigned to URIs, literals and blank nodes and one that can only be assigned to URIs, where only the latter kind can be used as predicates of triples. This is necessary, because otherwise we would have to deal with inconsistencies due to variables being on the one hand assigned to literals or blanks and on the other hand used as predicates.

Definition 2 (RDF Pattern and RDF Pattern Homomorphism). *An* RDF pattern $P = (P_{\mathrm{Bl}}, P_{\mathrm{V}}, P_{\mathrm{U}}, P_{\mathrm{Tr}})$ *consists of a set* P_{Bl} *of blank nodes, a set* P_{V} *of* variables*, a set* $P_{\mathrm{U}} \subseteq P_{\mathrm{V}}$ *of* URI variables *and a set* $P_{\mathrm{Tr}} \subseteq P_{\mathrm{Nd}} \times (\mathrm{URI} + P_{\mathrm{U}}) \times P_{\mathrm{Nd}}$ *of triples, where* $P_{\mathrm{Nd}} := \mathrm{URI} + \mathrm{Lit} + P_{\mathrm{Bl}} + P_{\mathrm{V}}$ *is the derived set of nodes.*

An RDF pattern homomorphism $h\colon P \to Q$ *between RDF patterns P and Q consists of a blank node function* $h_{\mathrm{Bl}}\colon P_{\mathrm{Bl}} \to Q_{\mathrm{Bl}}$ *and a* variable assignment $h_{\mathrm{V}}\colon P_{\mathrm{V}} \to Q_{\mathrm{Nd}}$*, such that* $h_{\mathrm{V}}(P_{\mathrm{U}}) \subseteq \mathrm{URI} + Q_{\mathrm{U}}$ *and* $h_{\mathrm{Tr}}(P_{\mathrm{Tr}}) \subseteq Q_{\mathrm{Tr}}$*, where* h_{Tr} *is the derived translation of triples given by the composed functions* $h_{\mathrm{Tr}} := h_{\mathrm{Nd}} \times (\mathrm{id}_{\mathrm{URI}} + h_{\mathrm{V}}) \times h_{\mathrm{Nd}}$ *and* $h_{\mathrm{Nd}} := \mathrm{id}_{\mathrm{URI}} + \mathrm{id}_{\mathrm{Lit}} + h_{\mathrm{Bl}} + h_{\mathrm{V}}$*.*[2]

We can now formally define transformation rules. The difference between blank nodes and variables is that blank nodes can be deleted and added by the rule, while the variable sets are required to stay the same during the whole transformation. This is required for theoretical reasons, since variables may be instantiated to URIs and literals, which are globally given and, hence, cannot be deleted or added.

Definition 3 (Transformation Rule). *A* transformation rule $tr = (L, I, R, l, r, NAC)$ *consists of RDF patterns L, I and R, called* left-hand side*,* interface *and* right-hand side*, respectively, RDF pattern homomorphisms* $l\colon I \to L$ *and* $r\colon I \to R$*, where the blank node functions* l_{Bl} *and* r_{Bl} *are injective (one-to-one) and the variable assignments* l_{V} *and* r_{V} *are injective with* $l_{\mathrm{V}}(I_{\mathrm{V}}) = L_{\mathrm{V}}$*,* $r_{\mathrm{V}}(I_{\mathrm{V}}) = R_{\mathrm{V}}$*,* $l_{\mathrm{V}}(I_{\mathrm{U}}) = L_{\mathrm{U}}$ *and* $r_{\mathrm{V}}(I_{\mathrm{U}}) = R_{\mathrm{U}}$*, i. e., the variable sets remain essentially the same, and a set NAC of* negative application conditions *(NACs)* $(N, c) \in NAC$*, where N is an RDF pattern and* $c\colon L \to N$ *is an RDF pattern homomorphism.*

In Fig. 8, we show the application of the transformation rule from Fig. 7 to an example graph. The application is determined by a match homomorphism m from the left-hand side L to the graph G.

[2] We use the symbols $\times$ and $+$ also for functions, where they denote the obvious generalisations from operations on sets to operations on functions between correspondingly created sets. Moreover, id_S denotes the identity function on a set S.

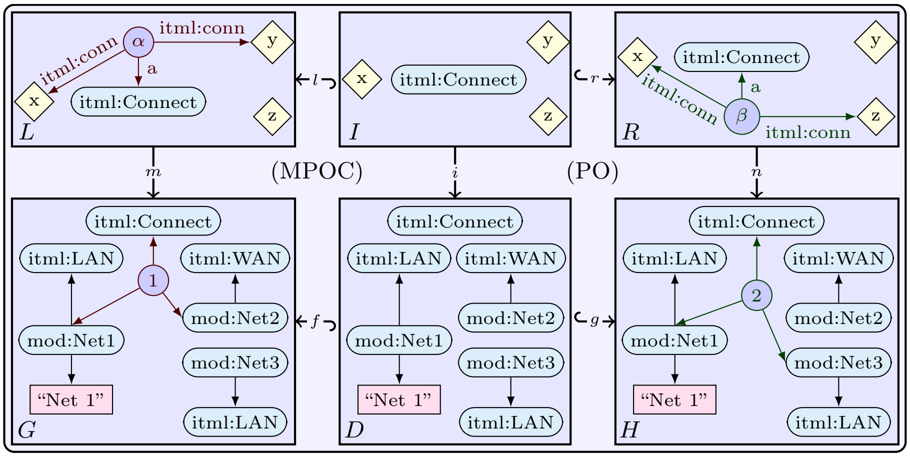


Fig. 8. Application of transformation rule retargetConnection

Formally, a rule is applicable via a match if different deleted elements are not identified, i. e., assigned to the same element, by the match, deleted elements are not connected to additional structure, which would "dangle" if the deletion would be executed and all the NACs are satisfied, i. e., none of the structures forbidden by a NAC are present.

Definition 4 (Applicability of Transformation Rule). *Given a transformation rule $tr = (L, I, R, l, r, NAC)$ and an RDF pattern homomorphism $m\colon L \to G$, called* match, *tr is* applicable *via m if the dangling condition (1), the identification condition (2) and all NACs (3) are satisfied by l and m.*

(1) The dangling condition *is satisfied by l and m if there is no deleted blank node $b \in L_{\mathrm{Bl}} \setminus l_{\mathrm{Bl}}(I_{\mathrm{Bl}})$ and non-deleted triple $(s, p, o) \in G_{\mathrm{Tr}} \setminus m_{\mathrm{Tr}}(L_{\mathrm{Tr}})$ with $m_{\mathrm{Bl}}(b) = s$ or $m_{\mathrm{Bl}}(b) = o$.*
(2) The identification condition *is satisfied by l and m if there is no deleted blank node $b \in L_{\mathrm{Bl}} \setminus l_{\mathrm{Bl}}(I_{\mathrm{Bl}})$, such that there is another blank node $b' \in L_{\mathrm{Bl}}$ with $m_{\mathrm{Bl}}(b) = m_{\mathrm{Bl}}(b')$ or a variable $v \in L_{\mathrm{V}}$ with $m_{\mathrm{Bl}}(b) = m_{\mathrm{V}}(v)$.*
(3) A NAC $(N, c\colon L \to N) \in NAC$ is satisfied by m if there is no occurrence homomorphism $o\colon N \to G$ with $o \circ c = m$.

The application of a transformation rule is formally defined in terms of pushouts and pushout complements. Pushout (PO) is an abstract notion of category theory that intuitively corresponds to a disjoint union over a common interface. This is used to glue the new elements of the right-hand side R to the context graph D over the interface I. Pushout complements are then those objects that extend a given situation to become a pushout. This is used to obtain the context object D from I, L and G, where there may be several pushout

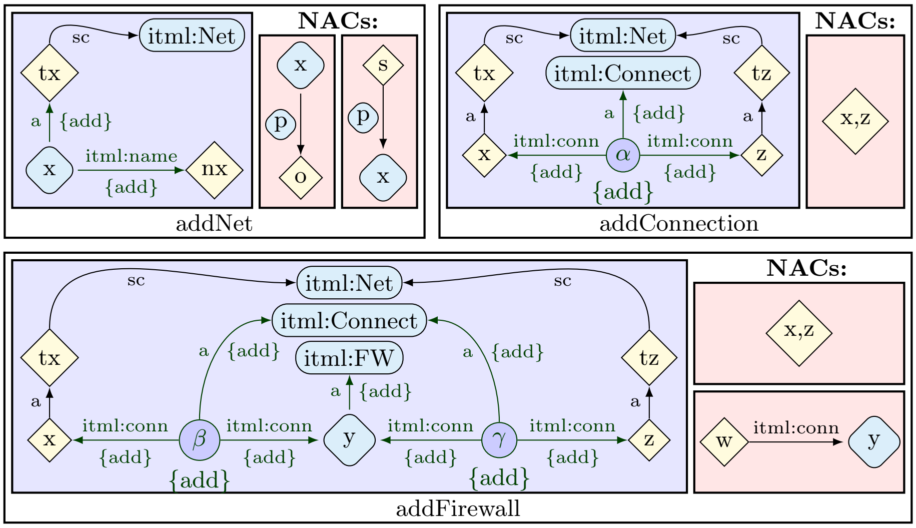


Fig. 9. Grammar for IT landscape DSML

complements and we choose the smallest of them, the minimal pushout complement (MPOC).

Definition 5 (Application of Transformation Rule). *Given a transformation rule* $tr = (L, I, R, l, r, NAC)$ *that is applicable via the match* $m: L \to G$, *the* application $G \overset{tr,m}{\Longrightarrow} H$ *is determined by first constructing a* minimal pushout complement *(MPOC)* (D, f, i) *of* l *and* m *and then a* pushout *(PO)* (H, g, n) *of* r *and* i.

Solution 2 (Grammars for Language Definition). Problem 2 of Sect. 2 is solved by using a set of RDF graph transformation rules as a *grammar* for a DSML. Legitimate models of the language are then all graphs that can be created by rules from this grammar starting from a graph that only contains the vocabulary. For example, the language from our application scenario is specified by the rules in Fig. 9. The variables x and z are used to represent networks, where there concrete types are represented by the variables tx and tz, whose assignments have to be subclasses of itml:Net. This facilitates the use of the same rules for all combinations of itml:LAN and itml:WAN instances. The NACs of the rule addNet, which introduces a new network, ensure that the URI that is used to identify the new network is not already used in any triples. The NACs of the rules addConnection and addFirewall, which introduce direct connections and protected connectios, respectively, ensure that no loops from a network to itself are created and the URI that is used for a new firewall is not already connected. The connections are represented by blank nodes, which are freshly created by the rules.

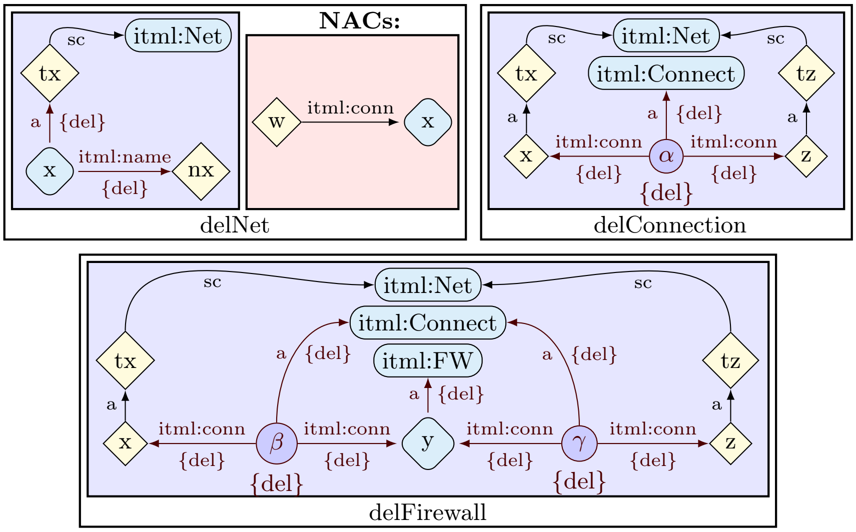


Fig. 10. Deletion rules for IT landscape DSML

Solution 3 (Grammars and Deletion Rules for Syntax-Directed Editing). In order to solve Problem 3 of Sect. 2, we also define deletion rules, such that a user can freely remove and add elements from and to a model. The deletion are in a one-to-one correspondence with the rules of the grammar, but the NACs are different. In order to obtain the NAC for a deletion of rule, we have to consider in which other grammar rules the structure is needed, since the application of these other rules may be invalidated by the deletion. There is hope to derive these deletion rules automatically, but this process is non-trivial.
A set of deletion rules corresponding to the grammar from Fig. 9 is given in Fig. 10. While connections and firewalls can be deleted in all circumstances, the NAC of the rule delNet ensures that the deleted network is not connected to another network or a firewall. This is necessary, because otherwise the deletion could lead to an illegal model, since the grammar only allows to create connections to nodes which have a network type.

Solution 4 (Composition for Sound Modification Rules). In order to solve Problem 4 of Sect. 2, we use composition of transformation rules to obtain more complex transformation rules. The composition ensures that each application of the composed rule corresponds exactly to sequential applications of the component rules. Hence, the language is not changed by additionally allowing rules that are compositions of grammar rules.

For example, the rules delConnection and addFirewall can be composed to obtain the rule protect in Fig. 11, which facilitates the protection of an unprotected connection in one transformation step.

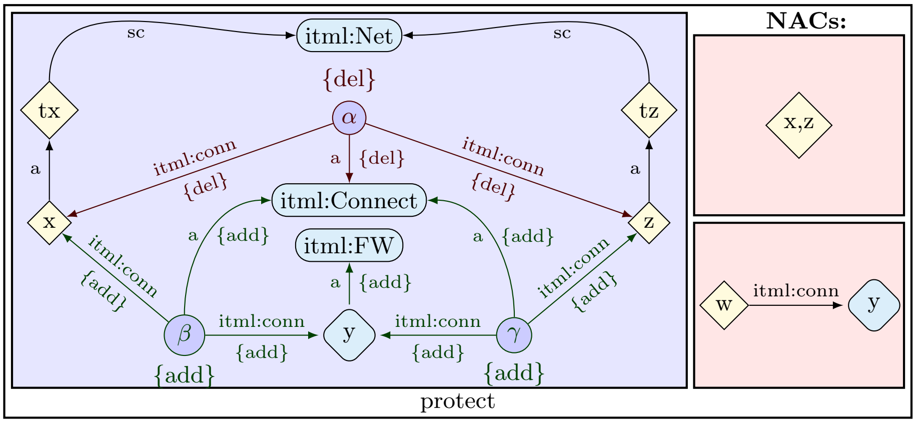


Fig. 11. Refactoring rule protecting a connection with a firewall

The use of rule-based graph transformations for modifying RDF graphs has several advantages. An implementation of a transformation engine may be reused for a multitude of purposes, where the search for matches and the transformation only have to be implemented once and for all. Modifications are specified on an adequate level of abstraction and automatically respect a given grammar if they are composed from it, which is also the main advantage of using grammars for language definition instead of meta modelling or OWL 2 constraints. Moreover, a single, uniform formalism can be used for language definition, syntax-directed editing and complex modifications. In the following sections, we also show how they can be used to allow model migration and language integration.

5 Evolution of Languages and Migration of Models

Solution 5 (Vocabulary Extension and Grammar Modification for Language Evolution). We solve Problem 5 of Sect. 2 by adding the new types and visualisation hints shown in Fig. 12 to the vocabulary of our DSML and the NACs shown in Fig. 13 to the corresponding rules of the grammar. In general, the modifications to the grammar could be more profound—adding completely new rules or removing them completely, but in this case this is not necessary, since the rule addNet is already designed to cover all subclasses of itml:Net. We just have to ensure that the deprecated type itml:LAN is not used anymore and that the intended meaning of the new type itml:Restr for restricted LANs—no direct, unprotected connection to WANs is allowed—is respected when connections are added to the model.

Solution 6 (Rules for Model Migration). In order to solve Problem 6 of Sect. 2, we define the rules in Fig. 14, which migrate legacy models that still contain the now forbidden type itml:LAN. The rules replace this typing either by itml:Public

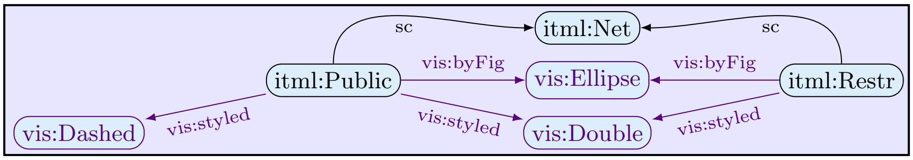


Fig. 12. New vocabulary with visualisation hints

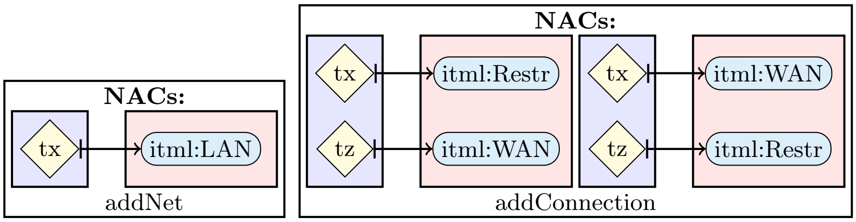


Fig. 13. Additional NACs in grammar for evolved language

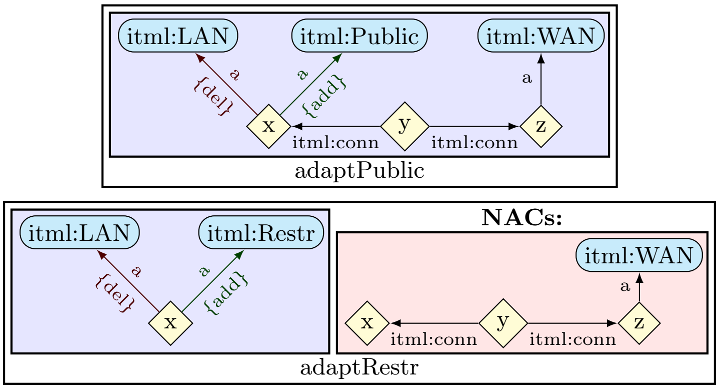


Fig. 14. Model migration rules

if there already is a connection to a WAN or by itml:Restr if there is no connection to a WAN. This is a design decision, since we could also retype all LANs to itml:Public, which has no further restrictions in the new grammar.

These rules can be applied on demand or automatically allowing us to migrate legacy models according to the practical needs of the organisation. Specifically, it is possible to execute the migration rules automatically on (parts of) the whole model repository or on demand only if and when legacy elements are encountered. This choice solely depends on the size of the repository and organisational requirements like availability of the model. The migration rules are equally well-suited for both approaches.

6 Integration of Languages

In Fig. 15, we show how the additional notation for protocols in the IT landscape language and the textual firewall configuration language are represented in RDF.

Since, the landscape language uses the names of protocols and the configuration language uses port numbers, we need a mapping between them, before we can start our integration effort. This mapping is given in Fig. 16, where blank nodes with corresponding int:prot and int:port predicates are used to represent this relation.

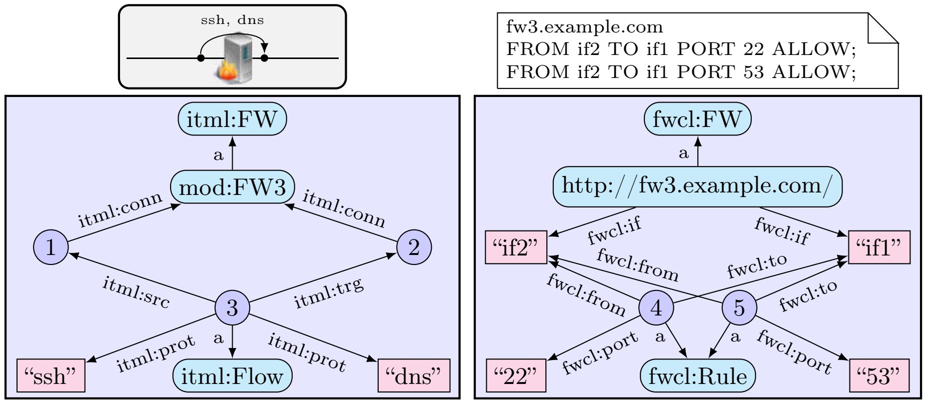


Fig. 15. RDF representation of firewall configurations

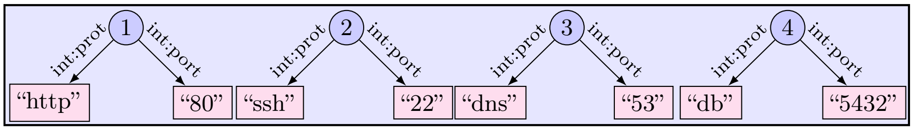


Fig. 16. Mapping between protocols and ports

Solution 7 (Rules for Language Integration). For the solution of Problem 7 of Sect. 2, we define some sets of RDF graph transformation rules.

First, the rules in Fig. 17 are used to *manually* establish the connection between the firewalls in the landscape model and the corresponding configuration

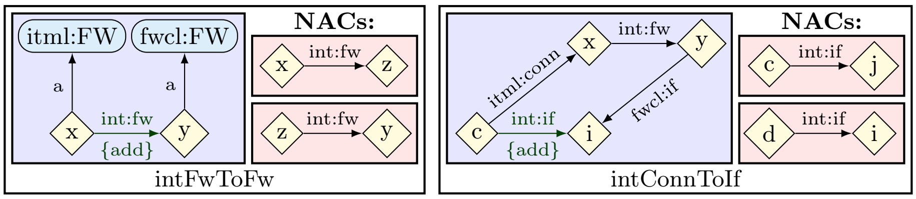


Fig. 17. Manual integration rules

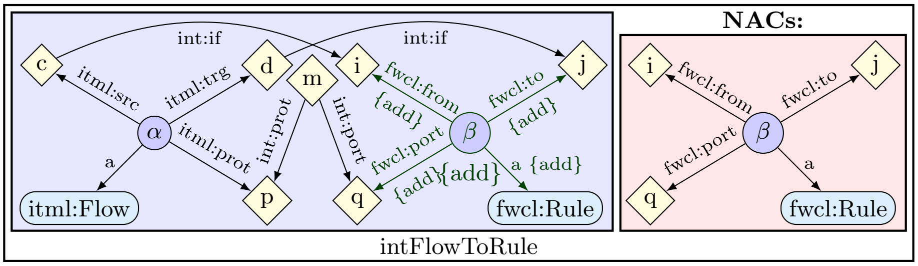


Fig. 18. Automatic integration rule

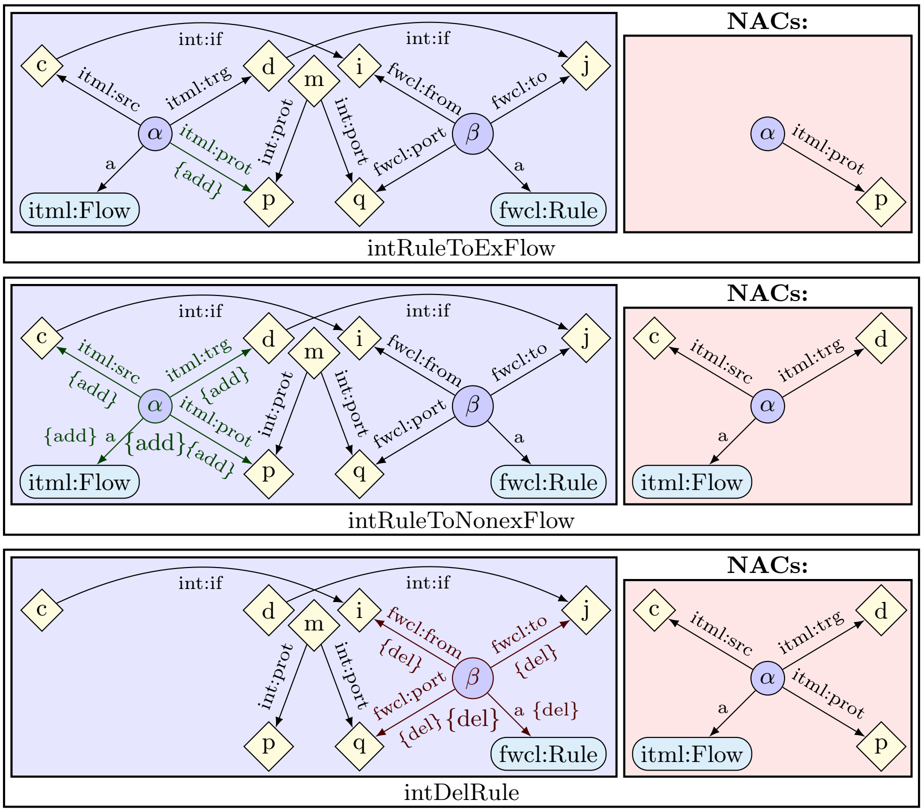


Fig. 19. Semi-automatic integration rules

language snippets and the connections in the landscape and the corresponding interfaces in the configurations. This has to be done manually, since there is not enough information in the models to deduce these correspondences automatically. It is a constructive choice.

The rule in Fig. 18 can be used to *automatically* add lines to the firewall configurations, where there is an additional protocol that is allowed according to the landscape model. This rule can be applied as long as possible, since we decide to consider the landscape model as superior in these situations.

If there is a line in the firewall configurations that has no corresponding protocol in the landscape model then the rules in Fig. 19 are used. These rules are supposed to be applied *semi-automatically.* The matches can be found automatically, but in each case two of the rules are applicable and the user has to decide which should be applied. Either the protocol is added to the landscape model by one of the first two rules—intRuleToExFlow if there is already another protocol in the same direction, intRuleToNonexFlow if the arrow also needs to be created—or the line is deleted from the firewall configuration by the third rule intDelRule.

We now compare this rule-based model integration with the process of manually integrating models or documents. In any case, the search for inconsistencies is replaced by the automatic match-finding for the integration rules. For the automatic integration rules from Fig. 18, the integration can be completely executed without user intervention, while for the semi-automatic integration rules from Fig. 19, the user still has to decide which of the two possibilities should be executed. When trying to manually integrate models, the much more error-prone task of constructing inconsistency eliminations is required, while these eliminations are given once and for all in the rule-based approach. Thus, the integration by rule-based graph transformation leads not only to less effort for the integration but also to quality gains by reducing missed inconsistencies and inappropriate eliminations.

7 Related Work

We like to focus on the main differences between the presented approach of how domain-specific modelling languages can be administrated by formal techniques of algebraic graph transformation and already available solutions in the area of the related work. In detail, priority is put here on the evolution and mitigation of models and meta-models, the realisation of complex editing operations, akin re-factoring operations, as well as the underlying tools. This focus was chosen because we believe that the RDF approach shows good potential to support the issues discussed in the following in an ideal way.

From a general point of view our approach builds on top of a generic implementation that realises operations for RDF graph transformation and is able to run different vocabularies and grammars that are used to create models. In contrast to that today's solutions often take a specific meta-model of a domain-specific modelling language which is used to build language specific tools that help to create and administrate models of such a meta-model. Therefore, today's solutions are in general not generic.

In [9], MetaEdit+ primarily focusses on meta-modelling and code generation. By doing this it increases the productivity by more than 750% and produces high quality code. Meta-modelling is of a similar expressiveness as the grammar-based approach we presented. However, complex operations that result from rule compositions cannot be realised by the approach of MetaEdit+. In addition to that, code generation can be realised by transformation rules, too, even though, it has not yet been defined for the RDF transformation approach.

In [10], some common pitfalls regarding the development of domain-specific modelling languages are mentioned. Probably, the key insight here is that the grammar based approach we presented helps to avoid some of these problems. Because of the extensibility and changeability of grammar rules as well as vocabularies, the resulting models as well as the language definition can be evolved interactively starting from a small core language. So, the risk of developing over-engineered domain-specific modelling languages can be reduced. Secondly, because of the common representation of the abstract syntax of models by RDF graphs the coupling of models is strongly facilitated.

In [11], problems like the example-driven development of domain-specific modelling languages as well as the induced need for tool updates are mentioned. Our RDF approach is able to address these issues smoothly because it does not assume any meta model, just a grammar rule set and a vocabulary. It can therefore support an example-driven language development much better. Further, a tool that can handle models represented by the help of RDF graphs does not need to be updated because the definition of RDF will remain stable even when definitions of domain-specific modelling languages or their artifacts are evolving.

In [12], a detailed study regarding the evolution of languages and models is presented by looking at the GMF framework. The key insight here is that the identified need for an operator driven evolution of models can be ideally supported by the formal techniques of algebraic graph transformation whereas this requires quite some work in the GMF framework.

In [13], the authors discuss the idea of language extensions without changing the tool environment. This can be ideally supported by our RDF approach by extending the grammar rule set or the vocabulary. A generic modelling tool that is able to process RDF graphs would not need to be updated in case of such changes.

In [14], the need for modularisation of language definitions of domain-specific modelling languages is stressed to support extension and reuse. Our RDF based approach can smoothly do this by providing the concept of sub-grammars as well as a lego-like system of already defined specifications of parts of a domain-specific modelling language.

In [15], the need for the handling of language families is discussed. It is further mentioned that by configuration domain-specific modelling languages should be instantiated out of a language family. Our approach shows good potential to be able to handle these requirements which is, however, left for future work.

8 Summary and Future Work

In this paper, we have presented a framework for domain-specific modelling languages (DSMLs). The requirements were derived from a small application scenario for DSML families and we have shown how to meet these requirements using RDF and algebraic graph transformation.

Since our proposal is a whole new framework for domain-specific modelling as an alternative to existing MOF-based and other frameworks, we are currently implementing tool support for this framework, where the models are stored and transformed in an RDF triple store that functions as a model repository. This repository will then be accessed by small, generic clients, which also have the task of visualising the models according to the visualisation hints show in the present paper.

With the help of a MOF to RDF mapping, refactoring and migration rules could be given directly for MOF models. On the other hand, they could be translated to grammar-based models by translation rules or integrated with native grammar-based models in a way that is very similar to the one in Section 6. A detailed treatment of this is, however, left for future work.

Another interesting line of research are the relations between graph transformations and the semantics of RDFS and OWL 2, where graph transformation rules can be used to implement the inferences themselves, but transformations on top of inference engines should also be considered.

References

1. Klyne, G., Carroll, J.J.: Resource Description Framework (RDF): Concepts and Abstract Syntax. World Wide Web Consortium (W3C) (February 2004), `http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/`
2. Braatz, B., Brandt, C.: Graph transformations for the Resource Description Framework. In: Ermel, C., Heckel, R., de Lara, J. (eds.) Proc. GT-VMT 2008. Electronic Communications of the EASST, vol. 10 (2008), `http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/view/158`
3. Braatz, B.: Formal Modelling and Application of Graph Transformations in the Resource Description Framework. PhD thesis, Technische Universität Berlin (2009)
4. Brickley, D., Guha, R.V.: RDF Vocabulary Description Language 1.0: RDF Schema. World Wide Web Consortium (W3C) (February 2004), `http://www.w3.org/TR/2004/REC-rdf-schema-20040210/`
5. W3C OWL Working Group: OWL 2 Web Ontology Language. Document Overview. World Wide Web Consortium (W3C) (October 2009), `http://www.w3.org/TR/2009/REC-owl2-overview-20091027/`
6. Object Management Group (OMG): Meta Object Facility (MOF) Core Specification (January 2006), `http://www.omg.org/spec/MOF/2.0/`
7. Object Management Group (OMG): Request for Proposal. MOF to RDF Structural Mapping in support of Linked Open Data (December 2009), `http://www.omg.org/cgi-bin/doc?ad/2009-12-09`
8. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. Springer, Heidelberg (2006), doi:10.1007/3-540-31188-2

9. Kärnä, J., Tolvanen, J.P., Kelly, S.: Evaluating the use of domain-specific modeling in practice. In: Rossi, M., Sprinkle, J., Gray, J., Tolvanen, J.P. (eds.) Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling, DSM 2009, Number B-108 in HSE Print (2009), http://www.dsmforum.org/events/DSM09/Papers/Karna.pdf
10. Sprinkle, J., Mernik, M., Tolvanen, J.P., Spinellis, D.: What kinds of nails need a domain-specific hammer? IEEE Software 26(4), 15–18 (2009), doi:10.1109/MS.2009.92
11. Kelly, S., Pohjonen, R.: Worst practices for domain-specific modeling. IEEE Software 26(4), 22–29 (2009), doi:10.1109/MS.2009.109
12. Herrmannsdoerfer, M., Ratiu, D., Wachsmuth, G.: Language evolution in practice: The history of GMF. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 3–22. Springer, Heidelberg (2010), doi:10.1007/978-3-642-12107-4_3
13. Bagge, A.H.: Yet another language extension scheme. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 123–132. Springer, Heidelberg (2010), doi:10.1007/978-3-642-12107-4_9
14. Wende, C., Thieme, N., Zschaler, S.: A role-based approach towards modular language engineering. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 254–273. Springer, Heidelberg (2010), doi:10.1007/978-3-642-12107-4_19
15. Zschaler, S., Kolovos, D.S., Drivalos, N., Paige, R.F., Rashid, A.: Domain-specific metamodelling languages for software language engineering. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 334–353. Springer, Heidelberg (2010), doi:10.1007/978-3-642-12107-4_23

Feature and Meta-Models in Clafer: Mixed, Specialized, and Coupled

Kacper Bąk[1], Krzysztof Czarnecki[1], and Andrzej Wąsowski[2]

[1] Generative Software Development Lab, University of Waterloo, Canada
{kbak,kczarnec}@gsd.uwaterloo.ca
[2] IT University of Copenhagen, Denmark
wasowski@itu.dk

Abstract. We present *Clafer*, a meta-modeling language with first-class support for feature modeling. We designed Clafer as a concise notation for meta-models, feature models, mixtures of meta- and feature models (such as components with options), and models that couple feature models and meta-models via constraints (such as mapping feature configurations to component configurations or model templates). Clafer also allows arranging models into multiple specialization and extension layers via constraints and inheritance. We identify four key mechanisms allowing a meta-modeling language to express feature models concisely and show that Clafer meets its design objectives using a sample product line. We evaluated Clafer and how it lends itself to analysis on sample feature models, meta-models, and model templates of an E-Commerce platform.

1 Introduction

Both feature and meta-modeling have been used in software product line engineering to model variability. Feature models are tree-like menus of mostly Boolean—but sometimes also integer and string—configuration options, augmented with cross-tree constraints [22]. These models are typically used to show the variation of *user-relevant* characteristics of products within a product line. In contrast, meta-models, as supported by the Meta Object Facility (MOF) [28], represent concepts of—possibly domain-specific—modeling languages, used to represent more detailed aspects such as behavioral or architectural specification. For example, meta-models are often used to represent the components and connectors of *product line architectures* and the valid ways to connect them. The nature of variability expressed by each type of models is different: feature models capture simple selections from predefined (mostly Boolean) choices within a fixed (tree) structure; and meta-models support making new structures by creating multiple instances of classes and connecting them via object references.

Over the last eight years, the distinction between feature models and meta-models (represented as class models) has been blurred somewhat in the literature due to 1) feature modeling extensions, such as *cardinality-based feature modeling* [15, 4], or 2) attempts to express feature models as class models in Unified Modeling Language (UML) [11, 16]; note that MOF is essentially the class modeling subset of UML. A key driver behind these developments has been the desire to express components and configuration options in a single notation [14]. Cardinality-based feature modeling achieves this by extending feature models with multiple instantiation and references. Class modeling, which natively supports multiple instantiation and references, enables feature modeling by a stylized use of composition and the profiling mechanisms of MOF or UML.

Both developments have notable drawbacks, however. An important advantage of feature modeling as originally defined by Kang et al. [22] is its simplicity; several respondents to a recent survey confirmed this view [23]. Extending feature modeling with multiple instantiation and references diminishes this advantage by introducing additional complexity. Further, models that contain significant amounts of multiply-instantiatable features and references can be hardly called feature models in the original sense; they are more of class models. On the other hand, whereas the model parts requiring multiple instantiation and references are naturally expressed as class models, the parts that have feature-modeling nature cannot be expressed elegantly in class models, but only clumsily simulated using composition hierarchy and certain modeling patterns.

We present *Clafer* (class, feature, reference), a meta-modeling language with first-class support for feature modeling. The language was designed to naturally express meta-models, feature models, mixtures of meta- and feature models (such as components with options), and models that couple feature models with meta-models and their instances via constraints (such as mapping feature configurations to component configurations or to *model templates* [13]). Clafer also allows arranging models into multiple specialization and extension layers via constraints and inheritance, which we illustrate using a sample product line.

We developed a translator from Clafer to Alloy [19], a class modeling language with a modern constraint notation. The translator gives Clafer precise translational semantics and enables model analyses using Alloy Analyzer. Different strategies are applied for distinct model classes. They all preserve meaning of the models, but speed up analysis by exploiting the Alloy constructions.

We evaluate Clafer analytically and experimentally. The analytic evaluation argues that Clafer meets its design objectives. It identifies four key mechanisms allowing a meta-modeling language to express feature models concisely. The experimental evaluation shows that a wide range of realistic feature models, meta-models, and model templates can be expressed in Clafer and that useful analyses can be run on them within seconds. Many useful analyses such as consistency checks, element liveness, configuration completion, and reasoning on model edits can be reduced to instance finding by combinatorial solvers [7, 9, 12]; thus, we use instance finding and element liveness as representatives of such analyses.

The paper is organized as follows. We introduce our running example in Sect. 2. We discuss the challenges of representing the example using either only class modeling or only feature modeling and define a set of design objectives for Clafer in Sect. 3. We then present Clafer in Sect. 4 and demonstrate that it satisfies these objectives. We evaluate the language analytically and experimentally in Sect. 5. We conclude in Sect. 7, after having compared Clafer with related work in Sect. 6.

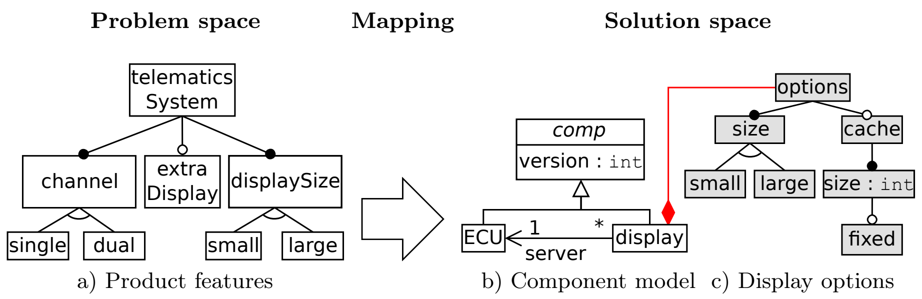


Fig. 1. Telematics product line

2 Running Example: A Telematics Product Line

Vehicle telematics systems integrate multiple telecommunication and information processing functions in an automobile, such as navigation, driving assistance, emergency and warning systems, hands-free phone, and entertainment functions, and present them to the driver and passengers via multimedia displays. Figure 1 presents a variability model of a sample telematics product line, which we will use as a running example. The features offered are summarized in the *problem-space* feature model (Fig. 1a). A concrete telematics system can support either a single or two channels; two channels afford independent programming for the driver and the passengers. The choice is represented as the xor-group channel, marked by the arch between edges. By default, each channel has one associated display; however, we can add one extra display per channel, as indicated by the optional feature extraDisplay. Finally, we can choose large or small displays (displaySize).

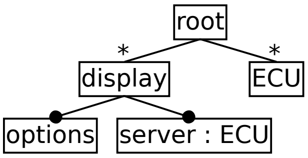
Figure 1b shows a meta-model of components making up a telematics system. There are two types of components: ECUs (electronic control units) and displays. Each display has exactly one ECU as its server. All components have a version.

Components themselves may have options, like the display size or cache (Fig. 1c). We can also specify the cache size and decide whether it is fixed or can be updated dynamically. Thus, the *solution space* model consists of a class model of component types and a feature model of component options.

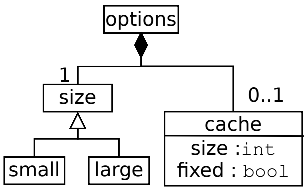
Finally, the variability model maps the problem-space feature configurations to the solution-space component and option configurations. A big arrow in Fig. 1 represents this mapping; we will specify it completely and precisely in Sect. 4.3.

3 Feature vs. Meta-modeling

The solution space in Fig. 1 contains a meta- and a feature model. To capture our intention, the models are connected via UML composition. Since the precise semantics of such notational mixture are not clear, this connection should be understood only informally for now.



a) Cardinality-based feature model of components



b) Meta-model of display options

Fig. 2. Feature model as meta-model and vice versa

We have at least two choices to represent components and options in a single notation. The first is to show the entire solution space model using cardinality-based feature modeling [15]. Figure 2a shows the component part of the model (the subfeatures of options are elided). The model introduces a synthetic root feature; display and ECU can be multiply instantiated; and display has server subfeature representing a reference to instances of ECU. Versions could be added to both display and ECU to match the meta-model in Fig. 1b or we could extend the notation with inheritance. The latter would bring the cardinality-based feature modeling notation very close to meta-modeling based on class modeling, posing the question whether class modeling should not be used for the entire solution space model instead.

We explore the class modeling alternative in Fig. 2b. The figure shows only the options model, as the component model remains unchanged (as in Fig. 1b). Subfeature relationships are represented as UML composition and feature cardinalities correspond to composition cardinalities at the part end. The xor-group is represented by inheritance and cache size and fixed as attributes of cache.

Representing a feature model as a UML class model worked reasonably well for our small example; however, it does have several drawbacks. First, the feature model showed fixed as a property of size by nesting; this intention is lost in the class model. A solution would be to create a separate class size, containing the size value and a class fixed; thus, adding a subfeature to a feature represented as a class attribute requires refactoring. The name of the new class size would clash with the class size representing the display size; thus, we would have to rename one of them, or use nested classes, which further complicates the model. Moreover, converting an xor-group to an or-group in feature modeling is simple: the empty arch needs to be replaced by a filled one. For example, displaySize (Fig. 1a) could be converted to an or-group in a future version of the product line to allow systems with both large and small displays simultaneously. Such change is tricky in UML class models: we would have to either allow one to two objects of type displaySize and write an OCL constraint forbidding two objects of the same subtype (small or large) or use overlapping inheritance (i.e., multiple classification). Thus, the representation of feature models in UML incurs additional complexity.

The examples in Fig. 2 lead us to the following two conclusions:

(1) *"Cardinality-based feature modeling" is a misnomer.* It encompasses multiple instantiation and references, mechanisms characteristic of class modeling, and could even be extended further towards class modeling, e.g., with inheritance; however, the result can hardly be called 'feature modeling', as it clearly goes beyond the original scope of feature modeling [22].
(2) *Existing class modeling notations such as UML and Alloy do not offer first-class support for feature modeling.* Feature models can still be represented in these languages; however, the result carries undesirable notational complexity.

The solution to these two issues is to design a *(class-based) meta-modeling language with first-class support for feature modeling.* We postulate that such a language should satisfy the following design goals:

1. *Provide a concise notation for feature modeling*
2. *Provide a concise notation for meta-modeling*
3. *Allow mixing feature models and meta-models*
4. *Use minimal number of concepts and have uniform semantics*

The last goal expresses our desire that the new language should unify the concepts of feature and class modeling as much as possible, both syntactically and semantically. In other words, we do not want a hybrid language.

4 Clafer: Meta-modeling with First-Class Support for Feature Modeling

We explain the meaning of Clafer models by relating them to their corresponding UML class models.[1] Figure 3 shows the display options feature model in Clafer (a) and the the corresponding UML model (c). Figure 4 shows the component meta-model in Clafer; Fig. 1b has the corresponding UML model.

A Clafer model is a set of type definitions, features, and constraints. A type can be understood as a class or feature type; the distinction is immaterial. Figure 3a contains options as single top-level type definition. The definition contains a hierarchy of features (lines 2-8) and a constraint (lines 10-11); the enclosing type provides a separate name space for this content. The `abstract` modifier prohibits creating an instance of the type, unless extended by a concrete type.

A type definition can contain one or more *features*; the type options has two (direct) features: size (line 2) and cache (line 6). Features are slots that can contain one or more instances or references to instances. Mathematically, features are binary relations. They correspond to attributes or role names of association or composition relationships in UML. For example, in Fig. 4, the feature version (line 2) corresponds to the attribute of the class comp in Fig. 1b; and the feature server (line 6) corresponds to the association role name next to the class ECU in Fig. 1b. Features declared using the arrow notation and having no subfeatures, like in server -> ECU, are *reference features*, i.e., they hold references to instances. Note

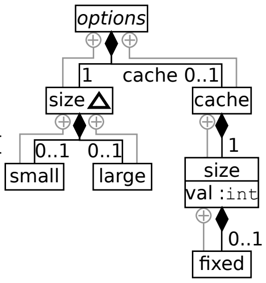
[1] For more precise documentation including meta-models see gsd.uwaterloo.ca/sle2010

```
1  abstract options
2    xor size
3      small
4      large
5
6    cache?
7      size -> int
8        fixed?
9
10   [ small && cache =>
11     fixed ]
```

a) Concise notation

```
1  abstract <0-*> options {
2    <1-1> size 1..1 {
3      <0-*> small 0..1 {}
4      <0-*> large 0..1 {}
5    }
6    <0-*> cache 0..1 {
7      <0-*> size -> int 1..1 {
8        <0-*> fixed 0..1 {}
9      }
10   }
11   [ some this.size.small &&
12     some this.cache =>
13     some this.cache.size.fixed ]
14 }
```

b) Full notation



c) UML class model

Fig. 3. Feature model in Clafer and corresponding UML class model

that we model integral features, like version (line 2) in Fig. 4, as references. Clafer has only one object representing a given number, which speeds up automated analyses.

Features that do not have their type declared using the arrow notation, such as size (line 2) and cache in Fig. 3a, or have subfeatures, such as size (line 7) in Fig. 3a, are *containment features*, i.e., features that contain instances. An instance can be contained by only one feature, and no cycles in instance containment are allowed. These features correspond to role names at the part end of composition relationships in UML. For example, the feature cache in Fig. 3a corresponds to the role name cache next to the class cache in Fig. 3c. By a UML convention, the role name at the association or composition end touching a class is, if not specified, same as the class name.

A containment feature definition creates a feature and, implicitly, a new concrete type, both located in the same name space. For example, the feature definition cache (line 6) in Fig. 3a defines both the feature cache, corresponding to the role name in Fig. 3c, and, implicitly, the type cache, corresponding to the class cache in Fig. 3c. The new type is nested in the type options; in UML this nesting means that the class cache is an inner class of the class options, i.e., its full name is options::cache. Figure 3c shows UML class nesting relations in light color. Class nesting permits two classes named size in a single model, because each enclosing class defines an independent name scope.

```
1  abstract comp
2    version -> int
3
4  abstract ECU extends comp
5  abstract display extends comp
6    server -> ECU
7    `options
8    [ version >= server.version ]
9
```

Fig. 4. Class model in Clafer

The feature size (line 7) in Fig. 3a is a containment feature of general form: the implicitly defined type is a structure containing a reference, here to `int`, and a subfeature, fixed. This type corresponds to the class cache::size in Fig. 2b.

Features have *feature cardinalities*, which constrain the number of instances or references that a given feature can contain. Cardinality of a feature is specified by an interval $m..n$, where $m \in \mathbb{N}, n \in \mathbb{N} \cup \{*\}, m \leq n$. Feature cardinality specification follows the feature name or its reference type, if any.

Conciseness is an important goal for Clafer; therefore, we provide syntactic sugar for common constructions. Figures 3a and 3b show the same Clafer model; the first one is written in concise notation, while the second one is completely desugared code with resolved names in constraints.

Clafer provides syntactic sugar similar to syntax of regular expressions: ? or `lone` (optional) denote 0..1; * or `any` denote 0..$*$; and + or `some` denote 1..$*$. For example, cache (line 6) in Fig. 3 is an optional feature. No feature cardinality specified denotes 1..1 (mandatory) by default, modulo four exceptions explained shortly. For example, size (line 7) in Fig. 3a is mandatory.

Features and types have *group cardinalities*, which constrain the number of child instances, i.e., the instances contained by subfeatures. Group cardinality is specified by an interval $\langle m{-}n \rangle$, with the same restrictions on m and n as for feature cardinalities, or by a keyword: `xor` denotes $\langle 1{-}1 \rangle$; `or` denotes $\langle 1{-}* \rangle$; `opt` denotes $\langle 0{-}* \rangle$; and `mux` denotes $\langle 0{-}1 \rangle$; further, each of the three keywords makes subfeatures optional by default. If any, a group cardinality specification precedes a feature or type name. For example, `xor` on size (line 2) in Fig. 3a states that only one child instance of either small or large is allowed. Because the two subfeatures small and large have no explicit cardinality attached to them, they are both optional (cf. Fig. 3b). No explicit group cardinality stands for $\langle 0{-}* \rangle$, except when it is inherited as illustrated later.

Constraints are a significant aspect of Clafer. They can express dependencies among features or restrict string or integer values. Constraints are always surrounded by square brackets and are a conjunction of first-order logic expressions. We modeled constraints after Alloy; the Alloy constraint notation is elegant, concise, and expressive enough to restrict both feature and class models. Logical expressions are composed of terms and logical operators. Terms either relate values (integers, strings) or are navigational expressions. The value of navigational expression is always a relation, therefore each expression must be preceded by a *quantifier*, such as `no`, `one`, `lone` or `some`. However, lack of explicit quantifier (Fig. 3a) stands for `some` (Fig. 3b), signifying that the relation cannot be empty.

Each feature in Clafer introduces a local namespace, which is rather different from namespaces in popular programming languages. Name resolution is important in two cases: 1) resolving type names used in feature and type definitions and 2) resolving feature names used in constraints. In both cases, names are path expressions, used for navigation like in OCL or Alloy, where the dot operator joins two relations. A name is resolved in a context of a feature in up to four steps. First, it is checked to be a special name like `this`. Secondly, the name is looked up in subfeatures in breadth-first search manner. If it is still not found, the algorithm searches in the top-level definition that contains the feature in its hierarchy. Otherwise, it searches in other top-level definitions. If the name cannot be resolved or is ambiguous within a single step, an error is reported.

Clafer supports single inheritance. In Fig. 4, the type ECU inherits features and group cardinality of its supertype. The type display extends comp by adding two features and a constraint. The reference feature server points to an existing ECU instance. The meaning of ‘options notation is explained in Sect. 4.1.

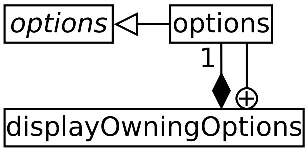
The constraint defined in the context of display states that display's version cannot be lower than server's version. To dereference the server feature, we use dot, which then returns version.

4.1 Mixing via Quotes and References

Mixing class and feature models in Clafer is achieved via *quotation* (see line 7 in Fig. 4) or references. Syntactically, quotation is just a name of abstract type preceded by left quote (‘), which in the example is expanded as options extends options. The first name indicates a new feature, and the second refers to the abstract type. Semantically, this notation creates a containment feature options with a new concrete type display.options, which extends the top-level abstract type options from Fig. 3a. The concrete type inherits group cardinality and features of its supertype. By using quotation only the quoted type is shared, but no instances. References, on the other hand, are used for sharing instances.

The following example highlights the difference:

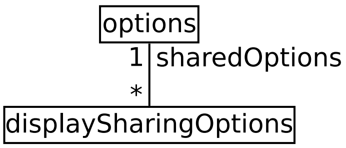
```
abstract options
  -- content as in options in Fig. 3a

displayOwningOptions *
  ‘options -- shorthand for options extends options
```



In the above snippet, each instance of displayOwningOptions will have its own instance of type options, as depicted in the corresponding UML diagram. Other types could also quote options to reuse it. Note that Clafer assumes the existence of an implicit *root object*; thus, a feature definition, such as displayOwningOptions above, defines both a subfeature of the root object and a new top-level concrete type.

Now consider the following code with corresponding UML diagram:

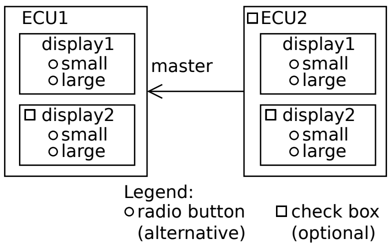
```
options *
  -- content as in options in Fig. 3a

displaySharingOptions *
  sharedOptions -> options
```



Each instance of displaySharingOptions has a reference named sharedOptions pointing to an instance of options. Although there can be many references, they might all point to the same instance living somewhere outside displaySharingOptions.

```
1 abstract plaECU extends ECU
2   ‘display 1..2
3     [ ~cache
4       server = parent ]
5 ECU1 extends plaECU
6 ECU2 extends plaECU ?
7   master -> ECU1
```

a) Clafer model



b) A possible graphical rendering

Fig. 5. Architectural template

4.2 Specializing via Inheritance and Constraints

Let us go back to our telematics product line example. The architectural meta-model as presented in Fig. 4 is very generic: the meta-model describes infinitely many different products, each corresponding to its particular instance. We would like to *specialize* and *extend* the meta-model to create a particular *template*. A template makes most of the architectural structure fixed, but leaves some points of variability. In previous work, we introduced *feature-based model templates* (FBMT in short) as models (instances of meta-models) with optional elements annotated with Boolean expressions over features known as *presence conditions* [13]. Below, we show how such templates can be expressed in Clafer.

Figure 5a shows such a template for our example. We achieve specialization via inheritance and constraints. In particular, we represent instances of meta-model classes as singleton classes. In our example, a concrete product must have at least one ECU and thus we create ECU1 to represent the mandatory instance. Then, optional instances are represented using classes with cardinality 0..1. Our product line can optionally have another ECU, represented by ECU2. Similarly, each ECU has either one display or two displays, but none of the displays has cache. Besides, we need to constrain the server reference in each display in plaECU, so that it points to its associated ECU. The constraint in line 3 in Fig. 5a is nested under display. The reference parent points to the current instance of plaECU, which is either ECU1 or ECU2. Also, ECU2 *extends* the base type with master, pointing to ECU1 as the main control unit.

Figure 5b visualizes the template in a domain-specific notation, showing both the fixed parts, e.g., mandatory ECU1 and display1, and the variable parts, e.g., alternative display sizes (radio buttons) and optional ECU2 and display2 (checkboxes). Note that model templates such as UML models annotated with presence conditions (e.g., [13]) can be translated into Clafer automatically by 1) representing each model element e by a class with cardinality 0..1 that extends the element's meta-class and 2) a constraint of the form p && c <=> e, with p being e's parent and c being e's presence condition. In our example, we keep these constraints separate from the template (see Sect. 4.3). Further, in contrast to annotating models with presence conditions, we can use subclassing and constraints to specialize and extend the meta-model in multiple layers.

```
telematicsSystem
  xor channel
    single
    dual
  extraDisplay?
  xor displaySize
    small
    large
```

```
[ dual <=> ECU2
  extraDisplay <=> #ECU1.display = 2
  extraDisplay <=>
      (ECU2 <=> #ECU2.display = 2)
  small <=> ~plaECU.display.options.size.large
  large <=> ~plaECU.display.options.size.small
]
```

Fig. 6. Feature model with mapping constraints

```
-- concrete product
[ dual && extraDisplay && telematicsSystem.size.large && comp.version == 1 ]
```

Fig. 7. Constraint specifying a single product

4.3 Coupling via Constraints

Having defined the architectural template, we are ready to expose the remaining variability points as a product-line feature model. Figure 6 shows this model (cf. Fig. 1a) along with a set of constraints coupling its features to the variability points of the template. Note that the template allowed the number of displays (`ECU1.display` and `ECU2.display`) and the size of every display to vary independently; however, we further restrict the variability in the feature model, requiring either all present ECUs to have two displays or all to have no extra display and either all present displays to be small or all to be large. Also note that we opted to explain the meaning of each feature in terms of the model elements to be selected rather than defining the presence condition of each element in terms of the features. Both approaches are available in Clafer, however.

Constraints allow us restricting a model to a single instance. Figure 7 shows a top-level constraint specifying a single product, with two ECUs, two large displays per ECU, and all components in version 1. Based on this constraint, we can automatically instantiate the product line using the Alloy analyzer, as described in Sect. 5.2.

5 Evaluation

5.1 Analytical Evaluation

We now discuss to what extent Clafer meets its design goals from Sect. 3.

(1) *Clafer provides a concise notation for feature modeling.* This can be seen by comparing Clafer to TVL, a state-of-the-art textual feature modeling language [8]. Feature models in Clafer look very similar to feature models in TVL, except that TVL uses explicit keywords (e.g., to declare groups) and braces for nesting. Figure 8a shows the TVL encoding of the feature model from Fig. 3.

```
Options group allof {
  Size group oneof { Small, Large },
  opt Cache group allof {
    CacheSize group allof {
      SizeVal { int val; },
      opt Fixed
    }
  },
  Constraint { (Small && Cache) -> Fixed; }
}
```

a) Options feature model in TVL

```
class Comp {
  reference version : Integer
}

class ECU extends Comp{ }

class Display extends Comp {
  reference server : ECU
  attribute options : Options
}
```

b) Component meta-model in KM3

Fig. 8. Our running example in TVL and KM3

Clafer's language design reveals four key ingredients allowing a class modeling language to provide a concise notation for feature modeling:

– *Containment features*: A containment feature definition creates both a feature (a slot) and a type (the type of the slot); for example, all features in Figs. 3 and 6 are of this kind. Neither UML nor Alloy provide this mechanism; in there, a slot and the class used as its type are declared separately.
– *Feature nesting*: Feature nesting accomplishes instance composition and type nesting in a single construct. UML provides composition, but type nesting is specified separately (cf. Fig. 3c). Alloy has no built-in support for composition and thus requires explicit parent-child constraints. It also has no signature nesting, so name clashes need to be avoided using prefixes or alike.
– *Group constraints*: Clafer's group constraints are expressed concisely as intervals. In UML groups can be specified in OCL, but using a lengthy encoding, explicitly listing features belonging to the group. Same applies to Alloy.
– *Constraints with default quantifiers*: Default quantifiers on relations, such as `some` in Fig. 3, allow writing constraints that look like propositional logic, even though their underlying semantics is first-order predicate logic.

(2) *Clafer provides a concise notation for meta-modeling.* Figure 8b shows the meta-model of Fig. 4 encoded in KM3 [21], a state-of-the-art textual meta-modeling language. The most visible syntactic difference between KM3 and Clafer is the use of explicit keywords introducing elements and mandatory braces establishing hierarchy. KM3 cannot express additional constraints in the model. They are specified separately, e.g. as OCL invariants.

It is instructive to compare the size of the Clafer and Alloy models of the running example. With similar code formatting (no comments and blank lines), Clafer representation has 43 LOC and the automatically generated Alloy code is over two times longer. Since the Alloy model contains many long lines, let us also compare source file sizes: 1kb for Clafer and over 4kb for Alloy. The code generator favors conciseness of the translation over uniformity of the generated code. Still, in the worst case, the lack of the previously listed constructs makes Alloy models necessarily larger. Other language differences tip the balance further in favor of Clafer. For example, an abstract type definition in Clafer guarantees that the type will not be automatically instantiated; however, unextended abstract sets can be still instantiated by Alloy Analyzer. Therefore, each abstract signature in Alloy needs to be extended by an additional signature.

(3) *Clafer allows mixing feature and meta-models.* Quotations allow reusing feature or class types in multiple locations; references allow reusing both types and instances. Feature and class models can be related via constraints (Fig. 6).

(4) *Clafer tries to use a minimal number of concepts and has uniform semantics.* While integrating feature modeling into meta-modeling, our goal was to avoid creating a hybrid language with duplicate concepts. In Clafer, there is no distinction between class and feature types. Features are relations and, besides their obvious role in feature modeling, they also play the role of attributes in meta-modeling. We also contribute a simplification to feature modeling: Clafer has no explicit feature group construct; instead, every feature can use a group cardinality to constrain the number of children. This is a significant simplification, as we no longer need to distinguish between "grouping features" (features used purely for grouping, such as menus) and feature groups. The grouping intention and grouping cardinalities are orthogonal: any feature can be annotated as a grouping feature and any feature may chose to impose grouping constraints on children. Finally, both feature and class modeling have a uniform semantics: a Clafer model instance, just like Alloy's, is a set of relations.

5.2 Experimental Evaluation

Our experiment aims to show that Clafer can express a variety of realistic feature models, meta-models and model templates and that useful analyses can be performed on these encodings in reasonable time. Then it follows that the richness of Clafer's applications, does not come at a cost of lost analysis potential with respect to models in more specialized languages.

The experiment methodology is summarized in the following steps:

1. *Identify a set of models representative for the three main use cases of Clafer: feature modeling, meta-modeling, and mixed feature and meta-modeling.*
2. *Select representative analyses.* We studied the analyses in published literature and decided to focus on a popular class of analyses, which reduce to model instance finding. These include inconsistency detection, element liveness analysis, offline and interactive configuration, guided editing, etc. Since all these have similar performance characteristics, we decided to use model instance finding, consistency and element liveness analysis as representative.
3. *Translate models into Clafer and record observations.* We created automatic translators for converting models to Clafer if it was enough to apply simple rewriting rules. In other cases, translation was done manually.
4. *Run the analyses and reporting performance results.* The analyses are implemented by using our Clafer-to-Alloy translator, and then employing Alloy Analyzer (which is an instance finder) to perform the analysis.

The Clafer-to-Alloy translator is written in Haskell and comprises several chained modules: lexer, layout resolver, parser, desugarer, semantic analyzer, and code generator. Layout resolver makes braces grouping subfeatures optional. Clafer is composed of two languages: the core and the full language. The first one is a minimal language with well-defined semantics. The latter is built on top of the core language and provides large amount of syntactic sugar (cf. Fig. 3). Semantic analyzer resolves names and deals with inheritance. The code generator translates the core language into Alloy. The generator has benefited from the knowledge about the class of models it is working with to optimize the translation, in the same way as analyzers for specialized languages have this knowledge.

The experiment was executed on a laptop with a Core Duo 2 @2.4GHz processor and 2.5GB of RAM, running Linux. Alloy Analyzer was configured to use Minisat as a solver. All Clafer and generated Alloy models are available at gsd.uwaterloo.ca/sle2010. In the subsequent paragraphs we present and discuss the results for the three subclasses of models.

Feature Models. In order to find representative models we have consulted SPLOT [27] — a popular repository of feature models. We have succeeded in automatically translating all 58 models from SPLOT to Clafer (non-generated, human-made models; available as of July 4th, 2010). These include models with and without cross-tree constraints, ranging from a dozen to hundreds of features.

Results for all models are available online at the above link. Here, we report the most interesting cases together with further four, which have been randomly generated; all listed in Table 1. Digital Video Systems is a small example with few cross-tree constraints. Dell Laptops models a set of laptops offered by Dell in 2009. This is one of few models that contains more constraints than features. Arcade Game describes a product line of computer games; it contains tens of features and constraints. EShop [25] is the largest realistic model that we have found on SPLOT. It is a domain model of online stores. The remaining models are randomly generated using SPLOT, with a fixed 10% constraint/variable ratio.

We checked consistency of each model by instance finding. Table 1 presents summary of results. The analysis time was less then a second for up to several hundred features and less than a minute for up to several thousand features. Interestingly, the biggest bottleneck was the Alloy Analyzer itself (which translates Alloy into a CNF formula)—reasoning about the CNF formula in a SAT-solver takes no more than hundreds of milliseconds.

Meta-Models. In order to identify representative meta-models, we have turned to the Ecore Meta-model Zoo (www.emn.fr/z-info/atlanmod/index.php/Ecore), from where we have selected the following meta-models: AWK Programs, ATL, ANT, BibTex, UML2, ranging from tens to hundreds of elements. We translated all these into Clafer automatically. One interesting mapping is the translation of EReference elements with eOpposite attribute (symmetric reference), as there is no first-class support for symmetric references in Clafer. We modeled them as constraints relating references with their symmetric counterparts. Moreover we have not handled multiple inheritance in our translation.

Since none of these meta-models contained OCL constraints, we extracted OCL constraints from the UML specification [29] and manually added them to the Clafer encoding of UML2. We did observe certain patterns during that translation and believe that this task can be automated for a large class of constraints. Table 2 presents sample OCL constraints translated into Clafer. Each constraint, but last, is written in a context of some class. Their intuitive meanings are as follows: 1) ownedReception is empty if there is no isActive; 2) endType aggregates all types of memberEnds; 3) if memberEnd's aggregation is different from none then there are two instances of memberEnd; 4) there are no two types of the same names. All Clafer encodings of the meta-models are available at the above link.

There are several reasons why Clafer constraints are more concise and uniform compared with OCL invariants. Similarly to Alloy, every Clafer definition is a relation. This approach, eliminates extra constructions such as OCL's collect, allInstances. Finally, assuming the default some quantifier before relational operations (e.g. memberEnd.aggregation - none), we can treat result of an operation as if it was a propositional formula, thus eliminating extra exists quantifiers.

We applied automated analyses to slices of the UML2 meta-model: Class Diagram from [10], State Machines, and Behaviors (Table 3). Each slice has tens of classes and our goal was to include a wide range of OCL constraints. We

Table 1. Results of consistency analysis for *feature models* expressed in Clafer

model name	nature	size [# features]	[# constraints]	running time [s]
Digital Video System	Realistic	26	3	0.012
Dell Laptops	Realistic	46	110	0.025
Arcade Game	Realistic	61	34	0.040
eShop	Realistic	287	21	0.15
FM-500-50-1	Generated	500	50	0.45
FM-1000-100-2	Generated	1000	100	1.5
FM-2000-200-3	Generated	2000	200	4.5
FM-5000-500-4	Generated	5000	500	28.0

Table 2. Constraints in OCL and Clafer

Context	OCL	Clafer
Class	(not self.isActive) implies self.ownedReception->isEmpty()	~isActive => no ownedReception
Association	self.endType = self.memberEnd-> collect(e | e.type)	endType = memberEnd.type
Association	self.memberEnd->exists(aggregation <> Aggregation::none) implies self.memberEnd->size() = 2	memberEnd.aggregation - none => #memberEnd = 2
–	Type.allInstances() -> forAll (t1, t2 | t1 <> t2 implies t1.name <> t2.name)	all disj t1, t2 : Type | t1.name != t2.name

checked the *strong consistency* property [9] for these meta-models. To verify this property, we instantiated meta-models' elements that were at the bottom of inheritance hierarchy, by restricting their cardinality to be at least one. The same constraints were imposed on containment references within all meta-model elements. The analysis confirmed that none of the meta-models had dead elements. Our results show that liveness analysis can be done efficiently for realistic meta-models of moderate size.

Feature-Based Model Templates. The last class of models are feature-based model templates akin to our telematics example. A FBMT consists of a feature model (cf. Fig. 6, left), a meta-model (cf. Fig. 4), a template (cf. Fig. 5a), and a set of mapping constraints (cf. Fig. 6, right). To the best of our knowledge, Electronic Shopping [25] is the largest example of a model template found in the literature. We used its templates, listed in Table 4, for evaluation: FindProduct and Checkout are activity diagram templates, and TaxRule is a class diagram template. Each template had substantial variability in it. All templates have between 10 and 20 features, tens of classes and from tens to hundreds constraints. For comparison, we also include our telematics example.

We manually encoded the above FBMTs in Clafer. For each of the diagrams in [25], we took a slice of UML2 meta-model and created a template that conforms to the meta-model, using mandatory and optional singleton classes as described in Sect. 4.2. To create useful and simple slices of UML diagrams, we removed unused attributes and flattened inheritance hierarchy, since many superclasses were left without any attributes. Thus, the slice preserved the core semantics. Furthermore, we sliced the full feature model, so that it contains only features that appear in diagram. Finally, we added mappings to express dependencies between features and model elements, as described in Sect. 4.3.

Table 3. Results of strong consistency analysis for UML2 *meta-model* slices in Clafer

meta-model/instance size	[#classes]	[#constraints]	running time [s]
State Machines	11	28	0.08
Class Diagram	19	17	0.15
Behaviors	20	13	0.23

Table 4. Analyses for *Feature-Based Model Templates* expressed in Clafer. Parentheses by the model names indicate the number of optional elements in each template.

FBMT	#features/#classes/#constraints	instantiation [s]	element liveness [s]
Telematics (8)	8/7/17	0.04	0.26
FindProduct (16)	13/29/10	0.07	0.18
TaxRules (7)	16/24/62	0.11	0.12
Checkout (41)	18/78/314	1.6	5.8

We performed two types of analyses on FBMTs. First, we created sample feature configurations (like in Fig. 7) and instantiated templates in the Alloy Analyzer. We inspected each instance and verified that it was the expected one.

Second, we performed element liveness analysis for the templates. The analysis is similar to element liveness for meta-models [9], but now applied to template elements. We performed the analysis by repeated instance finding; in each iteration we required the presence of groups of non-exclusive model elements.

Table 4 presents summary of inspected models and times of analyses. Often the time of liveness analysis is very close to the time of instantiation multiplied by the number of element groups. For instance, for FindProduct, liveness analysis was three times longer than time of instantiation, because elements were arranged into 3 groups of non-conflicting elements. This rule holds when the Alloy Analyzer uses the same scope for element instances.

We consider our results promising, since we obtained acceptable timings for slices of realistic models, without fully exploiting the potential of Alloy. The results can clearly be further improved by better encoding of slices (for example, representing activity diagram edges as relations instead of sets in Alloy) and using more intelligent slicing methods; e.g. some constraints are redundant, such as setting source and target edges in ActivityNodes, so removing these constraints would speed up reasoning process. However already now we can see that Clafer is a suitable vehicle for specifying FBMTs and analyzing them automatically.

Threats to Validity

External Validity. Our evaluation is based on the assumption that we chose representative models and useful and representative analyses.

All models, except the four randomly generated feature models, were created by humans to model real-word artifacts. As all, except UML2, come from academia, there is no guarantee that they share characteristics with industrial models. Majority of practical models have less than a thousand features [24], so reasoning about corresponding Clafer models is feasible and efficient. Perhaps the biggest real-world feature model up to date is the Linux Kernel model (almost 5500 features and thousands of constraints) [31]. It would presently pose a challenge for our tools. Working with models of this size requires proper engineering of analyses. Our objective here was to demonstrate feasibility of analyses. We will continue to work on robust tools for Clafer in future.

We believe that the slices of UML2 selected for the experiment are representative of the entire meta-model because we picked the parts with more complex constraints. While there are not many existing FBMTs to choose from, the e-commerce example [25] was reversed engineered from the documentation of an IBM e-commerce platform, which makes the model quite realistic.

Not all model analyses can be reduced to instance finding performed using combinatorial solvers (relational model finder in case of Alloy [34]). However combinatorial analyses belong to most widely recognized and effective [7].

Instance finding for models has similar uses to testing and debugging for programs [19]—it helps to uncover flaws in models, assists in evolution and configuration. For example it helped us discover that our original Clafer code was missing constraints (lines 9–10 and 14–15 in Fig. 5a and line 14 in Fig. 6). Some software platforms already provide configuration tools using reasoners; for example, Eclipse uses a SAT solver to help users select valid sets of plug-ins [26].

Liveness analysis for model elements has been previously exploited, for instance in [33, 9]. Tartler et al. [33] analyze liveness of features in the Linux kernel code, reporting about 60 previously unreported dead features in the released kernel versions. Linux is not strictly a feature-based model template, but its build architecture, which relies on (a form of) feature models and presence conditions on code (conditional compilation) highly resembles our model templates.

Analyzers based on instance finding solve an NP-hard problem. Thus no hard guarantees can be given for their running times. Although progress in solver technologies has placed these problems in the range of practically tractable, there do exist instances of models and meta-models, which will effectively break the performance of our tools. Our experiments aim at showing that this does not happen for realistic models.

There exist more sophisticated analyzes (and classes of models) that cannot be addressed with Clafer infrastructure, and are not reflected in our experiment. For example instance finding is limited to instances of bounded size. It is possible to build sophisticated meta-models that only have very large instances. This problem is irrelevant for feature models and model templates as they allow no no classes that can be instantiated without bounds.

Moreover special purpose languages may require more sophisticated analyses techniques such as behavioral refinement checking, model checking, model equivalence checking, etc. These properties typically go beyond static semantics expressed in meta-models and thus are out of scope for generic Clafer tools.

Internal Validity. Translating models from one language to another can introduce errors and change semantics of the resulting model.

We used our own tools to convert SPLOT and Ecore models to Clafer and then to translate Clafer to Alloy. We translated FBMTs and OCL constraints manually. The former is rather straightforward; the latter is more involved. We publish all the models so that their correctness can be reviewed independently.

Another threat to correctness is the slice extraction for UML2 and e-commerce models. Meta-model slicing is a common technique used to speed-up model analyses, where reasoner processes only relevant parts of the meta-model. We performed it manually, while making sure that all parts relevant to the selected constraints were included; however, the technique can be automated [30].

The correctness of the analyses relies on the correctness of the Clafer-to-Alloy translator and the Alloy analyzer. The Alloy analyzer is a mature piece of software. We tested Clafer-to-Alloy translator by translating sample models to Alloy and inspecting the results.

6 Related Work

We have already mentioned related work on model analysis; here we focus on work related to our main contribution, Clafer's novel language design.

Asikainen and Männistö present Forfamel, a unified conceptual foundation for feature modeling [4]. The basic concepts of Forfamel and Clafer are similar; both include subfeature, attribute, and subtype relations. The main difference is that Clafer's focus is to provide concise concrete syntax, such as being able to define feature, feature type, and nesting by stating an indented feature name. Also, the conceptual foundations of Forfamel and Clafer differ; e.g., features in Forfamel correspond to Clafer's instances, but features in Clafer are relations. Also, a feature instance in Forfamel can have several parents; in Clafer, an instance has at most one parent. These differences likely stem from the difference in perspective: Forfamel takes a feature modeling perspective and aims at providing a foundation unifying the many existing extensions to feature modeling; Clafer limits feature modeling to its original FODA scope [22], but integrates it into class modeling. Finally, Forfamel considers a constraint language as out of scope, hinting at OCL. Clafer comes with a concise constraint notation.

TVL is a textual feature modeling language [8]. It favors the use of explicit keywords, which some software developers may prefer. The language covers Boolean features and features of other primitive types such as integer. The key difference is that Clafer is also a class modeling language with multiple instantiation, references, and inheritance. It would be interesting to provide a translation from TVL to Clafer. The opposite translation is only partially possible.

As mentioned earlier, class-based meta-modeling languages, such as KM3 [21] and MOF [28] cannot express feature models as concisely as Clafer.

Nivel is a meta-modeling language, which was applied to define feature and class modeling languages [3]. It supports deep instantiation, enabling concise definitions of languages with class-like instantiation semantics. Clafer's purpose is different: to provide a concise notation for combining feature and class models within a single model. Nivel could be used to define the abstract syntax of Clafer, but it would not be able to naturally support our concise concrete syntax.

Clafer builds on our several previous works, including encoding feature models as UML class models with OCL [16]; a Clafer-like graphical profile for Ecore, having a bidirectional translation between an annotated Ecore model and its rendering in the graphical syntax [32]; and the Clafer-like notation used to specify framework-specific modeling languages [2]. None of these works provided a proper language definition and implementation like Clafer; also, they lacked Clafer's concise constraint notation.

Gheyi et al. [17] pioneered translating Boolean feature models into Alloy. Anastasakis et al. [1] automatically translated UML class diagrams with OCL constraints to Alloy. Clafer covers both types of models.

Relating problem-space feature models and solution-space models has a long tradition. For example, feature models have been used to configure model templates before [13, 18]. That work considered model templates as superimposed instances of a metamodel and presence conditions attached to individual elements of the instances; however, the solution in Sect. 4.2 implements model templates as specializations of a metamodel. Such a solution allows us treating the feature model, the metamodel, and the template at the same metalevel, simply as parts of a single Clafer model. This design allows us to elegantly reuse a single constraint language at all these levels. As another example, Janota and Botterweck show how to relate feature and architectural models using constraints [20]. Again, our work differs from this work in that our goal is to provide such integration within a single language. Such integration is given in Kumbang [5], which is a language that supports both feature and architectural models, related via constraints. Kumbang models are translated to Weight Constraint Rule Language (WCRL), which has a reasoner supporting model analysis and instantiation. Kumbang provides a rich domain-specific vocabulary, including features, components, interfaces, and ports; however, Clafer's goal is a minimal clean language covering both feature and class modeling, and serving as a platform to derive such domain specific languages, as needed.

7 Conclusion

The premise for our work are usage scenarios mixing feature and class models together, such as representing components as classes and their configuration options as feature hierarchies and relating feature models and component models using constraints. Representing both types of models in single languages allows us to use a common infrastructure for model analysis and instantiation.

We set off to integrate feature modeling into class modeling, rather than trying to extend feature modeling as previously done [15]. We propose the concept of a class modeling language with first-class support for feature modeling and define a set of design goals for such languages. Clafer is an example of such a language, and we demonstrate that it satisfies these goals. The design of Clafer revealed that a class modeling language can provide a concise notation for feature modeling if it supports containment feature definitions, feature nesting, group cardinalities, and constraints with default quantifiers. Our design contributes a precise characterization of the relationship between feature and class modeling and a uniform framework to reason about both feature and class models.

References

1. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. Software and Systems Modeling 9(1) (2008)
2. Antkiewicz, M., Czarnecki, K., Stephan, M.: Engineering of framework-specific modeling languages. IEEE TSE 35(6) (2009)
3. Asikainen, T., Männistö, T.: Nivel: a metamodelling language with a formal semantics. Software and Systems Modeling 8(4) (2009)
4. Asikainen, T., Männistö, T., Soininen, T.: A unified conceptual foundation for feature modelling. In: SPLC 2006 (2006)
5. Asikainen, T., Männistö, T., Soininen, T.: Kumbang: A domain ontology for modelling variability in software product families. Adv. Eng. Inform. 21(1) (2007)

6. Bart Veer, J.D.: The eCos Component Writer's Guide (2000)
7. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated analysis of feature models 20 years later: a literature review. Information Systems 35(6) (2010)
8. Boucher, Q., Classen, A., Faber, P., Heymans, P.: Introducing TVL, a text-based feature modelling language. In: VaMoS 2010 (2010)
9. Cabot, J., Clarisó, R., Riera, D.: Verification of UML/OCL Class Diagrams Using Constraint Programming. In: MoDeVVA 2008 (2008)
10. Cariou, E., Belloir, N., Barbier, F., Djemam, N.: Ocl contracts for the verification of model transformations. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795. Springer, Heidelberg (2009)
11. Clauß, M., Jena, I.: Modeling variability with UML. In: Dannenberg, R.B. (ed.) GCSE 2001. LNCS, vol. 2186. Springer, Heidelberg (2001)
12. Czarnecki, K., Pietroszek, K.: Verifying feature-based model templates against well-formedness ocl constraints. In: GPCE 2006 (2006)
13. Czarnecki, K., Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 422–437. Springer, Heidelberg (2005)
14. Czarnecki, K., Bednasch, T., Unger, P., Eisenecker, U.: Generative programming for embedded software: An industrial experience report. In: Batory, D., Blum, A., Taha, W. (eds.) GPCE 2002. LNCS, vol. 2487. Springer, Heidelberg (2002)
15. Czarnecki, K., Helsen, S., Eisenecker, U.: Formalizing cardinality-based feature models and their specialization. SPIP 10(1) (2005)
16. Czarnecki, K., Kim, C.H.: Cardinality-based feature modeling and constraints: A progress report. In: OOPSLA 2005 Workshop on Software Factories (2005)
17. Gheyi, R., Massoni, T., Borba, P.: A theory for feature models in Alloy. In: First Alloy Workshop (2006)
18. Heidenreich, F., Kopcsek, J., Wende, C.: FeatureMapper: Mapping Features to Models. In: ICSE 2008 (2008)
19. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press, Cambridge (2006)
20. Janota, M., Botterweck, G.: Formal approach to integrating feature and architecture models. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 31–45. Springer, Heidelberg (2008)
21. Jouault, F., Bézivin, J.: KM3: a DSL for Metamodel Specification. In: IFIP 2006 (2006)
22. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-21, CMU (1990)
23. Kang, K.C.: FODA: Twenty years of perspective on feature modeling. In: VaMoS 2010 (2010)
24. Kästner, C.: Virtual Separation of Concerns: Toward Preprocessors 2.0. Ph.D. thesis, University of Magdeburg (2010)
25. Lau, S.Q.: Domain Analysis of E-Commerce Systems Using Feature-Based Model Templates. Master's thesis, University of Waterloo (2006)
26. Le Berre, D., Rapicault, P.: Dependency management for the Eclipse ecosystem: Eclipse p2, metadata and resolution. In: IWOCE 2009 (2009)
27. Mendonça, M., Branco, M., Cowan, D.: S.P.L.O.T. - Software Product Lines Online Tools. In: OOPSLA 2009 (2009)
28. OMG: Meta Object Facility (MOF) Core Specification (2006)
29. OMG: OMG Unified Modeling Language (2009)
30. Shaikh, A., Clarisó, R., Wiil, U.K., Memon, N.: Verification-Driven Slicing of UML/OCL Models. In: ASE 2010 (2010)

31. She, S., Lotufo, R., Berger, T., Wasowski, A., Czarnecki, K.: Variability model of the linux kernel. In: VaMoS 2010 (2010)
32. Stephan, M., Antkiewicz, M.: Ecore.fmp: A tool for editing and instantiating class models as feature models. Tech. Rep. 2008-08, Univeristy of Waterloo (2008)
33. Tartler, R., Sincero, J., Lohmann, D.: Dead or Alive: Finding Zombie Features in the Linux Kernel. In: FOSD 2009 (2009)
34. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007)

Support for the Evolution of C++ Generic Functions

Peter Pirkelbauer[1], Damian Dechev[2], and Bjarne Stroustrup[3]

[1] Lawrence Livermore National Laboratory
Center for Applied Scientific Computing
Livermore, CA 94550-9698
`peter.pirkelbauer@llnl.gov`
[2] University of Central Florida
Department of Electrical Engineering and Computer Science
Orlando, FL 32816-2362
`dechev@eecs.ucf.edu`
[3] Texas A&M University
Department of Computer Science and Engineering
College Station, TX 77843-3112
`bs@cse.tamu.edu`

Abstract. The choice of requirements for an argument of a generic type or algorithm is a central design issue in generic programming. In the context of C++, a specification of requirements for a template argument or a set of template arguments is called a *concept*.

In this paper, we present a novel tool, TACE (template analysis and concept extraction), designed to help programmers understand the requirements that their code de facto imposes on arguments and help simplify and generalize those through comparisons with libraries of well-defined and precisely-specified concepts. TACE automatically extracts requirements from the body of function templates. These requirements are expressed using the notation and semantics developed by the ISO C++ standards committee. TACE converts implied requirements into concept definitions and compares them against concepts from a repository. Components of a well-defined library exhibit commonalities that allow us to detect problems by comparing requirements from many components: Design and implementation problems manifest themselves as minor variations in requirements. TACE points to source code that cannot be constrained by concepts and to code where small modifications would allow the use of less constraining concepts. For people who use a version of C++ with concept support, TACE can serve as a core engine for automated source code rejuvenation.

1 Introduction

A fundamental idea of generic programming is the application of mathematical principles to the specification of software abstractions [22]. ISO C++ [15][24] supports generic programming through the use of templates. Unfortunately, it does not directly support the specification of requirements for arguments to generic types and functions [23]. However, research into language-level support for specifying such requirements, known as *concepts*, for C++ has progressed to the point where their impact on software can be examined [13][7][14]. Our work is aimed at helping programmers cope with the current lack of direct support for concepts and ease the future transition to language-supported concepts.

Templates are a compile-time mechanism to parameterize functions and classes over types and values. When the concrete template argument type becomes known to the compiler, it replaces the corresponding type parameter (template instantiation), and type checks the instantiated template body. This compilation model is flexible, type safe, and can lead to high performance code [10]. For over a decade, C++ templates have helped deliver programs that are expressive, maintainable, efficient, and organized into highly reusable components [25]. Many libraries, such as the C++ Standard Template Library (STL) [5], the BOOST graph library [21], and the parallel computation system, STAPL [4], for which adaptability and performance are paramount, rest on the template mechanism.

C++ currently does not allow the requirements for the successful instantiation of a template to be explicitly stated. These requirements must be found in documentation or inferred from the template body. For attempts to instantiate a template with types that do not meet its requirements, current compilers often fail with hard to understand error messages [13]. Also, C++ provides only weak support for overloaded templates. While numerous programming techniques [1][3][18][20] offer partial solutions, they tend to raise the complexity.

Concepts [13][7][14] were developed to provide systematic remedies and deliver better support for the design and development of generic programs. As defined for C++0x, concepts improve expressiveness, make error messages more precise, and provide better control of the compile-time resolution of templates. Importantly, the use of concepts does not incur runtime overhead when compared to templates not using concepts. Despite many years design efforts, implementation work, and experimental use, concerns about usability, scalability, and the time needed to stabilize a design prevented concepts from being included as a language mechanism in the next revision of C++ [27][26]. However, we are left with a notation and a set of concepts developed for the STL and other libraries that can be used to describe and benchmark our use of design-level concepts.

In this paper, we present a novel tool for template analysis and concept extraction, TACE, that addresses some of these concerns. TACE extracts concept requirements from industrial-strength C++ code and helps apply concepts to unconstrained templated code. The paper offers the following contributions:

– A strategy for evolving generic code towards greater generality, greater uniformity, and more precise specification.
– Type level evaluation of uninstantiated function templates and automatic extraction of sets of requirements on template arguments.
– Concept analysis that takes called functions into account.

Experience with large amounts of generic C++ code and the development of C++ generic libraries, such as the generic components of the C++0x standard library [6], shows that the source code a template is not an adequate specification of its requirements. Such a definition is sufficient for type safe code generation, but even expert programmers find it hard to provide implementations that do not accidentally limit the applicability of a template (compared to its informal documentation). It is also hard to precisely specify template argument requirements and to reason about those.

Consequently, there is wide agreement in the C++ community that a formal statement of template argument requirements in addition to the template body is required. Using traditional type deduction techniques [8] modified to cope with C++, TACE generates such requirements directly from code. This helps programmers see the implications of implementation choices. Furthermore, the set of concepts generated from an implementation is rarely the most reusable or the simplest. To help validate a library implementation TACE compares the generated (implied) concepts to pre-defined library concepts.

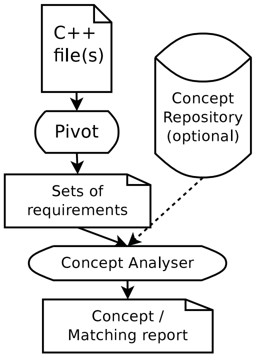


Fig. 1. The TACE tool chain

Fig. 1 shows TACE's tool chain. TACE utilizes the Pivot source-to-source transformation infrastructure [11] to collect and analyze information about C++ function templates. The Pivot's internal program representation (IPR) preserves high-level information present in the source code - it represents uninstantiated templates and is ready for concepts. TACE analyzes expressions, statements, and declarations in the body of function templates and extracts the requirements on template arguments. It merges the requirements with requirements extracted from functions that the template body potentially invokes. The resulting sets of requirements can be written out as concept definitions.

However, our goal is to find higher-level concepts that prove useful at the level of the design of software libraries. In particular, we do not just want to find the requirements of a particular implementation of an algorithm or the absolute minimal set of requirements. We want to discover candidates for concepts that are widely usable in interface specifications for algorithms. To recognize such concepts we need the "advice" of an experienced human. TACE achieves this by matching the extracted sets of requirements against concepts stored in a concept repository (e.g., containing standard concepts). In addition to reporting matches, the tool also reports close misses. In some cases, this allows programmers to reformulate their code to facilitate types that model a weaker concept. TACE does not try to discover semantic properties of a concept ("axioms" [12]). In general, doing so is beyond the scope of static analysis. Test results for STL indicate that our tool is effective when used in conjunction with a concept repository that contains predefined concepts.

The rest of the paper is organized as follows: §2 provides an overview of C++ with concepts, §3 presents the extraction of requirements from bodies of template functions, §4 describes requirement integration and reduction; §5 discusses matching the extracted requirements against predefined concepts from a repository, §7 discusses related work, and §8 presents a conclusion and an outlook on subsequent work.

2 Concepts for C++

Concepts as designed for C++0x [13][7][14] provide a mechanism to express constraints on template arguments as sets of syntactic and semantic requirements.

Syntactic requirements describe requirements such as associated functions, types, and templates that allow the template instantiation to succeed. Consider the following template, which determines the distance between two iterators:

```
template<typename Iterator>
size_t distance(Iterator first, Iterator last) {
  size_t n = 0;
  while (first != last) { ++first; ++n; }
  return n;
}
```

The function `distance` requires types that substitute for the type parameter `Iterator` have a copy constructor (to copy the arguments), a destructor (to destruct the argument at the end of the scope), an inequality (`!=`) operator, and an increment (`++`) operator. A requirement's argument type can be derived from the source code. Requirements can be stated using a C++ signature like notation.

```
concept DistanceRequirements<typename T> {
  T::T(const T&); // copy constructor
  T::~T(); // destructor
  bool operator!=(T, T);
  void operator++(T);
}
```

In order not to over-constrain templates, function signatures of types that model the concept need not match the concept signature exactly. Signatures in concepts allow automatic conversions of argument types. This means that an implementation of `operator!=` can accept types that are constructable from `T`.

The return type of a functional requirement has to be named but can remain unbound. The following example shows a function with two parameters of type `T`, where `T` is a template argument constrained by the concept `TrivialIterator`. The function tests whether the return values of the `operator*` are equal. The type of the return values is irrelevant as long as there exists an `operator==` that can compare the two. The result type of the equality comparison must be convertible to `bool`.

```
template <TrivialIterator T>
bool same_elements(T lhs, T rhs) {
  return (*lhs == *rhs);
}
```

Concepts introduce *associated types* to model such types. The following concept definition of `TrvialIterator` introduces such an associated type `ValueType` to specify the return type of `operator*`. Associated types can be constrained by nested requirements (e.g., the `requires` clause).

```
concept TrivialIterator<typename T> {
  typename ValueType;
  // nested requirements
  requires EqualityComparable<ValueType>; // operator== of ValueType
  ValueType operator*(T); // deref operator*
  ...
}
```

The compiler will use the concept definitions to type check expressions, declarations, and statements of the template body without instantiating it. Any type (or combination of types) that defines the required operations and types is a model of the concept. These types can be used for template instantiation.

Semantic requirements describe behavioral properties, such as the equivalence of operations or runtime complexity. Types that satisfy the semantic requirements are guaranteed to work properly with a generic algorithm. Axioms model some behavioral properties. Axioms can specify the equivalence of operations. The two operations are separated by an operator `<=>`. In the following example, the axiom `indirect_deref` specifies that the operations of the left and right side of `<=>` produce the same result. This is the case for pointers or random access iterators. Compilers are free to use axioms for code optimizations.

```
concept Pointer<typename T> {
  typename data;
  data operator*(T);
  T operator+(T, size_t);
  data operator[](T, size_t);

  axiom indirect_deref(T t, size_t n) {
    t[n] <=> *(t+n);
  }
}
```

Concepts can extend one or more existing concepts and add new requirements. Any requirement of the “base” concept remains valid for its *concept refinements.* Consider a trivial iterator abstraction, which essentially defines an operation to access its element (`operator*`). The concept `ForwardIterator` adds operations to traverse a sequential data structure in one way.

```
concept ForwardIterator<typename T> {
  requires TrivialIterator<T>;
  ...
}
```

Concept refinements are useful for the implementation of a family of generic functions. A base implementation constrains its template arguments with a general concept, while specialized versions exploit the stronger requirements of concept refinements to provide more powerful or more efficient implementations. Consider, the STL algorithm `advance(Iter, Size)` for which three different implementations exist. Its basic implementation is defined for input-iterators and has runtime complexity $O(Size)$. The version for bidirectional-iterators can handle negative distances, and the implementation for random access improves the runtime complexity to $O(1)$. The compiler selects the implementation according to the concept a specific type models [16].

Concepts can be used to constrain template arguments of stand-alone functions. In such a scenario, the extracted concept requirements reflect the function implementation directly. In the context of template libraries, clustering similar sets of requirements yields reusable concepts, where each concept constrains a family of types that posses similar qualities. Clustering requirements results in fewer and easier to comprehend concepts and makes concepts more reusable. An example of concepts, refinements, and their application is STL's iterator hierarchy, which groups iterators by their access capabilities.

Concept requirements are bound to concrete operations and types by the means of *concept maps*. Concept maps can be automatically generated. Should a type's operations not exactly match the requirement definition (e.g., when a function is named differently), concept maps allow for an easy adaptation [17].

3 Requirement Extraction

TACE extracts individual concept requirements from the body of function templates by inferring properties of types from declarations, statements, and expressions. Similar to the usage pattern style of concept specification [10], we derive the requirements by reading C++'s evaluation rules [9] backwards. We say backwards, because type checking usually tests whether expressions, statements, and declarations together with type (concept) constraints result in a well-typed program. In this work, we start with an empty set of constraints and derive the type (concept) constraints that make type-checking of expressions, statements, and declarations succeed. The derived constraints (ζ) reflect functional requirements and associated typenames.

3.1 Evaluation of Expressions

A functional requirement $op(arg_1, \ldots, arg_n) \rightarrow res$ is similar to a C++ signature. It consists of a list of argument types (arg) and has a result type (res). Since the concrete type of template dependent expressions is not known, the evaluator classifies the type of expressions into three groups:

Concrete types: this group comprises all types that are legal in non template context. It includes built-in types, user defined types, and templates that have been instantiated with concrete types. We denote types of this class with C.

Named template dependent types: this group comprises named but not yet known types (i.e., class type template parameters, dependent types, associated types, and instantiations that are parametrized on unknown types), and their derivatives. Derivatives are constructed by applying pointers, references, `const` and `volatile` qualifiers on a type. Thus, a template argument `T`, `T*`, `T**`, `const T`, `T&`, `typename traits<T>::value_type` are examples for types grouped into this category. We denote types of this class with T.

Requirement results: This group comprises fresh type variables. They occur in the context of evaluating expressions where one or more subexpressions have a non concrete type. The symbol R denotes types of this class. The types R are unique for each operation, identified by name and argument types. Only the fact that multiple occurrences of the same function must have the same return type, enables the accumulation of constraints on a requirement result. (e.g., the STL algorithm `search` contains two calls to `find`).

In the ensuing description, we use N for non concrete types $(T \cup R)$ and A for any type $(N \cup C)$. For each expression, the evaluator yields a tuple consisting of the return type and the extracted requirements. For example, $expr : C, \zeta$ denotes an expression that has a concrete type and where ζ denotes the requirements extracted for *expr* and its subexpressions. We use $X \rightsquigarrow Y$ to denote type X is convertible to type Y. Table 1 shows TACE's evaluation rules of expressions in a template body.

A *concrete expression* (*expr*) is an expression that does not depend on any template argument (e.g., literals, or expressions where all subexpressions (s) have concrete type). The result has a concrete type. The subexpressions have concrete type, but their subexpressions can be template dependent (e.g., `sizeof(T)`). Thus, ζ is the union of subexpression requirements.

Calls to *unbound functions* (*uf*) (and unbound overloadable operators, constructors, and destructor) have at least one argument that depends on an unknown type N. Since *uf* is unknown, its result type is denoted with a fresh type variable $R_{uf\ (s_1,\ \ldots\ s_n)}$.

Bound functions are functions, where the type of the function can be resolved at the compile time of the template body. Examples of bound functions include calls to member functions, where the type of the receiver object is known, and calls, where argument dependent lookup [15] is suppressed. Calls to bound functions (*bf*) have the result type of the callee. *bf*'s specification of parameter and return types can add conversion requirements to ζ (i.e., when the type of a subexpression differs from the specified parameter type and when at least one of these types is not concrete.)

The *conditional operator* (`?:`) cannot be overloaded. The ISO standard definition requires the first subexpression be convertible to `bool`. TACE's evaluation rules require the second and the third subexpression to be the same type. Here TACE is currently stricter than the ISO C++ evaluation which allows for a conversion of one of the result types.

The other not overloadable operators (i.e., `typeid` and `sizeof`) have a concrete result type. The set of extracted requirements is the same as for their subexpression or type.

Table 1. Evaluation rules for expressions

concrete expression	$\dfrac{\Gamma \overset{exp}{\vdash} s_1{:}C_1,\zeta_1 \ \dots \ s_n{:}C_n,\zeta_n}{\Gamma \overset{exp}{\vdash} expr(s_1,\ \dots,\ s_n){:}C_{expr},\ \bigcup\limits_{1\le i\le n} \zeta_i}$
unbound function	$\dfrac{\Gamma \overset{exp}{\vdash} s_1{:}A_1,\zeta_1 \ \dots \ s_n{:}A_n,\zeta_n}{\Gamma \overset{exp}{\vdash} uf(s_1,\dots,s_n){:}R_{uf(s1,\ \dots\ sn)},\ \bigcup\limits_{1\le i\le n} \zeta_i \cup \{uf(A_1,\ \dots\ ,A_n)\rightarrow R_{uf(s_1,\ \dots\ s_n)}\}}$
bound function	$\dfrac{\Gamma \overset{exp}{\vdash} (bf{:}(A_1^{bf},\dots,A_n^{bf})\rightarrow A_r^{bf})\ s_1{:}A_1,\zeta_1 \ \dots \ s_n{:}A_n,\zeta_n}{\Gamma \overset{exp}{\vdash} fn(s_1,\ \dots,\ s_n){:}A_r^{bf},\ \bigcup\limits_{1\le i\le n} \zeta_i \cup \{A_i \rightsquigarrow A_i^{bf}\}}$
conditional operator	$\dfrac{\Gamma \overset{exp}{\vdash} s_1{:}A_1,\zeta_1\ \ s_2{:}A_2,\zeta_2\ \ s_3{:}A_2,\zeta_3}{\Gamma \overset{exp}{\vdash} (s_1?s_2{:}s_3){:}\ A_2,\ \bigcup\limits_{1\le i\le 3} \zeta_i \cup \{A_1 \rightsquigarrow bool\}}$
member functions	$\dfrac{\Gamma \overset{exp}{\vdash} o{:}A_o,\zeta_o\ \ s_1{:}A_1,\zeta_1\ \ s_n{:}A_n,\zeta_n}{\Gamma \overset{exp}{\vdash} o.uf(s_1,\dots,s_n){:}R_{uf\ (o,s_1,\ \dots\ s_n)},\zeta_o \cup \bigcup\limits_{1\le i\le n} \zeta_i \cup \{uf(A_o,A_1,\ \dots\ ,A_n)\rightarrow R_{uf(o,s_1,\ \dots\ s_n)}\}}$
non concrete arrow	$\dfrac{\Gamma \overset{exp}{\vdash} o{:}A_o,\zeta_o}{\Gamma \overset{exp}{\vdash} o\texttt{->}{:}R_{\texttt{->}o},\{operator\texttt{->}(A_o)\rightarrow R_{\texttt{->}o}\}}$
static cast	$\dfrac{\Gamma \overset{exp}{\vdash} A_{tgt},\zeta_{tgt}\ \ o{:}A_o,\zeta_o}{\Gamma \overset{exp}{\vdash} A_{tgt},\{A_o \rightsquigarrow A_{tgt}\}}$
dynamic cast	$\dfrac{\Gamma \overset{exp}{\vdash} A_{tgt},\zeta_{tgt}\ \ o{:}A_o,\zeta_o}{\Gamma \overset{exp}{\vdash} A_{tgt},\{PolymorphicClass{<}A_o{>}\}}$
other casts	$\dfrac{\Gamma \overset{exp}{\vdash} A_{tgt},\zeta_{tgt}\ \ o{:}A_o,\zeta_o}{\Gamma \overset{exp}{\vdash} A_{tgt},\{\}}$

C++ concepts do not support modeling of member variables. A member selection (i.e, the `dot` or `arrow operator` can only refer to a member function name. The evaluator rejects any dot expression that occurs not in the context of evaluating the receiver of a call expression.

For *non concrete objects*, the evaluator treats the `arrow` as a unary operator that yields an object of unknown result type. The object becomes the receiver of a subsequent call to an unbound member function.

`Cast` expressions are evaluated according to the rules specified in Table 1. The target type of a cast expression is also evaluated and can add dependent name requirements to ζ. A `static_cast` requires the source type be convertible to the target type. A `dynamic_cast` requires the source type to be a polymorphic class (`PolymorphicClass` is part of C++ with concepts).

The evaluation of operations on pointers follows the regular C++ rules, thus the result of dereferencing $T*$ yields $T\&$, the arrow operator yields a member function selection of T, taking the address of $T*$ yields T**, and any arithmetic expression on $T*$ has type $T*$. *Variables* in expressions are typed as lvalues of their declared type.

3.2 Evaluation of Declarations and Statements

Table 2 shows the evaluation rules for statements and declarations (of variables and parameters).

Statements: The condition expressions of `if`, `while`, `for` require the expression to be convertible to `bool`. The `return` statement requires convertibility of the expression to the function return type. The expression of the `switch` statement is either convertible to `signed` or `unsigned` integral types. We introduce an artificial

Table 2. Evaluation rules for statements and declarations

statement context	$\dfrac{\Gamma \overset{stmt}{\vdash} \tau{\in}N, s{:}A, \zeta}{\epsilon, \zeta + A \rightsquigarrow \tau}$
default ctor	$\dfrac{\Gamma \overset{decl}{\vdash} o{:}(\Gamma, \tau{\in}N_o)}{\Gamma \overset{decl}{\vdash} o{:}\tau, \{\tau{::}ctor()\}}$
single argument ctor	$\dfrac{\Gamma \overset{decl}{\vdash} o{:}(\Gamma, \tau{\in}N_o), s_1{:}A_1, \zeta_1}{\Gamma \overset{decl}{\vdash} o{:}\tau, \zeta_1 + \tau{::}ctor(const\tau\&) + A_0 \rightsquigarrow \tau}$
constructor	$\dfrac{\Gamma \overset{decl}{\vdash} o{:}(\Gamma, \tau{\in}N_o), s_1{:}A_1, \zeta_1,\ ...,\ s_n{:}A_n, \zeta_n}{\Gamma \overset{decl}{\vdash} o{:}\tau, \bigcup_{1 \le i \le n} \zeta_i + \tau{::}ctor(A_1,\ ...,\ A_n)}$
parameter	$\dfrac{\Gamma \overset{decl}{\vdash} p{:}(\Gamma, \tau{\in}N_o)}{\Gamma \overset{decl}{\vdash} p{:}\tau, \{\tau{::}ctor(A_1,\ ...,\ A_n)\}}$

type `Integer` that subsumes both types. The type will be resolved later, if more information becomes available.

Object declarations: Variable declarations of object type require the presence of a constructor and destructor. Single argument construction (i.e., $T\ t\ =\ arg$) is modeled to require $A_{arg} \rightsquigarrow T$ and a copy constructor on T.

References: Bindings to lvalue (mutable) references (i.e., declarations, function calls, and `return` statements) impose stricter requirements. Instead of convertibility, they require the result type of an expression be an exact type (instead of a convertible type).

3.3 Evaluation of Class Instantiations

The current implementation focuses on extracting requirements from functions, and thus treats any instantiation of classes that have data members and where the template parameter is used as template argument as concrete type (e.g. `pair`, `reverse_iterator`); ζ remains unchanged. To allow the analysis of real world C++ function templates, TACE analyzes classes that contain static members (types, functions, and data). Particularly, trait classes can add dependent type requirements to ζ. For example, the instantiation of `iterator_traits<T>::value_type` leads to the type constraint `T::value_type`.

Static member function (templates) of class templates (e.g.: the various variants of `sort`) are treated as if they were regular function templates. The template parameters of the surrounding class extend the template parameters of the member function. For example:

```
template <class T>
struct S { template <class U> static T bar(U u); };
```

is treated as:

```
template <class T, class U> T bar(U u);
```

3.4 Example

We use the beginning of GCC's (4.1.3) implementation of the STL algorithm `search` to illustrate our approach. Fig. 2 shows a portion of the implementation and the requirements that get extracted from it.

We begin by extracting the requirements from the argument list. Their types are `FwdIter1` and `FwdIter2`. They are passed by value. According to the evaluation rule for parameters, their type has to support copy construction and destruction.

The condition of the `if` statement is evaluated bottom up. The right hand side of the `operator||` is an equality comparison (`operator==`) of two parameters of type `FwdIter1&`. Since `FwdIter1` is an unknown type, the operation is evaluated according to the unbound function rule. The result type of operator is a fresh type variable ($r1$). The set of requirements consists of the equality comparison `operator==(FwdIter1&, FwdIter1&)` $\rightarrow$ r_1. Similarly, the

```
template<typename FwdIter1,
           typename FwdIter2>
FwdIter1
search(FwdIter1 first1, FwdIter1 last1,
       FwdIter2 first2, FwdIter2 last2) {

  if (first1 == last1 || first2 == last2)
    return first1;

  FwdIter2 tmp(first2);
  ++tmp;
  if (tmp == last2)
    return find(first1, last1, *first2);

  FwdIter2 p1, p;
  p1 = first2;
  ++p1;
  FwdIter1 current = first1;
  while (first1 != last1) {
    first1 = find(first1, last1, *first2);
    // ...
```

```
concept Search <typename FwdIter1, typename FwdIter2> {
  // argument construction
  FwdIter1::FwdIter1(const FwdIter1&);
  FwdIter1::~FwdIter1();
  FwdIter2::FwdIter2(const FwdIter2&);
  FwdIter2::~FwdIter2();

  // if statement and return
  typename r1;
  r1 operator==(FwdIter1&, FwdIter1&);
  typename r2;
  r2 operator==(FwdIter2&, FwdIter2&);
  typename r3;
  r3 operator||(r1, r2);
  operator bool(r3);

  // second range has length 1
  r5 operator++(FwdIter2&);
  operator bool(r2);
  typename r7;
  r7 operator*(FwdIter2&);
  typename r8;
  r8 find(FwdIter1&, FwdIter1&, r7);
  operator FwdIter1(r8);

  // while loop
  FwdIter2::FwdIter2(); // default constructor
  void operator=(FwdIter2&, FwdIter2&);
  typename r12;
  r12 operator!=(FwdIter1&, FwdIter1&);
  operator bool(r12);
  void operator=(FwdIter1&, r8);
  // ...
```

Fig. 2. Requirement extraction

evaluation of the comparison of `first2` and `last2` yields $\mathtt{r}_2$ and the requirement `operator==(FwdIter2&, FwdIter2&)` $\longrightarrow \mathtt{r}_2$ The evaluation proceeds with the `operator||`. Both arguments have an undetermined type ($\mathtt{r}_1$ and $\mathtt{r}_2$). Thus, the operation is evaluated according to the unbound function rule. The result type is $\mathtt{r}_3$. `operator==`$(\mathtt{r}_1, \mathtt{r}_2) \longrightarrow \mathtt{r}_3$ and the requirements extracted for the subexpressions form the set of requirements. $\mathtt{r}_3$ is evaluated by an if statement. According to the rule statement context, $\mathtt{r}_3$ has to be convertible to `bool`. The return statement does not produce any new requirement, because the copy constructor of `FwdIter1` is already part of $\zeta_{\mathtt{search}}$.

The next source line declares a local variable `tmp`. Its initial value is constructed from a single argument of the same type. Thus, this line requires a copy constructor on `FwdIter2` (evaluation rule for constructors). The next line moves the iterator `tmp` forward by one. The source line is evaluated according to the unbound function rule. The return type is $\mathtt{r}_5$, the extracted requirement is `operator++(FwdIter2&)` $\longrightarrow \mathtt{r}_5$. The expression of the following `if` statement compares two expressions of type `FwdIter2&`. Unification with the already extracted requirements in $\zeta_{\mathtt{search}}$ yields the result type $\mathtt{r}_2$. Evaluating $\mathtt{r}_2$ according to the statement context rule yields an additional conversion requirement $\mathtt{r}_2 \rightsquigarrow$ `bool`. The next line of code returns the result of a function call. First, the argument list of the call is processed. The first two arguments are references to parameters of type `FwdIter1`; the third argument dereferences a parameter of type `FwdIter2`.

According to the unbound function rule, this expression yields to a new result type `r`$_7$ and the requirement `operator*(FwdIter2&)` $\rightarrow$ `r`$_7$. Then TACE applies the unbound function rule to the function call itself. This yields the result type `r`$_8$ and the requirement `find(FwdIter1&, FwdIter1&, r`$_7$`)` $\rightarrow$ `r`$_8$. From the statement context, we infer `r`$_8$ $\rightsquigarrow$ `FwdIter1`.

The declarations of `p1` and `p` require `FwdIter2` support default construction (`FwdIter2::FwdIter2()`). We skip the remaining code and requirements.

4 From Requirements to Concepts

The requirement extraction generates functional requirements for all calls. This includes calls to algorithms that potentially resolve to other generic functions. For example, the requirements that were extracted from `search` (§3.4) contain two calls to a function `find`. We choose to merge the requirements of the callee into the caller's set of requirements, if there is a function template with the same name and where that function's parameters are at least as general as the arguments of the call expression.

In the requirements set, any result of an operation is represented by a fresh type variable (i.e., associated types such as `r1` and `r2` in §3.4). However, the evaluation context contributed more information about these types in form of conversion requirements. TACE defines a function *reduce* that replaces type variables with the target type of conversion requirements and propagates the results in the requirement set.

$$\mathit{reduce}(\zeta) \rightarrow \zeta'$$

Should a requirement result have more than one conversion targets (for example, an unbound function was evaluated in the context of `bool` and `int`), we apply the following subsumption rule: assuming n conversion requirements with the same input type (R) but distinct target types A_i.

$$R' = \begin{cases} A_j & \text{if } \exists_j \forall_i \text{ such that } A_j \rightsquigarrow A_i \\ R & \text{otherwise} \end{cases}$$

Note, that the $A_j \rightsquigarrow A_i$ must be part of ζ, or defined for C++ built in types. If such an A_j exists, all operations that depend on R are updated, and become dependent on A_j. Any conversion requirement on R is dropped from ζ. When R is not evaluated by another function it gets the result type `void`. If R is evaluated by another expression, but no conversion requirement exists, the result type R' remains unnamed (i.e. becomes an associated type).

After the return type has been determined, the new type R' is propagated to all operations that use it as argument type. By doing so, the set of requirements can be further reduced (e.g., if all argument types of an operation are in C, the requirement can be eliminated, or in case the operation does not exist, an error reported) and more requirement result types become named (if an argument type becomes T, another operation on T might already exist). Reduction is a

```
// search's requirements
FwdIter1::FwdIter1(const FwdIter1&);
FwdIter1::~FwdIter1();
FwdIter2::FwdIter2(const FwdIter2&);
FwdIter2::~FwdIter2();
bool operator==(FwdIter1&, FwdIter1&);
bool operator==(FwdIter2&, FwdIter2&);
typename r4;
r4 operator++(FwdIter2&);
typename r5;
r5 operator*(FwdIter2&);
FwdIter2::FwdIter2();
void operator=(FwdIter2&, FwdIter2&);
bool operator!=(FwdIter1&, FwdIter1&);
void operator=(FwdIter1&, FwdIter1&);
typename r11;
r11 operator++(FwdIter1&);
bool operator==(r11, FwdIter1&);
typename r13;
r13 operator*(FwdIter1&);
bool operator==(r13, r5);
bool operator==(r4, FwdIter2&);
typename r529;              // from find
r529 operator==(r13, const r5&);
typename r530;
r530 operator!(r529);
bool operator&&(bool, r530);
```

Fig. 3. Requirements after reduction

```
// requirements on FwdIter1
FwdIter1::FwdIter1(const FwdIter1&);
FwdIter1::~FwdIter1();
typename r1652;
r1652 operator==(FwdIter1&, FwdIter1&);
typename r1654;
r1654 operator!=(FwdIter1&, FwdIter1&);
operator bool (r1654);
operator bool (r1652);
void operator=(FwdIter1&, FwdIter1&);
typename r1658;
r1658 operator++(FwdIter1&);
typename r1659;
r1659 operator==(r1658, FwdIter1&);
operator bool (r1659);
typename r1661;
r1661 operator*(FwdIter1&);
```

Fig. 4. Kernel of `FwdIter1`

repetitive process that stops when a fixed point is reached. We show two examples for the requirement set in Fig. 2. The conversion `operator FwdIter1(r8)` allows `FwdIter1&` substitute for `r8` in `void operator=(FwdIter1&, r8)`. Similar `r1` and `r2` are convertible to `bool`, thus `r3 operator||(r1, r2)` can be dropped. The number of reductions depends on how much context information is available in code. Fig. 3 shows the result after merging and reducing the requirements of `search` and `find`.

The result of `reduce` may constrain the type system more than the original set of requirements. Thus, `reduce` has to occur after merging all requirements from potential callees, when all conversion requirements on types are available.

5 Recovery from Repository

Template libraries utilize concepts to constrain the template arguments of a group of functions that operate on types with similar capabilities. These concepts provide a design vocabulary for library domain and help provide a degree of pluggability among algorithms and types. Without those, even the slightest change to an implementation will cause a recompilation of every user; an implementation is not a good general specification of an algorithm. To find reusable components within the extracted requirements, we use a concept repository, which contains a number of predefined concept definitions (e.g., core concepts or concepts that users define for specific libraries). The use of a concept repository offers users the following benefits:

– reduces the number of concepts and improves their structure
– exposes the refinement relationships of concepts
– replaces requirement results with named types (concrete, template dependent, or associated typenames)

The repository we use to drive the examples in this sections contains the following concepts: `IntegralType<T>`, `RegularType<T>`, `TrivialIterator<T>`, `ForwardIterator<T>`, `BidirectionalIterator<T>`, `RandomaccessIterator<T>`, and `EqualityComparable<T>`. `IntegralType` specifies operations that are defined on type `int`. `RegularType` specifies operations that are valid for all built-in types (i.e., default construction, copy construction, destruction, assignment, equality comparison, and address of). `TrivialIterator` specifies the dereference operation and associated iterator types. The other iterators have the operations defined in the STL. In addition, the repository contains concepts defined over multiple template arguments (`UnaryFunction`, `UnaryPredicate`, `BinaryFunction` and `BinaryPredicate`). The predicates refine the functions and require the return type be convertible to `bool`.

5.1 Concept Kernel

In order to match the extracted requirements of each template argument against concepts in the repository that depend on fewer template arguments, we partition the unreduced set into smaller sets called *kernels*. We define a concept kernel over a set of template arguments $\widehat{T}$ to be a subset of the original set ζ.

$$kernel(\zeta_{function}, \widehat{T}) \rightarrow \zeta_{kernel}$$

ζ_{kernel} is a projection that captures all operations on types that directly or indirectly originate from the template arguments in $\widehat{T}$.

$$\zeta_{kernel} \Leftrightarrow \{op|op \in \zeta_{src}, \phi_{\widehat{T}}(op)\}$$

For the sake of brevity, we also say that a type is in ζ_{kernel}, if the type refers to a result R of an operation in ζ_{kernel}.

$$\phi_{\widehat{T}}(op) = \begin{cases} 1 & \text{true for a } op(arg_1, \dots, arg_n) \rightarrow res \\ & \text{if } \forall_i \; arg_i \in \widehat{T} \cup \zeta_{kernel} \cup C \\ 0 & \text{otherwise} \end{cases}$$

$\phi_{\widehat{T}}(op)$ is true, if all arguments of *op* either are in $\widehat{T}$, are result types from operations in ζ_{kernel}, or are concrete. Fig. 4 shows the concept kernel for the first template argument of `search`.

5.2 Concept Matching

For each function, TACE type checks the kernels against the concepts in the repository. The mappings from a kernel's template parameters to a concept's template parameters are generated from the arguments of the first operation in a concept kernel and the parameter declarations of operations with the same name in the concept.

For any requirement in the kernel, a concept has to contain a single best matching requirement (multiple best matching signatures indicate an ambiguity). TACE checks the consistency of a concept requirement's result type with all conversion requirements in the kernel.

For each kernel and concept pair, TACE partitions the requirements into satisfiable, unsatisfiable, associated, and available functions. An empty set of unsatisfiable requirements indicates a match. TACE can report small sets of unsatisfiable requirements (i.e., near misses), thereby allowing users to modify the function implementation (or the concept in the repository) to make a concept-function pair work together. The group of associated requirements contains unmatched requirements on associated types. For example, any iterator requires the value type to be regular. Besides regularity, some functions such as `lower_bound` require less than comparability of container elements. Associated requirements are subsequently matched. The group of available functions contains requirements, where generic implementations exist.

This produces a set of candidate concepts. For example, the three iterator categories match the template parameters of `search`. `find` has two template arguments. Any iterator concept matches the first argument. Every concept in the repository matches the second argument.

For functions with more than one template argument, TACE generates the concept requirements using a Cartesian join from results of individual kernels.

The final step in reducing the candidate concepts is the elimination of superfluous refinements. A refinement is superfluous if one of its base concepts is also a candidate, and if the refinement cannot provide better typing. In the case of `search`, this results in `ForwardIterator<FwdIter1>`, `ForwardIterator<FwdIter2>`, and the following unmatched requirement:

```
bool operator==( iterator_traits<FwdIter1>::value_type&, iterator_traits<FwdIter2>::value_type& );
```

Matching currently does not generate any new conversion requirements. Conversions could be generated for simple cases, but in general several alternative resolutions are possible and the tool has no way of choosing between them. For example, to match `EqualityComparable<T>`, the operands of `operator==` must be converted to a common type, but knowing only that each operand is Regular, we cannot know which to convert.

With *better typing* we mean that a concept's constraints can be used to type-check more requirements. Consider STL's algorithm `find_if`, that iterates over a range of iterators until it finds an element, where a predicate returns `true`.

```
while (first != last && !pred(*first)) // ...
```

For `pred(*first)`, both UnaryFunction and UnaryPredicate are candidates. While the latter is able to type-check the `operator!` based on `bool`, the former needs additional requirements for the negation and logical-and operators.

5.3 Algorithm Families

A generic function can consist of a family of different implementations, where each implementation exploits concept refinements (e.g., `advance` in §2). A template that calls a generic function needs to incorporate the requirements of the most general algorithm. Finding the base algorithm is non trivial with real code. Consider STL's `advance` family. TACE extracts the following requirements:

```
// for Input−Iterators
concept AdvInputIter <typename Iter, typename Dist> {
  Dist::Dist(const Dist&);
  void operator++(Iter&);
  bool operator−−(Dist&, int);
}
// for Bidirectional−Iterators
concept AdvBidirectIter <typename Iter, typename Dist> {
  Dist::Dist(const Dist&);
  void operator++(Iter&);
  void operator−−(Iter&);
  bool operator++(Dist&, int);
  bool operator−−(Dist&, int);
}
// for Randomaccess−Iterators
concept AdvRandomaccessIter <typename Iter, typename Dist> {
  Dist::Dist(const Dist&);
  void operator+=(Iter&, Dist&);
}
```

The sets of extracted requirements for the implementations based on input- and bidirectional-iterator are in a subset/superset relation, the set of requirements for the random access iterator based implementation is disjoint with the former sets. If such calls occur under scenario §4, TACE requires the user mark the least specific function implementation. A concept repository helps infer the correct refinement relationship.

6 Validation

We validated the approach by matching the functions defined in GCC's header file `algorithm`. The file contains more than 9000 non-comment (and non empty) lines of code and defines 115 algorithms plus about 100 helper functions. The `algorithm` header exercises some of the most advanced language features and design techniques used for generic programming in C++.

The success rate of the concept recovery depends on the concepts in the repository. A repository containing syntactically similar concepts will lead to ambiguous results. We ran the tests against the repository introduced in §5.

TACE found a number of functions, where the annotations in code overly constrain the template arguments, such as `__unguarded_linear_insert` (STL's specifications are meaningful though, as the identified functions are helpers of algorithms requiring random access.) Static analysis tools, such as TACE, are limited to recovering syntactic requirements (such as, operations used and typenames referred to), but cannot deduce semantic details from code. For example, a forward iterator differs only semantically from an input iterator (a forward iterator can be used in multi-pass algorithms). Also, consider `find_end` for bidirectional iterators, which takes two different iterator types. The second iterator type requires only forward access and the existence of `advance(it, n)`. `n`'s possible negativity is what requires bidirectional access. Over the entire test set, TACE currently recognizes about 70% of iterator concepts correctly and unambiguously. For about 10% TACE produces a false positive match (e.g., `IntegralType`) alongside the correct iterator concept. For the input set, TACE classifies all functions and predicates (including template arguments marked as `Compare` and `StrictWeakOrding`) correctly, but due to the reason stated at the end of §5.2 does not generate the conversion requirement on result types.

7 Related Work

Sutton and Maletic [28] describe an approach to match requirements against a set of predefined concepts based on formal concept analysis. Their analysis tool finds combinations of multiple concepts, if needed, to cover the concept requirements of function templates. In order to avoid complications from "the reality of C++ programming" [28], the authors validate their approach with a number of re-implemented STL algorithms, for which they obtain results that closely match the C++ standard specification. The authors discuss how small changes in the implementation (e.g., the use of variables to store intermediate results provides more type information) can cause small variations in the identified concepts.

Dos Reis and Stroustrup [10] present an alternative idea for concept specification and checking. Their approach states concepts in terms of usage patterns, a form of requirement specification that mirrors the declarations and expressions in the template body that involve template arguments. If type checking of a concrete type against the usage pattern succeeds, then template instantiation will succeed too. In essence, TACE reverses this process and derives the requirements from C++ source code and converts them into signature based concepts.

The aim of type inference for dynamically typed languages, the derivation of type annotations from dynamically typed code, is somewhat similar to concept recovery. For example, Agesen et al [2]'s dynamic type inference on SELF generates local constraints on objects from method bodies. By analyzing edges along trace graphs their analysis derives global constraints from local constraints. This kind of analysis differs from our work in various ways. Agesen et al start at a specific entry point of a complete program (i.e., function `main` in a C++ program). This provides concrete information on object instantiations from prototypes. Concept recovery neither depends on a single entry point, nor does the analyzed program have to be complete (instantiations are not required). Moreover, concept recovery is concerned with finding higher level abstractions that describe multiple types (concepts).

Matching the requirements against the definitions in the concept repository is similar to the Hindley-Milner-Damas (HMD) type inference algorithm for functional languages [8]. HMD and its variations derive a type scheme for untyped entities of a function from type annotations and the utilization of these entities in the context of defined functions. Type classes [29] help overcome problems of the type inference algorithm that stem from operator overloading. Peterson and Jones [19] present an extension to the HMD algorithm, which utilizes type classes to derive an unambiguous type scheme for polymorphic functions.

Our type checking mechanism that tests the requirement sets against concepts in a repository differs from type checking with type classes in several ways:

– C++ template code follows regular C++ programming style, where variables have to be declared before they can be used. The type of a variable can be template argument dependent. C++ concepts allow overloaded function requirements (e.g., random access iterator's subtraction and difference requirement). The types of variable declarations provide the type information that we use for overload resolution.
– C++'s type system allows type coercions. Based on C++ binding rules and conversion (and constructor) requirements that are defined in the concept, TACE generates all possible type combinations that a specific function requirement can handle. For example, if a signature is defined over `const T&` another signature for `T&` is added to the concept.
  Consequently, checking whether a requirement kernel is expressible by a concept in the repository relies on signature unification. The result type of a function is inferred from the requirement specification in the repository.
– in C++, type checking of expressions is context dependent. For example the disambiguation of overloaded functions relies on the expression context. In a call context, this is the argument list of the call. When a function is assigned to a function pointer, the context is provided by the type of the function pointer (i.e., the left hand side of an assignment determines the type that is expected on the right hand side). When code suppresses argument dependent lookup, the rules become more subtle. In this circumstance, the overload set only contains functions that were available at template definition time.
– Haskell type checking utilizes context reduction. For example, an equality implementation on lists may require that the list elements be comparable. Context reduction, extracts the requirements on the element type from the container context. Extending TACE with context reduction would be useful for deriving requirements from templated data-structures (see §3.3).

8 Conclusion and Future Work

TACE is a part of a larger project to raise the level of abstraction of existing C++ programs through the use of high-level program analysis and transformations. In this paper, we have presented our tool, TACE, that extracts sets of requirements from real C++ code. TACE analyzes these requirements and generates concept definitions for functions. Alternatively, our tool clusters requirement sets into concepts by matching against predefined concepts in a repository.

Our results show that a static tool, limited to analyzing syntactic requirements, will not always be able to infer concepts correctly. However, for a large number of industrial-strength function templates, TACE can recover and cluster constraints correctly. The Pivot system provides an extensible compiler based framework that enables enhancements of our analysis.

TACE's current implementation searches the repository for suitable concepts. A formal analysis based approach [28] could be applicable to more library domains. Also, a more precise analysis of class templates (data structures, trait classes, tag hierarchies, etc.) will improve the analysis precision.

Acknowledgments

The authors would like to thank Gabriel Dos Reis, Gary Leavens, Jaakko Järvi, Yuriy Solodkyy, and the anonymous referees for their helpful comments and suggestions.

References

1. Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series). Addison-Wesley Professional, Reading (2004)
2. Agesen, O., Palsberg, J., Schwartzbach, M.: Type inference of SELF. In: Wang, J. (ed.) ECOOP 1993. LNCS, vol. 707, pp. 247–267. Springer, Heidelberg (1993)
3. Alexandrescu, A.: Modern C++ design: generic programming and design patterns applied. Addison-Wesley Longman Publishing Co., Inc., Boston (2001)
4. An, P., Jula, A., Rus, S., Saunders, S., Smith, T., Tanase, G., Thomas, N., Amato, N., Rauchwerger, L.: STAPL: A standard template adaptive parallel C++ library. In: Dietz, H.G. (ed.) LCPC 2001. LNCS, vol. 2624, pp. 193–208. Springer, Heidelberg (2003)
5. Austern, M.H.: Generic programming and the STL: using and extending the C++ Standard Template Library. Addison-Wesley Longman Publishing Co., Inc., Boston (1998)
6. Becker, P.: The C++ Standard Library Extensions: A Tutorial and Reference, 1st edn. Addison-Wesley Professional, Boston (2006)
7. Becker, P.: Working draft, standard for programming language C++. Tech. Rep. N2914 (June 2009)
8. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: POPL 1982: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 207–212. ACM, New York (1982)
9. Dos Reis, G., Stroustrup, B.: A C++ formalism. Tech. Rep. N1885, JTC1/SC22/WG21 C++ Standards Committee (2005)
10. Dos Reis, G., Stroustrup, B.: Specifying C++ concepts. In: POPL 2006: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 295–308. ACM Press, New York (2006)
11. Dos Reis, G., Stroustrup, B.: A principled, complete, and efficient representation of C++. In: Suzuki, M., Hong, H., Anai, H., Yap, C., Sato, Y., Yoshida, H. (eds.) The Joint Conference of ASCM 2009 and MACIS 2009, COE, Fukuoka, Japan. MI Lecture Note Series, vol. 22, pp. 151–166 (December 2009)
12. Dos Reis, G., Stroustrup, B., Meredith, A.: Axioms: Semantics aspects of C++ concepts. Tech. Rep. N2887, JTC1/SC22/WG21 C++ Standards Committee (June 2009)

13. Gregor, D., Järvi, J., Siek, J., Stroustrup, B., Dos Reis, G., Lumsdaine, A.: Concepts: linguistic support for generic programming in C++. In: OOPSLA 2006: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 291–310. ACM Press, New York (2006)
14. Gregor, D., Stroustrup, B., Siek, J., Widman, J.: Proposed wording for concepts (revision 4). Tech. Rep. N2501, JTC1/SC22/WG21 C++ Standards Committee (February 2008)
15. ISO/IEC 14882 International Standard: Programming languages: C++. American National Standards Institute (September 1998)
16. Järvi, J., Gregor, D., Willcock, J., Lumsdaine, A., Siek, J.: Algorithm specialization in generic programming: challenges of constrained generics in C++. In: PLDI 2006: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 272–282. ACM, New York (2006)
17. Järvi, J., Marcus, M.A., Smith, J.N.: Library composition and adaptation using C++ concepts. In: GPCE 2007: Proceedings of the 6th International Conference on Generative Programming and Component Engineering, pp. 73–82. ACM Press, New York (2007)
18. Järvi, J., Willcock, J., Hinnant, H., Lumsdaine, A.: Function overloading based on arbitrary properties of types. C/C++ Users Journal 21(6), 25–32 (2003)
19. Peterson, J., Jones, M.: Implementing type classes. In: PLDI 1993: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, pp. 227–236. ACM Press, New York (1993)
20. Siek, J., Lumsdaine, A.: Concept checking: Binding parametric polymorphism in C++. In: First Workshop on C++ Template Programming, Erfurt, Germany, October 10 (2000)
21. Siek, J.G., Lee, L.Q., Lumsdaine, A.: The Boost Graph Library: user guide and reference manual. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
22. Stepanov, A., McJones, P.: Elements of Programming. Addison-Wesley Professional, Reading (2009)
23. Stroustrup, B.: The design and evolution of C++. ACM Press/Addison-Wesley Publishing Co, New York, NY, USA (1994)
24. Stroustrup, B.: The C++ Programming Language. Addison-Wesley Longman Publishing Co., Inc., Boston (2000)
25. Stroustrup, B.: Abstraction and the C++ machine model. In: Wu, Z., Chen, C., Guo, M., Bu, J. (eds.) ICESS 2004. LNCS, vol. 3605, pp. 1–13. Springer, Heidelberg (2005)
26. Stroustrup, B.: The C++0x "remove concept" decision. Dr. Dobb's Journal 92 (August 2009) (republished with permission in Overload Journal July 2009)
27. Stroustrup, B.: Expounds on concepts and the future of C++. Interview with Danny Kalev (August 2009)
28. Sutton, A., Maletic, J.I.: Automatically identifying C++0x concepts in function templates. In: ICSM 2008: 24th IEEE International Conference on Software Maintenance, Beijing, China, pp. 57–66. IEEE, Los Alamitos (2008)
29. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: POPL 1989: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 60–76. ACM, New York (1989)

Automated Co-evolution of GMF Editor Models

Davide Di Ruscio[1], Ralf Lämmel[2], and Alfonso Pierantonio[1]

[1] Computer Science Department, University of L'Aquila, Italy
[2] Software Languages Team, Universität Koblenz-Landau, Germany

Abstract. The Eclipse Graphical Modeling (GMF) Framework provides the major approach for implementing visual languages on top of the Eclipse platform. GMF relies on a family of modeling languages to describe abstract syntax, concrete syntax as well as other aspects of the visual language and its implementation in an editor. GMF uses a model-driven approach to map the different GMF models to Java code. The framework, as it stands, lacks support for evolution. In particular, there is no support for propagating changes from the domain model (i.e., the abstract syntax of the visual language) to other editor models. We analyze the resulting co-evolution challenge, and we provide a solution by means of GMF model adapters, which automate the propagation of domain-model changes. These GMF model adapters are special model-to-model transformations that are driven by difference models for domain-model changes.

1 Introduction

In the context of Model Driven Engineering (MDE) [2], the definition of a domain-specific modeling language (DSML) or its implementation in an editor (or another tool) consists of *a collection of coordinated models*. These models specify the abstract and concrete syntaxes of the language, and possibly further aspects related to semantics or the requirements of a particular DSML tool. The increasing understanding of the problem domain for DSML may necessitate continuous evolution. Hence, DSML have to evolve, and DSL tools have to co-evolve [11].

In the present paper, we make a contribution to the general theme of *evolution for DSMLs* by addressing the more specific problem of supporting co-evolution between the coordinated models that constitute the definition of a DSML, or, in fact, its implementation in a graphical editor. We focus on the *propagation of abstract-syntax changes to other models*, e.g., the model for a graphical, concrete syntax.

In MDE, the abstract syntax of DSMLs is typically expressed in terms of metamodels which are created by means of generic modeling environments, e.g., the Eclipse Modeling Framework (EMF) [9,3]. Indeed, we leverage EMF in the present paper. Further, we build upon the Eclipse Graphical Modeling Framework (GMF) for developing graphical editors based on EMF and other Eclipse components [14]. Arguably, GMF defines the mainstream approach to graphical editor development within the Eclipse platform. GMF uses a generative approach to obtain a working editor implementation (in Java) from the coordinated models of the editor for a DSML.

For illustration, consider the simple mind-map editor in Fig. 1. We have annotated the different panes of the editor with the associated GMF models underneath. The domain

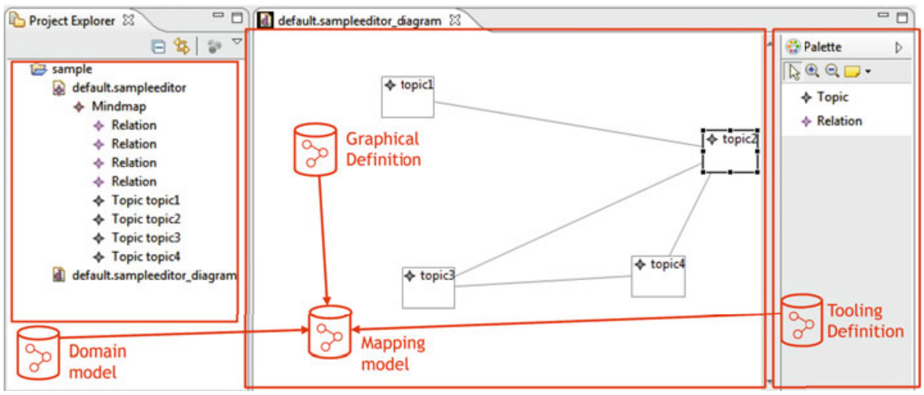


Fig. 1. Snapshot of a simple editor with indications of underlying models

model is concerned with the abstract syntax. The graphical and the tooling definition are concerned with concrete syntax and the editor functionality. The mapping model connects the various models. We will describe the architecture of the editor in more detail later. In the GMF approach, model-to-model and model-to-code transformations derive the implementation of a graphical editor which provides the means for editing models conforming to the specified domain model.

With such a multi-model and generative approach to editor implementation, changes to the abstract syntax (i.e., the metamodel) invalidate instances (i.e., models), other editor models, generated code, and all model transformations that may depend on the aforementioned models. In previous work, the problem of metamodel/model co-evolution has been addressed [4,29], but the problem of co-evolution among coordinated editor models is largely unexplored. The present paper specifically contributes to this open problem, which helps with the co-evolution of DSML editors as opposed to the co-evolution of pre-existing DSML models. More specifically, we are concerned with the questions what and how GMF models need to be co-changed in reply to changes of the domain model (say, metamodel, or abstract syntax definition) of the editor. The co-evolution challenge at hand is to adapt GMF editors when changes are operated on the domain model.

The GMF framework does not support such co-evolution, and this somewhat diminishes the original goal of GMF to aggressively simplify the development of graphical editors. That is, while it is reasonable simple to draft and connect all GMF models from scratch, it is notably difficult to evolve an editor through changes of specific GMF models. A recurring focus for evolution is the domain model of the editor. When the domain model is changed, the user may notice that the editor has to be fixed through unsuccessful runs of some generator, the compiler, or the editor itself, and in all cases, subject to error messages at a low level of abstraction. Alternatively, the user may attempt to regenerate some models through the available wizards (model-to-model transformations of GMF), which however means that the original, possibly customized models are lost.

The complexity of the co-evolution problem for DSML editors has been recognized also by others (e.g., see [20,27]). Also, in [27], the authors discuss that the GMF infrastructure has a number of limitations, some of them related to co-evolution, and they even propose an alternative solution to define graphical editors for modeling languages. Even outside the MDE context, the identified co-evolution challenge is relevant, and it has not been generally addressed. For instance, in compiler construction and domain-specific language implementation for languages with textual syntax, the same kind of breaking changes to abstract or concrete syntaxes can happen, and not even the most advanced transformation- or mapping-based approaches to the coordination of the syntaxes readily address the challenge (e.g., see [18,30]). The challenge is only exacerbated by the multiplicity of coordinated models in an approach for DSML editors such as with GMF.

The contributions of the paper can be summarized as follows:

- We analyze GMF's characteristics in terms of the co-evolution of the various models that contribute to a GMF editor. Starting from conceived domain-model changes, their implications for the editor itself and other GMF models are identified.
- Even though catalogues for metamodel changes are available from multiple existing works (e.g., [4,28,29]—to mention a few), the application of such a change catalogue to a different scenario (i.e., the editor models in a "dependency" relation) is a novelty.
- We address the resulting co-evolution challenge by complementing GMF's wizard- and generator-biased architecture with GMF adapters, which are model-to-model transformations that propagate changes of the domain model to other models.
- The GMF adapters leverage on a difference model which is used to represent differences between subsequent versions of a given metamodel. Such difference models have been used in previous works on co-evolution, but the illustration of their applicability to the new kind of co-evolution challenge at hand is an important step towards their promotion to a general MDE technique.

We make available some reusable elements of our development publicly (scenarios, transformations, difference models, etc.)[1].

Road-map of the paper

In Sec. 2, we recall the basics of the GMF approach to graphical editor development, and we clarify GMF's use of a collection of coordinated editor models. In Sec. 3, we study a detailed evolution scenario to analyse the co-evolution challenge at hand. In Sec. 4, we develop an initial list of domain model changes and derive a methodology of co-evolution based on propagating changes to all relevant GMF models. In Sec. 5, we describe a principled approach for the automation of the required co-evolution transformation based on the interpretation of difference models for the domain-model changes. In Sec. 6, we sketch a proof-of-concept implementation that is also available online. Related work is described in Sec. 7, and the paper is concluded in Sec. 8.

[1] `http://www.emfmigrate.org/gmfevolution`

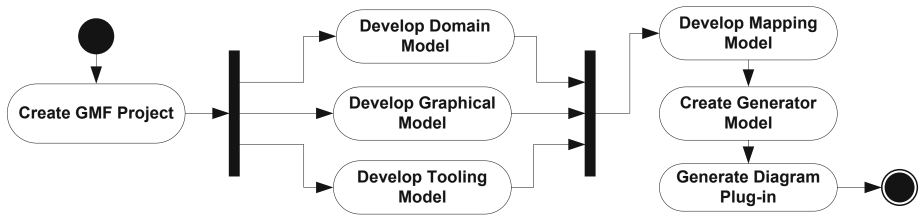


Fig. 2. The model-driven approach to GMF-based editor development

2 GMF's Coordinated Editor Models

GMF consists of a generative component (GMF Tooling) and runtime infrastructure (GMF Runtime) for developing graphical editors based on the Eclipse Modeling Framework (EMF) [9,3] and the Graphical Editing Framework (GEF) [12]. The GMF Tooling supports a model-driven process (see Fig. 2) for generating a fully functional graphical editor based on the GMF Runtime starting from the following models:

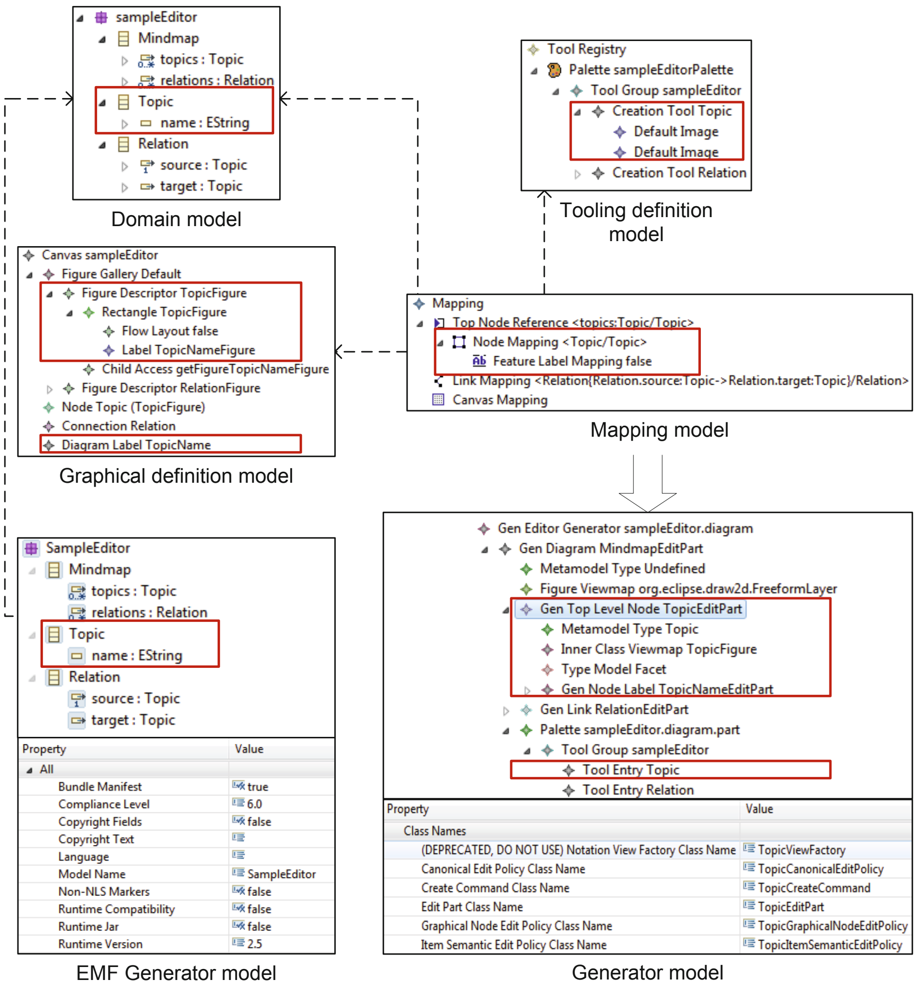
- The *domain model* is the Ecore-based metamodel (say, abstract syntax) of the language for which representation and editing have to be provided.
- The *graphical definition model* contains part of the concrete syntax; it identifies graphical elements that may, in fact, be reused for different editors.
- The *tooling definition model* contains another part of the concrete syntax; it contributes to palettes, menus, toolbars, and other periphery of the editor.
- Conceptually, the aforementioned models are reusable; they do not contain references to each other. It is the *mapping model* that establishes all links.

Consider again Fig. 1 which illustrates the role of these models for a simple mind-map editor.[2] Fig. 3 shows all the models involved in the definition and implementation of the mind-map editor. The rectangular boxes highlight contributions that are related to the *Topic* domain concept. It is worth noting how information about domain concepts is scattered over the various models. Most of these recurrences are not remote since most of the correspondences are name-based. Remote references are only to be found in the mapping and the generator models as clarified in the rest of the work.

Domain model. This model contains all the concepts and relationships which have to be implemented in the editor. In the example, the class *Mindmap* is introduced as a container of instances of the classes *Topic* and *Relation*.

Graphical definition model. This model specifies a figure gallery including shapes, labels, lines, etc., and canvas elements to define nodes, connections, compartments, and diagram labels. For instance, in the graphical model in Fig. 3, a rectangle named

[2] A mind map is a diagram used to represent words, ideas, tasks, or other items linked to and arranged around a central keyword or idea. The initial mind-map editor suffices with "topics" and "relations", but some extensions will be applied eventually.



Property	Value
All	
Bundle Manifest	true
Compliance Level	6.0
Copyright Fields	false
Copyright Text	
Language	
Model Name	SampleEditor
Non-NLS Markers	false
Runtime Compatibility	false
Runtime Jar	false
Runtime Version	2.5

Property	Value
Class Names	
(DEPRECATED, DO NOT USE) Notation View Factory Class Name	TopicViewFactory
Canonical Edit Policy Class Name	TopicCanonicalEditPolicy
Create Command Class Name	TopicCreateCommand
Edit Part Class Name	TopicEditPart
Graphical Node Edit Policy Class Name	TopicGraphicalNodeEditPolicy
Item Semantic Edit Policy Class Name	TopicItemSemanticEditPolicy

Fig. 3. The GMF models and model dependencies for the editor of Fig. 1

TopicFigure is defined, and it is referred to by the node *Topic*. A diagram label named *TopicName* is also defined. Such graphical elements will be used to specify the graphical representations for *Topic* instances and their *name*s.

Tooling definition model. This model defines toolbars and other periphery to facilitate the management of diagram content. In Fig. 3, the sample model consists of the *Topic* and *Relation* tools for creating the *Topic* and *Relation* elements.

Mapping model. This model links together the various models. For instance, according to the mapping model in Fig. 3, *Topic* elements are created by means of the *Creation*

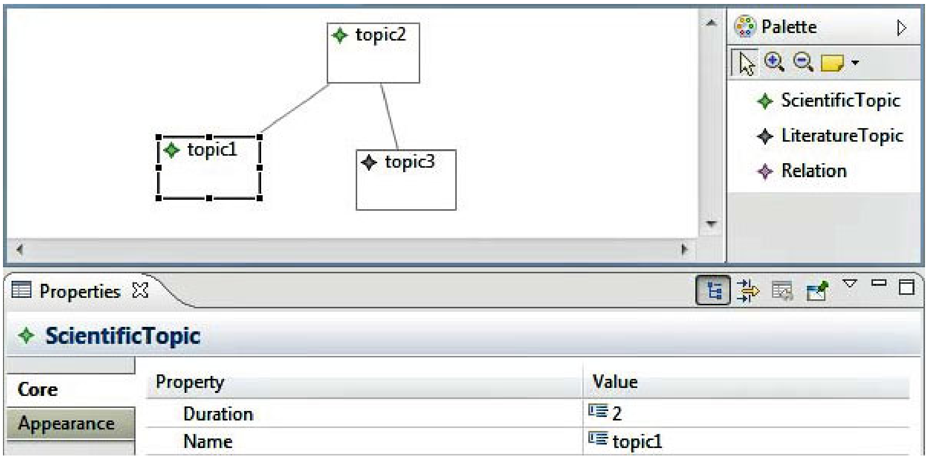
Tool Topic and the graphical representation for them is *Node Topic*. For each topic the corresponding *name* is also visualized because of the specified *Feature Label Mapping* which relates the attribute *name* of the class *Topic* with the diagram label *TopicName* defined in the graphical definition model. The meaning of the *false* value near the *Feature Label Mapping* element is that the attribute *name* is not read-only, thus it will be freely edited by the user.

EMF and GMF generator models. Once a domain model is defined, it is possible to automatically produce Java code to manage models (instances), say mind maps in our example. To this end an additional model, the *EMF generator model*, is required to control the execution of the EMF generator. A uniform version of the extra model can be generated by EMF tooling.

Once the mapping model is obtained, the GMF Tooling generates (by means of a model-to-model transformation) the *GMF generator model* that is used by a code generator to produce the real code of the modeled graphical editor.

3 GMF's Co-evolution Challenge

Using a compound change scenario, we will now demonstrate GMF's co-evolution challenges. We will describe how domain-model changes break other editor models, and the editor's code, or make them unsound otherwise. Hence, domain-model changes must be propagated. Such change propagation is not supported currently by GMF, and it is labor-intensive and error-prone, when carried out manually. Conceptually, it turns out to be difficult to precisely predict when and how co-changes must be performed.



Property	Value
Duration	2
Name	topic1

Fig. 4. An evolved mind-map editor with different kinds of topics

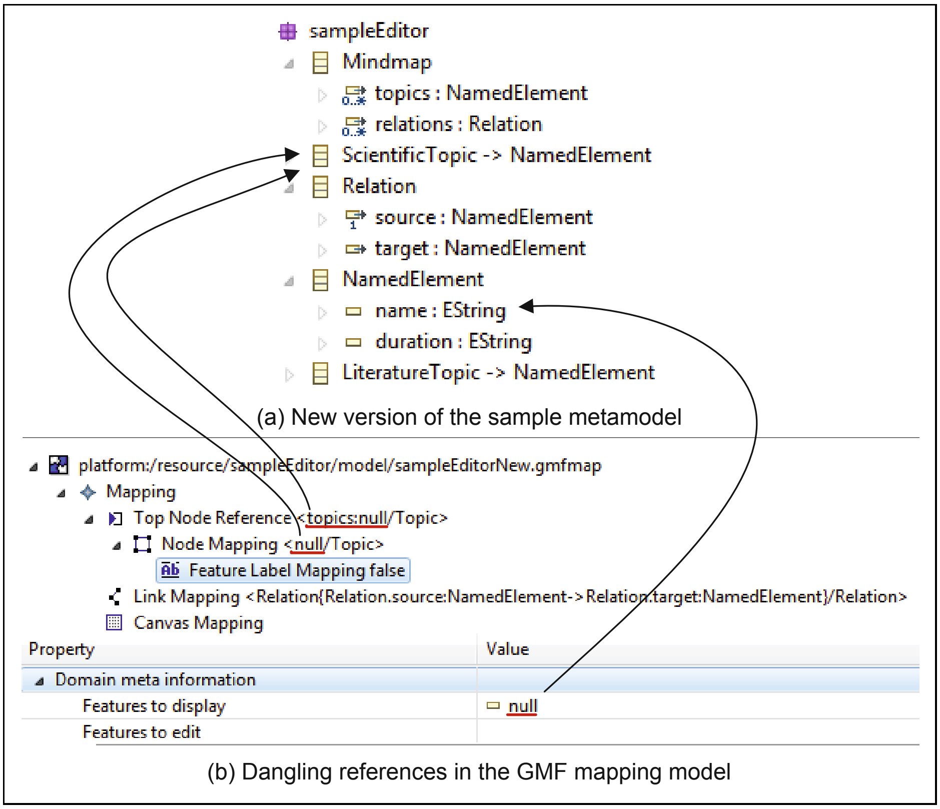

sampleEditor
Mindmap
topics : NamedElement
relations : Relation
ScientificTopic -> NamedElement
Relation
source : NamedElement
target : NamedElement
NamedElement
name : EString
duration : EString
LiteratureTopic -> NamedElement
(a) New version of the sample metamodel
platform:/resource/sampleEditor/model/sampleEditorNew.gmfmap
Mapping
Top Node Reference <topics:null/Topic>
Node Mapping <null/Topic>
Feature Label Mapping false
Link Mapping <Relation{Relation.source:NamedElement->Relation.target:NamedElement}/Relation>
Canvas Mapping
Property
Value
Domain meta information
Features to display
null
Features to edit
(b) Dangling references in the GMF mapping model

Fig. 5. The domain model for the evolved mind-map editor with the “broken” mapping model

3.1 A Compound Change Scenario

Consider the enhanced mind-map editor of Fig. 4. Compared to the initial version of Fig. 1, *scientific* vs. *literature topics* are distinguished, and topics have a *duration* property in addition to the *name* property.

Now consider Fig. 5; it shows the evolved metamodel at the top, and the status of the, as yet, unamended mapping model at the bottom. We actually show the mapping model as it would appear to the user if it was inspected in Eclipse. Some of the links in the mapping model are no longer valid; in fact, they are dangling (c.f., “null”). Through extra edges, we show what the links are supposed to be like.

We get a deeper insight into the situation if we comprehend the evolved domain model through a series of simple, potentially atomic changes:

1. The *Topic* class was renamed to *ScientificTopic*.
2. The abstract class *NamedElement* was added.
3. The attribute *name* is pulled up from the *Topic* class to the *NamedElement* class.
4. The attribute *duration* was added to the *NamedElement* class.
5. The class *LiteratureTopic* was added as a further specialization of *NamedElement*.

Table 1. Idiosyncratic symptoms of broken and and unsound GMF editors

1	The EMF generator or the GMF generator fails (with an error).
2	The EMF generator or the GMF generator completes with a warning.
3	The generator for the GMF generator model fails.
4	The compiler fails on the generated EMF or GMF code.
5	The editor plugin fails at runtime, e.g., at launch-time.
6	A GMF model editor reports an error upon opening a GMF model.
7	The editor plugin apparently executes, but misses concepts of the domain.
8	The editor plugin apparently executes, but there are GUI events without handlers.

3.2 Broken vs. Unsound GMF Models and Editors

In practice, these changes would have been carried out in an ad-hoc manner through editing directly the domain model. Because of these changes, the existing mapping model is no longer valid—as shown in Fig. 5. In particular, references to *Topic* or the attribute *name* thereof are dangling. The other GMF models are equally out-of-sync after these domain model changes. For clarity, in Table 1, we sketch a classification of the symptoms that may indicate a broken or unsound GMF editors. Due to space limitation we do not provide an explanation of the reported symptoms which are listed in Table 1 only for the sake of completeness.

Let us consider two specific examples. First, the addition of a new class to the domain model, e.g., *LiteratureTopic*, should probably imply a capability of the editor to create instances of the new class. However, such a creation tool would need to be added in the mapping and tooling models. Second, the renaming of a class, e.g., the renaming of *Topic* to *ScientificTopic*, may lead to an editor with certain functionality not having any effect because elements are referenced that changed or do not exist anymore in the domain model. Both examples are particularly interesting in so far that the editor apparently still works. i.e., it is not *broken* in a straightforward sense. However, we say that the editor is *unsound*; the editor does not meet some obvious expectations.

4 Changes and Co-changes

We will now describe a catalogue of domain-model changes and associated co-changes of other editor models. It turns out that there are different options for deciding on the impact of the co-changes. We capture those options by corresponding strategies. As far as the catalogue of changes is concerned, we can obviously depart from catalogues of metamodel changes as they are available in the literature, e.g., [29,15], and previous work by the authors [4]. For brevity's sake, we make a selection here. That is, we consider only atomic changes that are needed for the compound scenario of the previous section, completed by a few additional ones. Many of the missing changes would refer to technical aspects of the EMF implementation, and as such, they do not contribute to the discussion.

Table 2. Levels of editor soundness along evolution.

Level	Description
1	Unsound in the sense of being broken; there are reported issues (errors, warnings).
2	Unsound in the sense that the editor "obviously" lacks capabilities.
3	Sound as far as it can be achieved through automated transformations.
4	Sound; established by human evaluation.

4.1 Strategies for Co-changes

Such a distinction of *broken* vs. not broken but nevertheless *unsound* also naturally relates to a spectrum of *strategies* for co-changes. A *minimalistic strategy* would focus on fixing the broken editor. That is, co-changes are supposed to bring the editor models to a state where no issues are reported at generation, compile or runtime. A *best-effort strategy* would try to bring the editor to a sound state, or as close to it as possible with a general (perhaps automated) strategy.

Consider again the example of adding a new class *C*:

Minimalistic strategy. The execution of the EMF generator emits a warning, which we take to mean that the editor is broken. Hence, we would add the new class *C* to the EMF generator model. This small co-change would be sufficient to re-execute all generators without further errors or warnings, and to build and run the editor successfully. The editor would be agnostic to the new class though because the mapping and tooling models were not co-changed.

Best-effort strategy. Let us make further assumptions about the added class *C*. In fact, let us assume that *C* is a concrete class, and it has a superclass *S* with at least one existing concrete subclass *D*. In such a case, we may co-change the other GMF models by providing management for *C* based on the replication of the management for *D*.

Here we assume that a best-effort strategy may be amenable to an automated transformation approach in that it does not require any domain-specific insight. The modeler will still need to perform additional changes to complete the evolution, i.e., to obtain a sound editor.

4.2 Editor Soundness Related to Co-changes

In continuation of the soundness discussion from the previous section, Table 2 identifies different levels of soundness for an evolving editor. The idea here is that we assess the level of the editor *before* and *after* all (automated) co-changes were applied. The proposed transformations can never reach Level 4 because it requires genuine evaluation by the modeler. In other words, Level 4 refers to situations where GMF models can not be automatically migrated and they have to be adapted by the modeler in order to support all the modeling constructs defined in the new version of the considered metamodel.

However, we are not just interested in the overall level of the editor, but we also want to *blame* one or more editor models for the editor's unsoundness. In Table 3, we list

Table 3. Considered Ecore metamodel changes

	before co-change					after co-change				
	EMFGen	Graph	Tooling	Mapping	Level	EMFGen	Graph	Tooling	Mapping	Level
Add empty, concrete class	×	○	○	○	1	●	○	○	○	2
Add empty, abstract class	×	●	●	●	1	●	●	●	●	3
Add specialization	●	●	●	●	3	●	●	●	●	3
Delete concrete class	×	○	×	×	1	●	○	●	●	2
Rename class	×	○	○	×	1	●	●	●	●	3
Add property	×	○	○	○	1	●	○	○	●	2
Delete property	×	○	×	×	1	●	○	●	●	2
Rename property	×	○	○	×	1	●	●	●	●	3
Move property	×	○	×	×	1	●	○	○	○	2
Pull up property	×	○	×	×	1	●	○	●	●	2
Change property type	●	○	×	×	1	●	○	○	○	2

atomic changes with the soundness levels for the editor before and after co-changes, and all the indications as to what models are to blame. We use "×" to blame a model for causing the editor to be broken, i.e., to be at Level 1. We use "○" and "●" likewise for Level 2 and Level 3.

The EMFGen model is frequently to blame for a broken editor *before* the co-changes; the Graph model is never to blame for a broken editor; the remaining models are to blame occasionally for a broken editor. Obviously, there is trend towards less blame after the co-changes: no occurrences of "×", more occurrences of "●". In different terms, for all domain-model changes, all other models can be co-changed so that the editor is no longer broken. In several cases, we reach Level 3 for the editor.

There are clearly constellations for which changes cannot be propagated in an automated manner that resolves all Level 2 blame. For instance, the metamodel change *add empty, concrete class* does not require a co-changed Graph model as long as some existing graphical element can be reused. However, avoidance of Level 2 blame would require a manual designation of a new element or genuine selection of a specific element as opposed to an automatically chosen element.

4.3 Specific Couples of Changes and Co-changes

In the rest of the section, the changes reported in Table 3 and the corresponding co-changes, which have to be operated on the GMF models, are described in more detail.

Add empty, concrete class. Apart from the EMFGen model, the other ones are not affected; the editor simply does not take into account the added class. Thus, modelers cannot create or edit instances of the new class. The co-change may replicate the model from existing classes as discussed in Sec. 3. Ultimately, the modeler may need to manually complete the management of the new class.

Add empty, abstract class. In comparison to the previous case, the co-change of the EMFGen model is fully sufficient since abstract classes cannot be instantiated, and hence, no additional functionality is needed in the editor.

Add specialization. The change consists of modifying an existing class by specifying it as specialization of another one. In particular, in the simple case of the superclass being empty, this modification does not affect any model; thus, no co-changes are required.

Delete concrete class. Deleting an existing class is more problematic since all the GMF models except the Graph model are affected. Especially the Mapping model has to be fixed to solve possible dangling references to the deleted class. The Tooling model is also co-changed by removing the creation tool used to create instances of the deleted class. Even if the model is not adapted, the generator model and thus the editor can be generated—even though the palette will contain a useless tool without associated functionality. The Graph model can be left unchanged. The graphical elements which were used for representing instances of the deleted class, may be re-used in the future.

Rename class. Renaming a class requires co-change of the Mapping model which can have, as in the case of class deletion, invalid references which have to be fixed by considering the new class name. The Graph model does not require any co-change since the graphical elements used for the old class can be used even after the rename operation. The Tooling model can be left untouched, or alternatively the label and the description of the tool related to the renamed class can be modified to reflect the same name. However, even with the same Tooling model, a working editor will be generated.

Add property. The strategy for co-change is similar to the addition of new classes.

Delete property. Deleting a property which has a diagrammatic representation requires a co-change of the Mapping model in order to fix occurred dangling references. Moreover if some tools were available to manage the deleted property, also the Tooling model has to be co-changed. As in the case of class removals, the graphical model can be left unchanged.

Rename property. The strategy for co-change is similar to the renaming of classes.

Move property. When a property is moved from one class to another, then dangling references may need to be resolved in the Mapping model. If the moved property is managed by means of some tools, the Tooling model require co-changes, too. We only offer a simple, generic strategy for co-changes: the repaired editor does not consider the moved property.

Pull up property. Given a class hierarchy, a given property is moved from an extended to a base class. This modification is similar to the previous one—even though an automatic resolution can be provided to co-change Tooling and Mapping models in a satisfactory manner.

Change property type. The EMFGen model is not affected. However, by changing the type of a property some dangling references can occur in the Mapping model; their resolution cannot be fully automated. Also, if the affected property is managed by some tool, then the Tooling model must be co-changed as well.

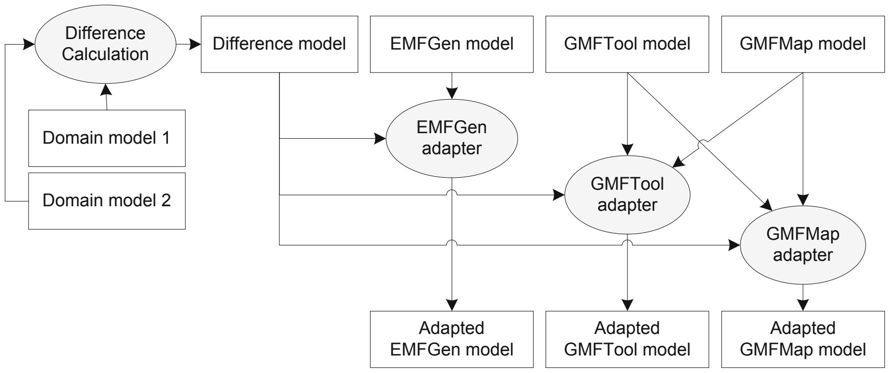


Fig. 6. Overview of the process of co-evolution with automated transformations

5 Automated Adaptation of GMF Models

Having a catalogue of changes like the one previously discussed is preparatory for supporting the adaptation of GMF models. In particular, it can be exploited to automatically detect the modifications that have been operated on a given domain model, and to instruct corresponding migration procedures as proposed in the rest of the paper.

We have developed a general process for GMF co-evolution which involves model differencing techniques and automated transformations to adapt existing GMF models with respect to the changes operated on domain models. The approach is described in Fig. 6 and consists of the following elements:

- *Difference calculation*, given two versions of the same *domain model*, their differences are calculated to identify the changes which have been operated on the first version of the model to obtain the last one. The calculation can be operated by any of the existing approaches able to detect the differences between any kind of models, like EMFCompare [7];
- *Difference representation*, the detected differences have to be represented in a way which is amenable to automatic manipulations and analysis. To take advantage of standard model driven technologies, the calculated differences should be represented by means of another model;
- *Generation of the adapted GMF models*, the differences represented in the difference model are taken as input by specific adapters each devoted to the adaptation of a given GMF model with respect to the metamodel modifications and corresponding co-changes reported in Table 3. In particular, the GMFMap and the GMFTool adapters are devoted to the adaptation of the *GMFMap model* and the *GMFTool model*, respectively. Such adapters take both models because of dependencies between them which have to be updated simultaneously. The *EMFGen model* is updated by means of a specific adapters, whereas no adapter is provided for the *Graph model*. In fact, the discussion of the previous sections suggested that we can always reasonable continue with the old *Graph model*. The adapters can be implemented as model transformations which take as input the old version of the GMF models and produce the adapted ones.

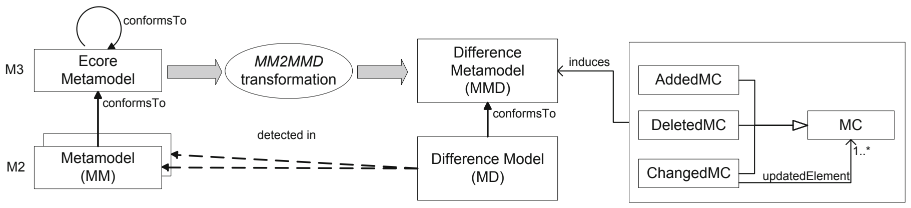


Fig. 7. Difference metamodel generation

Interestingly, the process in Fig. 6 is independent from the technologies which have been adopted both for calculating and managing domain model differences, and to automatically manipulate them for generating the adapted GMF models.

6 Proof-of-Concept Implementation of the GMF Adapters

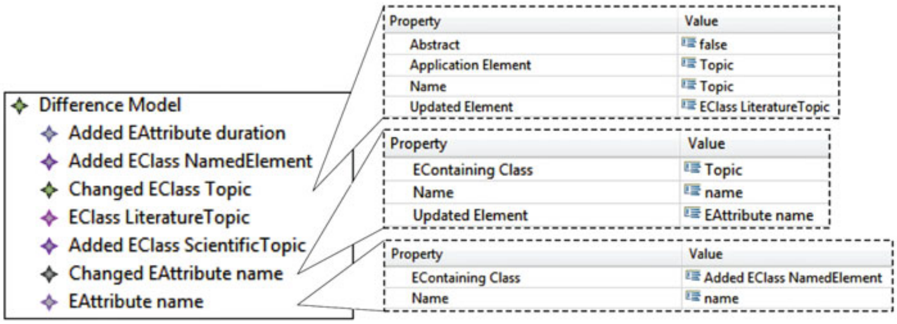
In this section we propose the support for the GMF model adaptation approach we described in the previous section. That is, in Sec. 6.1, we outline a technique for representing the differences between two versions of a same metamodel. Such a representation approach has been already used by the authors for managing other co-evolution problems [4]. Further, in Sec. 6.2, the ATL transformation language [17] is adopted for implementing the different model adapters which have been identified to evolve existing GMF models. The implementation of the approach is available publicly as described in the introduction of the paper.

6.1 Model-Based Representation of Domain Model Differences

The differences between different versions of a same domain model can be represented by exploiting the *difference metamodel* concept, presented by the authors in [5]. The approach is summarized in Fig. 7: given two Ecore metamodels, their difference conforms to a difference metamodel *MMD* derived from Ecore by means of the *MM2MMD* transformation. For each class *MC* of the Ecore metamodel, the additional classes *AddedMC*, *DeletedMC*, and *ChangedMC* are generated in the extended Ecore metamodel by enabling the representation of the possible modifications that can occur on domain models and that can be grouped as follows:

– *additions*, new elements are added in the initial metamodel;
– *deletions*, some of the existing elements are deleted;
– *changes*, some of the existing elements are updated.

In Fig. 8, a fragment of the difference model representing the changes between the domain models in Fig. 3 and Fig. 5 is shown. Such a difference model conforms to a difference metamodel automatically obtained from the ECore metamodel. For instance, from the metaclass *EClass* of the ECore metamodel, the metaclasses *AddedEClass*, *DeletedEClass*, and *ChangedEClass* are generated in the corresponding difference metamodel.


Difference Model
- Added EAttribute duration
- Added EClass NamedElement
- Changed EClass Topic
- EClass LiteratureTopic
- Added EClass ScientificTopic
- Changed EAttribute name
- EAttribute name

Property	Value
Abstract	false
Application Element	Topic
Name	Topic
Updated Element	EClass LiteratureTopic

Property	Value
EContaining Class	Topic
Name	name
Updated Element	EAttribute name

Property	Value
EContaining Class	Added EClass NamedElement
Name	name

Fig. 8. Fragment of the difference model for the evolution scenario of Sec. 3

For some of the reported differences in Fig. 8, the corresponding properties are shown. For instance, the renaming of the *Topic* class is represented by means of a *ChangedEClass* instance which has as updated element an instance of *EClass* named *LiteratureTopic* (see the *updatedElement* property of the changed class *Topic* shown on the right-hand side of Fig. 8). The addition of the class *NamedElement* is represented by means of an *AddedEClass* instance. The move operation of the attribute *name* from the class *Topic* to the added class *NamedElement* is represented by means of a *ChangedEAttribute* instance which has one *EAttribute* instance as updated element with a different value for the *eContainingClass* property. In fact, in the initial version it was *Topic* (see the second property window) whereas in the last one, it is *NamedElement* (as specified in the third property window).

6.2 ATL-Based Implementation of GMF Model Adapters

Our prototypical implementation of the GMF model adapters leverages ATL [17], a QVT [24] compliant language which contains a mixture of declarative and imperative constructs. In particular, each model adapter is implemented in terms of model transformations which use a common query library described in rest of the section.

An ATL transformation consists of a module specification containing a header section (e.g. lines 1-3 in Listing 1.1), transformation rules (lines 5-42 in Listing 1.1) and a number of helpers which are used to navigate models and to define complex calculations on them (some helpers which have been implemented are described in Table 4). In particular, the header specifies the source models, the corresponding metamodels, and the target ones; the helpers and the rules are the constructs used to specify the transformation behaviour.

Small excerpts of the GMFMap and GMFTool adapters are shown in Listing 1.1 and Listing 1.2, respectively. For instance, the *AddedSpecializationClassTo...* transformation rules manage new classes which have been added in the domain model as specializations of an existing one. The code excerpts involve the replication strategy that we have described in previous sections.

ATL transformation rules consist of source and target patterns: the former consist of source types and an OCL [25] guard stating the elements to be matched; the latter are

Table 4. Some helpers of the *gmfAdaptationLib*

Helper name	Context	Return type	Description
getEClassInNewMetamodel	EClass	EClass	Given a class of the old metamodel, it returns the corresponding one in the new metamodel.
getNewContainer	EAttribute	EClass	Given an EAttribute in the old metamodel, the corresponding container in the new one is retrieved. To this end, the helper checks if the EAttribute has been moved to a new added class, if not an existing class is returned.
isMoved	EAttribute	Boolean	It checks if the considered EAttribute has been moved to another container
isMovedToAddedEClass	EAttribute	Boolean	It checks if the considered EAttribute has been moved to a new added EClass.
isRenamed	EAttribute	Boolean	It checks if the given EAttribute has been renamed.

composed of a set of elements, each of them specifies a target type from the target metamodel and a set of bindings. A binding refers to a feature of the type, i.e. an attribute, a reference or an association end, and specifies an expression whose value initializes the feature. For instance, the *AddedSpecializationClassToNodeMapping* rule in Listing 1.1 is executed for each match of the source pattern in lines 8-17 which describes situations like the one we had in the sample scenario where the *LiteratureTopic* class (see *s1*) is added as specialization of an abstract class (see *s2*) which is specialized by another class (see *s3*). In this case, the Mapping model is updated by adding a new *TopNodeReference* and its contained elements (see lines 24-41) which are copies of those already existing for *s3*.

A similar source pattern is used in the rule of Listing 1.2 (lines 7-12) in order to add a creation tool for the new added class *s1* to the Tooling model (see lines 19-23).

```
 1 module GMFMapAdapter;
 2 create OUT : GMFMAPMM from IN : GMFMAPMM, GMFTOOL: GMFTOOLMM, DELTA: DELTAMM,
 3       NEWECORE : ECORE, OLDECORE : ECORE ;
 4 ...
 5 rule AddedSpecializationClassToNodeMapping {
 6
 7   from
 8    s1: DELTAMM!AddedEClass, s2: DELTAMM!AddedEClass,
 9    s3: DELTAMM!ChangedEClass, s4: DELTAMM!ChangedEAttribute,
10    s5: DELTAMM!EAttribute
11      ((not s1.abstract)
12       and s1.eSuperTypes->first() = s2
13       and s2.abstract
14       and s3.updatedElement->first().eSuperTypes->first() = s2
15       and s4.updatedElement->first() = s5
16       and s4.eContainingClass = s3
17       and s5.eContainingClass = s2 ))
18
19   using {
20     siblingFeatureLabelMapping : GMFMAPMM!FeatureLabelMapping =
            s3.getNodeMappingFromChangedClass().labelMappings
             ->select(e | e.oclIsTypeOf(GMFMAPMM!FeatureLabelMapping))->first()
                  ; }

  to
   t1 : GMFMAPMM!TopNodeReference (
      containmentFeature <- s3.getTopNodeReferenceFromChangedClass().
                containmentFeature.getFeatureInNewMetamodel(),
      ownedChild <- t2
    ),
   t2 : GMFMAPMM!NodeMapping (
      domainMetaElement <- s1.getAddedClassInNewMetamodel(),
      relatedDiagrams <- s3.getNodeMappingFromChangedClass().relatedDiagrams,
      tool <- s1.name.getNewToolFromTitle(),
      diagramNode <- s3.getNodeMappingFromChangedClass().diagramNode
    ),
   t3 : GMFMAPMM!FeatureLabelMapping (
      diagramLabel <- siblingFeatureLabelMapping.diagramLabel,
      features <- siblingFeatureLabelMapping.features->collect(e |
            e.getFeatureInNewMetamodel())
    ),
    ...
}
```

Listing 1.1. Fragment of the GMFMap Adapter

To summarize, the implementation of the GMF adapters consists of transformation rules which copy the given source model to a target one; during this operation they evaluate if changes are needed. A number of helpers have been defined; they navigate models and perform complex queries on them. Many of the helpers are common to all the adapters, and hence, they are available through a library *gmfAdaptationLib*. Table 4 describes some of these helpers.

```
module GMFToolAdapter;
create OUT : GMFTOOLMM from IN : GMFTOOLMM, GMFMAP : GMFMAPMM, DELTA: DELTAMM,
      NEWECORE : ECORE, OLDECORE : ECORE ;
...
rule AddedSpecializationClassToCreationTool {

  from
   s1: DELTAMM!AddedEClass, s2: DELTAMM!AddedEClass, s3: DELTAMM!ChangedEClass
    ( (not s1.abstract)
      and s1.eSuperTypes->first() = s2
      and s2.abstract
      and s3.updatedElement->first().eSuperTypes->first() = s2 )

  using {
   toolGroup : GMFTOOLMM!ToolGroup = OclUndefined;
  }

  to
   t : GMFTOOLMM!CreationTool (
     title <- s3.getToolFromChangedClass().title.regexReplaceAll(s3.
                getToolFromChangedClass().title, s1.name),
     description <- 'Create␣new␣' + s1.name
     ),
        ...
}
```

Listing 1.2. Sample transformation rule of the GMFTool Adapter

7 Related Work

7.1 Graphical Model Editors

In [1], a number of technologies for the development of domain-specific modeling languages (DSMLs) are evaluated; Eclipse (EMF with GEF) is covered, but not GMF. The evaluation criteria include language evolution to mean the ability to co-evolve models when the domain model changes. There is no criterion though that relates to GMF's particular characteristics of using multiple editor models.

Other GMF- or GEF-based frameworks have been proposed. For instance, the MuvitorKit framework [23] is based on EMF and GEF and specifically meant as an alternative to GMF for the benefit of additional editor capabilities (e.g., multiple panes) as well as additional modeling capabilities, thereby requiring less customization of generated code. There is also the EuGENia framework [20,19] which raises the level of abstraction in GMF-based development by using annotations on the domain model, thereby feeding into code generation. We are not aware of any prior effort to propagate changes across GMF models.

The ViatraDSM framework [27] replaces GMF in that it allows for versatile mappings between abstract and concrete syntax. Live transformations are leveraged to maintain the coherence of the two models. Our uni-directional, difference-driven transformations propagate domain-model changes elsewhere. Our work is specifically targeted at the mainstream GMF-based approach with its various models.

7.2 Model Consistency

The status of GMF models being out-of-sync can be compared to the notion of model inconsistency in (UML-based) modeling where different models providing different views may require synchronization. For instance, in [8], inconsistencies between the different diagrammatic forms in UML models are considered, and possible fixes are proposed in the form of value changes. In [13], the dependencies between models are modeled through triple graph grammars in a manner that enables incremental model synchronization. Our specific contribution is one of reverse engineering: discovering the GMF model dependencies, and making them operational through automated transformations.

7.3 Co-evolution of Metamodels and Models

The techniques and the methodology of our work are inspired by research on co-evolution in model-driven engineering [10,28]. Much of this work is concerned with co-transforming models in reply to metamodel changes [29,15]. In that case metamodel changes can invalid existing models that have to be adapted to recover the conformance with the new version of the metamodel.

In this work, we analyze another kind of co-evolution, even though related to the previous one, which aims at propagating metamodel changes to the other GMF models according to a given soundness level of the editor. The overall proposal leverages the difference representation approach proposed by the authors in [5] and already used to manage co-evolution problems in [6].

7.4 Syntax Relationships for Textual Languages

In [18,30,26], approaches for the operationalization of the link between concrete and abstract syntax definition are described. That is, concrete syntax definitions are customized into abstract syntax definitions. In fact, the approach of [18] is based on the idea that concrete and abstract syntax definitions can be incomplete but they automatically complete each other based on name mapping and other heuristics. In contrast, the approach of [30] is based on grammar transformations where the concrete syntax is mapped operationally to the abstract syntax. In [26], yet another approach is exercised, where the abstract syntax definition is associated with the concrete syntax definition through annotations. (A similar MDE approach is the one of TCS for KM3 [16].) None of these approaches provides any automated capabilities for change propagation. The classical approach to concrete-to-abstract syntax mappings is to use an attribute grammar. There are a number of approaches to align grammar transformations with attribution transformations, see, e.g., [21,22], but none of these approaches are directly applicable to the synchronization of abstract and concrete syntax. We contend that the problem of collections of coordinated GMF editor models seems to be even more complicated than concrete/abstract syntax synchronization.

8 Concluding Remarks

We have described the challenge of sound evolution for graphical editors based on model-driven development with GMF in particular, and we have addressed this challenge by a system of co-transformations that propagate changes from domain models to the other editor models.

We have identified a range of options for evolved editors to be unsound, and we have described corresponding resolution strategies. In the more established area of metamodel/model co-evolution, models either are not broken, or they are broken and can be reasonably resolved in an automated manner, or a well-understood problem-specific contribution to the resolution must be provided manually or through a heuristic. In the case of co-evolution for editor models, there is a scale of models being broken or unsound. Also, each of the various models calls for a designated analysis. Finally, there are intricate inter-model dependencies.

The existing GMF infrastructure is obviously rather complicated: it consists of a number of metamodels, libraries, generators, model transformations of industrial scale. We cannot claim to provide a full-fledged solution to the co-evolution challenge of GMF—this would require full coverage of Ecore, the metamodeling language of EMF, and full understanding of the implicit semantics of GMF model dependencies and tools.

The focus of this paper is on the conceptual co-evolution challenge at hand. The development of industrial-strength tools for co-evolution or the revision of the GMF suite is a clearly a major undertaking that is beyond the scope of this paper. The prototypical implementation of the proposed approach supports all the metamodel changes reported in Table 3. Nevertheless, we are confident that our transformational approach can be scaled incrementally over time to cover an increasing number of concrete evolution scenarios. In the future we plan to support them by providing additional effort in the implementation of the overall approach. The most critical omission in our methodology is that we do not currently cover co-evolution of custom code. This is a very intricate problem by itself, to which we hope to contribute through future work.

In our ongoing research, we try to better understand the co-evolution issues and associated strategies for the *code level* of GMF where generated code has been possibly customized. Based on preliminary research, we can already report that customization is used by some GMF projects extensively, and hence designated co-evolution support may provide significant help with real-world editor development.

References

1. Amyot, D., Farah, H., Roy, J.-F.: Evaluation of Development Tools for Domain-Specific Modeling Languages. In: Gotzhein, R., Reed, R. (eds.) SAM 2006. LNCS, vol. 4320, pp. 183–197. Springer, Heidelberg (2006)
2. Bézivin, J.: On the Unification Power of Models. Jour. on Software and Systems Modeling (SoSyM) 4(2), 171–188 (2005)
3. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.: Eclipse Modeling Framework. Addison-Wesley, Reading (2003)
4. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: Automating Co-evolution in Model-Driven Engineering. In: Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference, ECOC 2008, pp. 222–231. IEEE Computer Society, Los Alamitos (2008)
5. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: A Metamodel Independent Approach to Difference Representation. Journal of Object Technology 6(9), 165–185 (2007)
6. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: Model Patches in Model-Driven Engineering. In: Ghosh, S. (ed.) MODELS 2009. LNCS, vol. 6002, pp. 190–204. Springer, Heidelberg (2010)
7. Eclipse Foundation. EMF Compare (2010), `http://www.eclipse.org/modeling/emft/?project=compare`
8. Egyed, A., Letier, E., Finkelstein, A.: Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models. In: 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), pp. 99–108. IEEE, Los Alamitos (2008)
9. Eclipse project: Eclipse Modeling Framework Project (EMF), `http://www.eclipse.org/modeling/emf/`
10. Favre, J.-M.: Meta-Model and Model Co-evolution within the 3D Software Space. In: Proceedings of International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA 2003), Co-located with the IEEE International Conference on Software Maintenance, ICSM 2003 (2003)
11. Favre, J.-M.: Languages Evolve Too! Changing the Software Time Scale. In: IEEE (ed.) 8th Interntational Workshop on Principles of Software Evolution, IWPSE (2005)
12. Eclipse project: GEF - Graphical Editing Framework, `http://www.eclipse.org/gef/`
13. Giese, H., Wagner, R.: Incremental Model Synchronization with Triple Graph Grammars. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 543–557. Springer, Heidelberg (2006)
14. Eclipse project: GMF - Graphical Modeling Framework, `http://www.eclipse.org/gmf/`
15. Herrmannsdoerfer, M., Benz, S., Jürgens, E.: COPE - Automating Coupled Evolution of Metamodels and Models. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 52–76. Springer, Heidelberg (2009)

16. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: Proceedings of Generative Programming and Component Engineering (GPCE 2006), pp. 249–254. ACM, New York (2006)
17. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
18. Kadhim, B., Waite, W.: Maptool—supporting modular syntax development. In: Gyimóthy, T. (ed.) CC 1996. LNCS, vol. 1060, pp. 268–280. Springer, Heidelberg (1996)
19. Kolovos, D.S., Rose, L.M., Abid, S.B., Paige, R.F., Botterweck, G.: Taming EMF and GMF Using Model Transformation. In: 13th ACM/IEEE International Conference on Model Driven Engineering, Languages and Systems, MoDELS (to appear, 2010)
20. Kolovos, D.S., Rose, L.M., Paige, R.F., Polack, F.A.C.: Raising the level of abstraction in the development of GMF-based graphical model editors. In: MISE 2009: Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering, pp. 13–19. IEEE, Los Alamitos (2009)
21. Lämmel, R., Riedewald, G.: Reconstruction of paradigm shifts. In: Proceedings of the Second Workshop on Attribute Grammars and their Applications (WAGA 1999), pp. 37–56 (March 1999); INRIA Technical Report ISBN 2-7261-1138-6
22. Lohmann, W., Riedewald, G., Stoy, M.: Semantics-preserving migration of semantic rules after left recursion removal in attribute grammars. In: Proceedings of 4th Workshop on Language Descriptions, Tools and Applications (LDTA 2004). ENTCS, vol. 110, pp. 133–148. Elsevier Science, Amsterdam (2004)
23. Modica, T., Biermann, E., Ermel, C.: An Eclipse Framework for Rapid Development of Rich-featured GEF Editors based on EMF Models. In: Proceedings of Informatik 2009: Im Focus das Leben, Beiträge der 39. Jahrestagung der Gesellschaft füur Informatik e.V (GI). LNI, vol. 154, pp. 2972–2985. GI (2009)
24. Object Management Group (OMG). MOF QVT Final Adopted Specification, OMG Adopted Specification ptc/05-11-01 (2005)
25. Object Management Group (OMG). OCL 2.0 Specification, OMG Document formal/2006-05-01 (2006)
26. Overbey, J.L., Johnson, R.E.: Generating Rewritable Abstract Syntax Trees. In: Gašević, D., Lämmel, R., Van Wyk, E. (eds.) SLE 2008. LNCS, vol. 5452, pp. 114–133. Springer, Heidelberg (2009)
27. Ráth, I., Ökrös, A., Varró, D.: Synchronization of abstract and concrete syntax in domain-specific modeling languages—By mapping models and live transformations. Journal of Software and Systems Modeling (2009)
28. Vermolen, S., Visser, E.: Heterogeneous Coupled Evolution of Software Languages. In: Busch, C., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 630–644. Springer, Heidelberg (2008)
29. Wachsmuth, G.: Metamodel Adaptation and Model Co-adaptation. In: Bateni, M. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 600–624. Springer, Heidelberg (2007)
30. Wile, D.: Abstract syntax from concrete syntax. In: Proceedings, International Conference on Software Engineering (ICSE 1997), pp. 472–480. ACM Press, New York (1997)

An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models

Markus Herrmannsdoerfer[1], Sander D. Vermolen[2], and Guido Wachsmuth[2]

[1] Institut für Informatik,
Technische Universität München
Boltzmannstr. 3, 85748 Garching b. München, Germany
herrmama@in.tum.de
[2] Software Engineering Research Group,
Delft University of Technology
The Netherlands
{S.D.Vermolen,G.H.Wachsmuth}@tudelft.nl

Abstract. Modeling languages and thus their metamodels are subject to change. When a metamodel is evolved, existing models may no longer conform to it. Manual migration of these models in response to metamodel evolution is tedious and error-prone. To significantly automate model migration, operator-based approaches provide reusable coupled operators that encapsulate both metamodel evolution and model migration. The success of an operator-based approach highly depends on the library of reusable coupled operators it provides. In this paper, we thus present an extensive catalog of coupled operators that is based both on a literature survey as well as real-life case studies. The catalog is organized according to a number of criteria to ease assessing the impact on models as well as selecting the right operator for a metamodel change at hand.

1 Introduction

Like software, modeling languages are subject to evolution due to changing requirements and technological progress [9]. A modeling language is adapted to the changed requirements by evolving its metamodel. Due to *metamodel evolution*, existing models may no longer conform to the evolved metamodel and thus need to be migrated to reestablish conformance to the evolved metamodel. Avoiding *model migration* by downwards-compatible metamodel changes is often a poor solution, since it reduces the quality of the metamodel and thus the modeling language [5]. Manual migration of models is tedious and error-prone, and hence model migration needs to be automated. In *coupled evolution* of metamodels and models, the association of a model migration to a metamodel evolution is managed automatically. There are two major coupled evolution approaches: difference-based and operator-based approaches.

Difference-based approaches use a declarative evolution specification, generally referred to as difference model [6,11]. The difference model is mapped onto a model migration, which may be specified declaratively as well as imperatively.

Operator-based approaches specify metamodel evolution by a sequence of operator applications [24,13]. Each operator application can be coupled to a model migration separately. Operator-based approaches generally provide a set of reusable coupled operators which work at the metamodel level as well as at the model level. At the metamodel level, a coupled operator defines a metamodel transformation capturing a common evolution. At the model level, a coupled operator defines a model transformation capturing the corresponding migration. Application of a coupled operator to a metamodel and a conforming model preserves model conformance.

In both operator-based and difference-based approaches, evolution can be specified manually [24], can be recorded [13], or can be detected automatically [6,11]. When recording, the user is restricted to a recording editor. Using automated detection, the building process can be completely automated, but may lead to an incorrect model migration.

In this paper, we follow an operator-based approach to automate building a model migration for EMOF-like metamodels [18]. The success of an operator-based approach highly depends on the library of reusable coupled operators it provides [20]. The library of an operator-based approach needs to fulfill a number of requirements. A library should seek completeness so as to be able to cover a large set of evolution scenarios. However, the higher the number of coupled operators, the more difficult it is to find a coupled operator in the library. Consequently, a library should also be organized in a way that it is easy to select the right coupled operator for the change at hand.

To provide guidance for building a library, we present an extensive catalog of coupled operators in this paper. To ensure completeness, the coupled operators in this catalog are either motivated from the literature or from case studies that we performed. However, we do not target theoretical completeness—to capture all possible migrations—but rather practical completeness—to capture migrations that likely happen in practice. To ease usability, the catalog is organized according to a number of criteria. The criteria do not only allow to select the right coupled operator from the catalog, but also to assess the impact of the coupled operator on the modeling language and its models. For difference-based approaches, the catalog serves as a set of composite changes that such an approach needs to be able to handle.

The paper is structured as follows: Section 2 presents the EMOF-like metamodeling formalism on which the coupled operators are based. Section 3 introduces the papers and case studies from which the coupled operators originate. Section 4 defines different classification criteria for coupled operators. Section 5 lists and groups the coupled operators of the catalog. Section 6 discusses the catalog, and Section 7 concludes the paper.

2 Metamodeling Formalism

Metamodels can be expressed in various metamodeling formalisms. Well-known examples are the Meta Object Facility (MOF) [18], the metamodeling standard proposed by the Object Management Group (OMG) and Ecore [21], the metamodeling formalism underlying the Eclipse Modeling Framework (EMF). In this paper, we focus only on the core metamodeling constructs that are interesting for coupled evolution of metamodels and models. We leave out annotations, derived features, and operations, since these cannot be instantiated in models. An operator catalog will need additional operators addressing these metamodeling constructs in order to reach full compatibility with Ecore or MOF.

Metamodel. Figure 1 gives a textual definition of the metamodeling formalism used in this paper. A metamodel is organized into *Package*s which can themselves be composed of *sub packages*. Each package defines a number of *Type*s which can be either primitive (*PrimitiveType*) or complex (*Class*). Primitive types are either *DataType*s like Boolean, Integer and String or *Enumeration*s of *literals*. Classes consist of a number of *features*. They can have *super types* to inherit features and might be *abstract*, i.e. are not allowed to have objects. The *name* of a feature needs to be unique among all features of a class, including inherited ones. A *Feature* has a multiplicity (*lower bound* and *upper bound*) and is either an *Attribute* or a *Reference*. An attribute is a feature with a primitive *type*, whereas a reference is a feature with a complex *type*. An attribute can serve as an *id*entifier for objects of a class, i.e. the values of this attribute must be unique among all objects. A reference may be *composite* and two references can be combined to form a bidirectional association by making them *opposite* of each other.

```
abstract class NamedElement {
    name        :: String (1..1)
}

class Package : NamedElement {
    subPackages <> Package (0..*)
    types       <> Type (0..*)
}

abstract class Type : NamedElement {}

abstract class PrimitiveType : Type {}

class DataType : PrimitiveType {}

class Enumeration : PrimitiveType {
    literals    <> Literal (0..*)
}

class Literal : NamedElement {}
```

```
class Class : Type {
    isAbstract  :: Boolean
    superTypes  -> Class (0..*)
    features    <> Feature (0..*)
}

abstract class Feature : NamedElement {
    lowerBound  :: Integer
    upperBound  :: Integer
    type        -> Type
}

class Attribute : Feature {
    isId        :: Boolean
}

class Reference : Feature {
    isComposite :: Boolean
    opposite    -> Reference
}
```

Fig. 1. Metamodeling formalism providing core metamodeling concepts

Model. At the model level, instances of classes are called *objects*, instances of primitive data types are called *values*, instances of features are called *slots*, and instances of references are called *links*. The set of all links of composite references forms a containment structure, which needs to be tree-shaped and span all objects in a model.

Notational Conventions. Throughout the paper, we use the textual notation from Figure 1 for metamodels. In this notation, features are represented by their name followed by a separator, their type, and an optional multiplicity. The separator indicates the kind of a feature. We use `::` for attributes, `->` for ordinary references, and `<>` for composite references.

3 Origins of Coupled Operators

The coupled operators are either motivated from the literature or from case studies that we performed.

Literature. First, coupled operators originate from the literature on the evolution of metamodels as well as object-oriented database schemas and code.

Wachsmuth first proposes an operator-based approach for *metamodel* evolution and classifies a set of operators according to the preservation of metamodel expressiveness and existing models [24]. Gruschko et al. envision a difference-based approach and therefore classify all primitive changes according to their impact on existing models [2,4]. Cicchetti et al. list a set of composite changes which they are able to detect using their difference-based approach [6].

Banerjee et al. present a complete and sound set of primitives for schema evolution in the *object-oriented database* system ORION and characterize the primitives according to their impact on existing databases [1]. Brèche introduces a set of high-level operators for schema evolution in the object-oriented system O_2 and shows how to implement them in terms of primitive operators [3]. Pons and Keller propose a three-level catalog of operators for object-oriented schema evolution which groups operators according to their complexity [19]. Claypool et al. list a number of primitives for the adaptation of relationships in object-oriented systems [7].

Fowler presents a catalog of operators for the refactoring of *object-oriented code* [10]. Dig and Johnson show—by performing a case study—that most changes on object-oriented code can be captured by a rather small set of refactoring operators [8].

Case Studies. Second, coupled operators originate from a number of case studies that we have performed. Table 1 gives an overview over these case studies. It mentions the tool that was used in a case study, the name of the evolving metamodel, an abbreviation for the case study which we use in other tables throughout the paper, and whether the evolution was obtained in a forward or reverse engineering process. In forward engineering, the tool is used to aid and possibly record evolution as it happens, whereas in reverse engineering, the tool is used to reconstruct evolution after it occurred. To provide evidence that the case studies are considerable in size, the table also shows the number of different kinds of metamodel elements at the end of the evolution as well as the number of operator applications needed to perform the evolution.

Table 1. Statistics for case studies

Tool	Abbreviation	Name	Kind	Packages	Classes	Attributes	References	Data Types	Enumerations	Literals	Operator Applications
[12]	F	FLUID	reverse	8	155	95	155	0	1	10	223
	T	TAF-Gen		15	97	81	114	1	13	76	134
COPE	[13]	PCM	reverse	19	99	18	135	0	4	19	101
	[14]	GMF		4	252	379	278	0	27	166	737
	U	Unicase	forward	17	77	88	161	0	11	49	58
	Q	Quamoco		1	22	14	35	0	1	2	423
Acoda	B	BugZilla	reverse	–	51	208	64	–	–	–	237
	R	Researchr	forward	–	125	380	278	–	6	31	64
	Y	YellowGrass		–	12	33	21	–	0	0	30

Herrmannsdoerfer et al. performed a case study on the evolution of two industrial metamodels to show that most of the changes can be captured by reusable coupled operators [12]: Flexible User Interface Development (FLUID) for the specification of automotive user interfaces and Test Automation Framework - Generator (TAF-Gen) for the generation of test cases for these user interfaces.

Based on the requirements derived from this study, Herrmannsdoerfer implemented the operator-based tool *COPE*[1] [13] which records operator histories on metamodels of the Eclipse Modeling Framework (EMF). To demonstrate its applicability, COPE has been used to reverse engineer the operator history of a number of metamodels: Palladio Component Model (PCM) for the specification of software architectures [13] and Graphical Modeling Framework (GMF) for the model-based development of diagram editors [14]. Currently, COPE is applied to forward engineer the operator history of a number of metamodels: Unicase[2] for UML modeling and project management and Quamoco[3] for modeling the quality of software products.

Vermolen implemented the operator-based tool *Acoda*[4] [22] which detects operator histories on object-oriented data models. To demonstrate its applicability, Acoda has been used to reverse engineer the operator history of the data model behind BugZilla which is a well-known tool for bug tracking. Currently, Acoda is applied to forward engineer the operator-based evolution of a number of data models: Researchr[5] for maintaining scientific publications and YellowGrass[6] for tag-based issue tracking. The crossed-out cells in Table 1 indicate

[1] COPE web site, `http://cope.in.tum.de`

[2] Unicase web site, `http://unicase.org`

[3] Quamoco web site, `http://www.quamoco.de`

[4] Acoda web site, `http://swerl.tudelft.nl/bin/view/Acoda`

[5] Researchr web site, `http://researchr.org`

[6] YellowGrass web site, `http://yellowgrass.org`

that the metamodeling constructs are currently not supported by the used data modeling formalism.

4 Classification of Coupled Operators

Coupled operators can be classified according to several properties. We are interested in language preservation, model preservation, and bidirectionality. Therefore, we stick to a simplified version of the terminology from [24].

Language Preservation. A metamodel is an intensional definition of a language. Its extension is a set of conforming models. When an operator is applied to a metamodel, this has an impact on its extension and thus on the expressiveness of the language. We distinguish different classes of operators according to this impact [15,24]: An operator is a *refactoring* if there exists always a bijective mapping between extensions of the original and the evolved metamodel. An operator is a *constructor* if there exists always an injective mapping from the extension of the original metamodel to the extension of the evolved metamodel. An operator is a *destructor* if there exists always a surjective mapping from the extension of the original metamodel to the extension of the evolved metamodel.

Model Preservation. Model preservation properties indicate when migration is needed. An operator is *model-preserving* if all models conforming to an original metamodel also conform to the evolved metamodel. Thus, model-preserving operators do not require migration. An operator is *model-migrating* if models conforming to an original metamodel might need to be migrated in order to conform to the evolved metamodel. It is *safely model-migrating* if the migration preserves distinguishability, i.e. different models (conforming to the original metamodel) are migrated to different models (conforming to the evolved metamodel). In contrast, an *unsafely model-migrating* operator might yield the same model when migrating two different models.

Classification of operators w.r.t. model preservation is related to the classification w.r.t. language preservation: Refactorings and constructors are either model-preserving or safely model-migrating operators. Destructors are unsafely model-migrating operators. Furthermore, the classification is related to a classification of changes known from difference-based approaches [2,4]: model-preserving operators perform *non-breaking changes*, whereas model-migrating operators perform *breaking, resolvable changes*. However, there is no correspondence for *breaking, non-resolvable changes*, since coupled operators always provide a migration to resolve the breaking change.

Bidirectionality. Another property we are interested in is the reversibility of evolution. Bidirectionality properties indicate that an operator can be safely undone on the language or model level. An operator is *self-inverse* iff a second application of the operator—possibly with other parameters—always yields the original metamodel. An operator is the *inverse* of another operator iff there is always a sequential composition of both operators which does not change the metamodel. Finally, an operator is a *safe inverse* of another operator iff it is an inverse and there is always a sequential composition of both operators which is model-preserving.

5 Catalog of Coupled Operators

In this section, we present a catalog of 61 coupled operators that we consider complete for practical application. As discussed in Section 3, we included all coupled operators found in nine related papers as well as all coupled operators identified by performing nine real-life case studies. In the following, we explain the coupled operators in groups which help users to navigate the catalog. We start with *primitive* operators which perform an atomic metamodel evolution step that can not be further subdivided. Here, we distinguish *structural* primitives which create and delete metamodel elements and *non-structural* primitives which modify existing metamodel elements. Afterwards, we continue with *complex* operators. These can be decomposed into a sequence of primitive operators which has the same effect at the metamodel level but not neccessarily at the model level. We group complex operators according to the metamodeling techniques they address—distinguishing specialization and generalization, delegation, and inheritance operators—as well as their semantics—distinguishing replacement, and merge and split operators.

Each group is discussed separately in the subsequent sections. For each group, a table provides an overview over all operators in the group. Using the classifications from Section 4, the table classifies each coupled operator according to language preservation (`L`) into refactoring (`r`), constructor (`c`) and destructor (`d`) as well as according to model preservation (`M`) into model-preserving (`p`), safely (`s`) and unsafely (`u`) model-migrating. The table further indicates the safe (`s`) and unsafe (`u`) inverse (`I`) of each operator by referring to its number. Finally, each paper and case study has a column in each table. An `x` in such a column denotes occurrence of the operator in the corresponding paper or case study. Papers are referred to by citation, while case studies are referred to by the abbreviation given in Table 1. For each coupled operator, we discuss its semantics in terms of metamodel evolution and model migration.

5.1 Structural Primitives

Structural primitive operators modify the structure of a metamodel, i.e. create or delete metamodel elements. Creation operators are parameterized by the specification of a new metamodel element, and deletion operators by an existing metamodel element.

		Class.			MM			OODB				OOC		[12]		COPE				Acoda		
#	Operator Name	L	M	I	[24]	[2]	[6]	[1]	[3]	[19]	[7]	[10]	[8]	F	T	[13]	[14]	U	Q	B	R	Y
1	Create Package	r	p	2s		x									x	x						
2	Delete Package	r	p	1s		x									x							
3	Create Class	c	p	4s	x	x	x	x						x	x	x	x	x	x	x	x	x
4	Delete Class	d	u	3u	x	x	x	x	x				x	x		x		x	x	x	x	
5	Create Attribute	c	s	7s	x	x	x	x						x	x	x	x	x	x	x	x	x
6	Create Reference	c	s	7s	x	x	x	x						x	x	x	x	x	x	x	x	x
7	Delete Feature	d	u	5/6u	x	x	x	x						x	x	x	x	x	x	x	x	
8	Create Opposite Ref.	d	u	9u		x					x	x					x	x	x		x	x
9	Delete Opposite Ref.	c	p	8s		x					x	x			x							
10	Create Data Type	r	p	11s		x																
11	Delete Data Type	r	p	10s		x													x			
12	Create Enum	r	p	13s		x									x	x	x	x	x			
13	Delete Enum	r	p	11s		x												x				
14	Create Literal	c	p	15s		x												x				
15	Merge Literal	d	u	14u		x											x					

Creation of non-mandatory metamodel elements (packages, classes, optional features, enumerations, literals and data types) is model-preserving. Creation of mandatory features is safely model-migrating. It requires initialization of slots using default values or default value computations.

Deleting metamodel elements, such as classes and references, requires deleting instantiating model elements, such as objects and links, by the migration. However, deletion of model elements poses the risk of migration to inconsistent models: For example, deletion of objects may cause links to non-existent objects and deletion of references may break object containment. Therefore, deletion operators are bound to metamodel level restrictions: Packages may only be deleted when they are empty. Classes may only be deleted when they are outside inheritance hierarchies and are targeted neither by non-composite references nor by mandatory composite references. Several complex operators discussed in subsequent sections can deal with classes not meeting these requirements. References may only be deleted when they are neither composite, nor have an opposite. Enumerations and data types may only be deleted when they are not used in the metamodel and thus obsolete.

Deletion operators which may have been instantiated in the model (with the exception of *Delete Opposite Reference*) are unsafely model-migrating due to loss of information. Deletion provides a safe inverse to its associated creation operator. Since deletion of metamodel elements which may have been instantiated in a model is unsafely model-migrating, creation of such elements provides an unsafe inverse to deletion: Lost information cannot be restored.

Creating and deleting references which have an opposite are different from other creation and deletion operators. *Create Opposite Reference* restricts the set of valid links and is thus an unsafely model-migrating destructor, whereas *Delete Opposite Reference* removes a constraint from the model and is thus a model-preserving constructor.

Create / Delete Data Type and *Create / Delete Enumeration* are refactorings, as restrictions on these operators prevent usage of created or deleted elements. Deleting enumerations and data types is thus model-preserving. *Merge Literal* deletes a literal and replaces its occurrences in a model by another literal. Merging a literal provides a safe inverse to *Create Literal*.

5.2 Non-structural Primitives

Non-structural primitive operators modify a single, existing metamodel element, i.e. change properties of a metamodel element. All non-structural operators take the affected metamodel element, their subject, as parameter.

		Class.			MM			OODB				OOC		[12]		COPE				Acoda		
#	Operator Name	L	M	I	[24]	[2]	[6]	[1]	[3]	[19]	[7]	[10]	[8]	F	T	[13]	[14]	U	Q	B	R	Y
1	Rename	r	s	1s	x	x	x	x				x	x	x	x	x	x	x	x	x	x	x
2	Change Package	r	s	2s		x								x	x	x		x				
3	Make Class Abstract	d	u	4u		x								x		x			x			
4	Drop Class Abstract	c	p	3s		x											x		x			
5	Add Super Type	c	p	6s		x		x							x	x	x	x	x		x	x
6	Remove Super Type	d	u	5u		x		x							x	x	x	x	x			
7	Make Attr. Identifier	d	u	8u		x								x							x	
8	Drop Attr. Identifier	c	p	7s		x								x			x					
9	Make Ref. Composite	d	u	10u		x						x		x	x		x		x			
10	Switch Ref. Composite	c	s	9s		x		x				x		x	x				x			
11	Make Ref. Opposite	d	u	12u		x					x				x						x	x
12	Drop Ref. Opposite	c	p	11s		x					x						x		x			

Change Package can be applied to both package and type. Additionally, the value-changing operators *Rename*, *Change Package* and *Change Attribute Type* are parameterized by a new value. *Make Class Abstract* requires a subclass parameter indicating to which class objects need to be migrated. *Switch Reference Composite* requires an existing composite reference as target.

Packages, types, features and literals can be renamed. *Rename* is safely model-migrating and finds a self-inverse in giving a subject its original name back. *Change Package* changes the parent package of a package or type. Like renaming, it is safely model-migrating and a safe self-inverse.

Classes can be made abstract, requiring migration of objects to a subclass, because otherwise, links targeting the objects may have to be removed. Consequently, mandatory features that are not available in the super class have to be initialized to default values. *Make Class Abstract* is unsafely model-migrating, due to loss of type information and has an unsafe inverse in *Drop Class Abstract*. Super type declarations may become obsolete and may need to be removed. *Remove Super Type* `S` from a class `C` implies removing slots of features inherited from `S`. Additionally, references targeting type `S`, referring to objects of type `C`, need to be removed. To prevent breaking multiplicity restrictions, *Remove Super Type* is restricted to types `S` which are not targeted by mandatory references—neither directly, nor through inheritance. The operator is unsafely model-migrating and can be unsafely inverted by *Add Super Type*.

Attributes defined as identifier need to be unique. *Make Attribute Identifier* requires a migration which ensures uniqueness of the attribute's values and is thus unsafely model-migrating. *Drop Attribute Identifier* is model-preserving and does not require migration.

References can have an opposite and can be composite. An opposite reference declaration defines the inverse of the declaring reference. References combined with a multiplicity restriction on the opposite reference restrict the set of valid links. *Make Reference Opposite* needs a migration to make the link set satisfy the added multiplicity restriction. The operator is thereby unsafely model-migrating. *Drop Reference Opposite* removes cardinality constraints from the link set and does not require migration, thus being model-preserving.

Make Reference Composite ensures containment of referred objects. Since all referred objects were already contained by another composite reference, all objects must be copied. To ensure the containment restriction, copying has to be recursive across composite references (deep copy). Furthermore, to prevent cardinality failures on opposite references, there may be no opposite references to any of the types of which objects are subject to deep copying. *Switch Reference Composite* changes the containment of objects to an existing composite reference. If objects of a class `A` were originally contained in class `B` through composite reference `b`, *Switch Reference Composite* changes containment of `A` objects to class `C`, when it is parameterized by reference `b` and a composite reference `c` in class `C`. After applying the operator, reference `b` is no longer composite. *Switch Reference Composite* provides an unsafe inverse to *Make Reference Composite*.

5.3 Specialization / Generalization Operators

Specializing a metamodel element reduces the set of possible models, whereas generalizing expands the set of possible models. Generalization and specialization can be applied to features and super type declarations. All specialization and generalization operators take two parameters: a subject and a generalization or specialization target. The first is a metamodel element and the latter is a class or a multiplicity (lower and upper bound).

		Class.			MM			OODB				OOC		[12]		COPE				Acoda		
#	Operator Name	L	M	I	[24]	[2]	[6]	[1]	[3]	[19]	[7]	[10]	[8]	F	T	[13]	[14]	U	Q	B	R	Y
1	Generalize Attribute	c	p	2s	x	x	x							x	x	x	x		x	x	x	x
2	Specialize Attribute	d	u	1u	x	x	x							x		x	x		x			
3	Generalize Reference	c	p	4s	x	x	x							x	x		x		x		x	x
4	Specialize Reference	d	u	3u	x	x	x							x	x	x	x		x			
5	Specialize Composite Ref.	d	u	3u										x		x			x			
6	Generalize Super Type	d	u	7u		x										x					x	
7	Specialize Super Type	c	s	6s		x				x				x	x	x	x		x			

Generalization of features does not only generalize the feature itself, but also generalizes the metamodel as a whole. Feature generalizations are thus model-preserving constructors. Generalizing a super type declaration may require removal of feature slots and is only unsafely model-migrating. Feature specialization is a safe inverse of feature generalization. Due to the unsafe nature of the migration resulting from feature specialization, generalization provides an unsafe inverse to specialization. Super type generalization is an safe inverse of super type specialization which is a unsafe inverse vice versa.

Specialize Attribute either reduces the attribute's multiplicity or specializes the attribute's type. When reducing multiplicity, either the lower bound is increased or the upper bound is decreased. When specializing the type, a type conversion maps the original set of values onto a new set of values conforming the new attribute type. Specializing type conversions are surjective. *Generalize Attribute* extends the attribute's multiplicity or generalizes the attribute's type. Generalizing an attribute's type involves an injective type conversion. Type conversions are generally either implemented by transformations for each type to an intermediate format (e.g. by serialization) or by transformations for each combination of types. The latter is more elaborate to implement, yet less fragile. Most generalizing type conversions from type `x` to `y` have a specializing type conversion from type `y` to `x` as safe inverse. Applying the composition vice versa yields an unsafe inverse.

Similar to attributes, reference multiplicity can be specialized and generalized. *Specialize / Generalize Reference* can additionally specialize or generalize the type of a reference by choosing a sub type or super type of the original type, respectively. Model migration of reference specialization requires deletion of links not conforming the new reference type. *Specialize Composite Reference* is a special case of reference specialization at the metamodel level, which requires contained objects to be migrated to the targeted subclass at the model level, to ensure composition restrictions. *Specialize Composite Reference* is unsafely model-migrating.

Super type declarations are commonly adapted, while refining a metamodel. Consider the following example, in which classes `A`, `B` and `C` are part of a linear inheritance structure and remain unadapted:

```
class A     { }
class B : A { f :: Integer (1..1) }
class C : A { }
```

```
class A     { }
class B : A { f :: Integer (1..1) }
class C : B { }
```

From left to right, *Specialize Super Type* changes a declaration of super type `A` on class `C` to `B`, a sub type of `A`. Consequently, a mandatory feature `f` is inherited, which needs the creation of slots by the migration. In general, super type specialization requires addition of feature slots which are declared mandatory by the new super type. From right to left, *Generalize Super Type* changes a declaration of super type `B` on class `C` to `A`, a super type of `B`. In the new metamodel, feature `f` is no longer inherited in `C`. Slots of features which are no longer inherited need to be removed by the migration. Furthermore, links to objects of `A` that target class `B`, are no longer valid, since `A` is no longer a sub type of `B`. Therefore, these links need to be removed, if multiplicity restrictions allow, or adapted otherwise.

5.4 Inheritance Operators

Inheritance operators move features along the inheritance hierarchy. Most of them are well-known from refactoring object-oriented code [10]. There is always a pair of a constructor and destructor, where the destructor is the safe inverse of the constructor, and the constructor is the unsafe inverse of the destructor.

		Class.			MM			OODB				OOC		[12]		COPE				Acoda		
#	Operator Name	L	M	I	[24]	[2]	[6]	[1]	[3]	[19]	[7]	[10]	[8]	F	T	[13]	[14]	U	Q	B	R	Y
1	Pull up Feature	c	p	2s	x		x					x	x	x	x		x		x			
2	Push down Feature	d	u	1u	x		x					x	x		x		x					
3	Extract Super Class	c	p	4s	x		x		x	x		x	x	x	x		x		x		x	x
4	Inline Super Class	d	u	3u	x		x			x		x	x		x		x	x	x			
5	Fold Super Class	c	s	6s										x					x			
6	Unfold Super Class	d	u	5u													x		x			
7	Extract Sub Class	c	s	8s					x	x		x					x		x			
8	Inline Sub Class	d	u	7u						x		x			x				x			

Pull up Feature is a constructor which moves a feature that occurs in all subclasses of a class to the class itself. For migration, slots for the pulled up feature are added to objects of the class and filled with default values. The corresponding destructor *Push down Feature* moves a feature from a class to all its subclasses. While objects of the subclasses stay unaltered, slots for the original feature must be removed from objects of the class itself.

Extract Super Class is a constructor which introduces a new class, makes it the super class of a set of classes, and pulls up one or more features from these classes. The corresponding destructor *Inline Super Class* pushes all features of a class into its subclasses and deletes the class afterwards. References to the class are not allowed but can be generalized to a super class in a previous step. Objects of the class need to be migrated to objects of the subclasses. This might require the addition of slots for features of the subclasses.

The constructor *Fold Super Class* is related to *Extract Super Class*. Here, the new super class is not created but exists already. This existing class has a set of (possibly inherited) features. In another class, these features are defined as well. The operator then removes these features and adds instead an inheritance relation to the intended super class. In the same way, the destructor *Unfold Super Class* is related to *Inline Super Class*. This operator copies all features of a super class into a subclass and removes the inheritance relation between both classes. Here is an example for both operators:

```
class A     { f1 :: Integer }
class B : A { f2 :: Integer }
class C     { f1 :: Integer
              f2 :: Integer
              f3 :: Integer }
```

```
class A     { f1 :: Integer }
class B : A { f2 :: Integer }
class C : B { f3 :: Integer }
```

From left to right, the super class `B` is folded from class `C` which includes all the features of `B`. These features are removed from `C`, and `B` becomes a super class of `C`. From right to left, the super class `B` is unfolded into class `C` by copying features `A.f1` and `B.f2` to `C`. `B` is not longer a super class of `C`.

The constructor *Extract Subclass* introduces a new class, makes it the subclass of another, and pushes down one or more features from this class. Objects of the original class must be converted to objects of the new class. The corresponding destructor *Inline Subclass* pulls up all features from a subclass into its non-abstract super class and deletes the subclass afterwards. References to the class are not allowed but can be generalized to a super class in a previous step. Objects of the subclass need to be migrated to objects of the super class.

5.5 Delegation Operators

Delegation operators move metamodel elements along compositions or ordinary references. Most of the time, they come as pairs of corresponding refactorings being safely inverse to each other.

		Class.			MM			OODB				OOC		[12]		COPE				Acoda		
#	Operator Name	L	M	I	[24]	[2]	[6]	[1]	[3]	[19]	[7]	[10]	[8]	F	T	[13]	[14]	U	Q	B	R	Y
1	Extract Class	r	s	2s	x		x		x	x		x	x	x	x		x		x		x	x
2	Inline Class	r	s	1s	x		x			x		x	x		x				x		x	
3	Fold Class	r	s	4s										x	x	x						
4	Unfold Class	r	s	3s																		
5	Move Feature over Ref.	c	s	6s	x		x			x		x	x		x		x			x	x	
6	Collect Feature over Ref.	d	u	5u													x		x			

Extract Class moves features to a new delegate class and adds a composite reference to the new class together with an opposite reference. During migration, an object of the delegate class is created for each object of the original class, slots for the moved features are moved to the new delegate object, and a link to the delegate object is created. The corresponding *Inline Class* removes a delegate class and adds its features to the referring class. There must be no other references to the delegate class. On the model level, slots of objects of the delegate class are moved to objects of the referring class. Objects of the delegate class and links to them are deleted. The operators become a pair of constructor and destructor, if the composite reference has no opposite.

Fold and *Unfold Class* are quite similar to *Extract* and *Inline Class*. The only difference is, that the delegate class exists already and thus is not created or deleted. The following example illustrates the difference:

```
class A { a1 :: Integer
          a2 :: Boolean
          r1 -> B (1..1)
          r2 -> B (0..*) }
class B { }
class C { a1 :: Integer
          r1 -> B (1..1) }
```

```
class A { c <> C (1..1)
          d <> D (1..1) opposite a }
class B {  }
class C { a1 :: Integer
          r1 -> B (1..1) }
class D { a2 :: Boolean
          r2 -> B (0..*)
          a  -> A (1..1) opposite d }
```

From left to right, the features `a1` and `r1` of class `A` are folded to a composite reference `A.c` to class `C` which has exactly these two features. In contrast, the features `a2` and `r2` of class `A` are extracted into a new delegate class `D`. From right to left, the composite reference `A.c` is unfolded which keeps `C` intact while `A.d` is inlined which removes `D`.

Move Feature along Reference is a constructor which moves a feature over a single-valued reference to a target class. Slots of the original feature must be moved over links to objects of the target class. For objects of the target class which are not linked to an object of the source class, slots with default values must be added. The destructor *Collect Feature over Reference* is a safe inverse of the last operator. It moves a feature backwards over a reference. The multiplicity of the feature might be altered during the move depending on the multiplicity of the reference. For optional and/or multi-valued references, the feature becomes optional respectively multi-valued, too. Slots of the feature must be moved over links from objects of the source class. If an object of the source class is not linked from objects of the target class, slots of the original feature are removed. Here is an example for both operators:

```
class A { f1 :: Integer (1..*)
          r1 -> B (1..1)
          r2 -> C (0..*) }
class B { }
class C { f2 :: Integer (1..1) }
```

```
class A { f2 :: Integer (0..*)
          r1 -> B (1..1)
          r2 -> C (0..*) }
class B { f1 :: Integer (1..*) }
class C { }
```

From left to right, the feature `A.f1` is moved along the reference `A.r1` to class `B`. Furthermore, the feature `C.f2` is collected over the reference `A.r2` and ends up in class `A`. Since `A.r2` is optional and multi-valued, `A.f2` becomes optional and multi-valued, too. From right to left, the feature `B.f1` is collected over the reference `A.r1`. Its multiplicity stays unaltered. Note that there is no single operator for moving `A.f2` to class `C` which makes *Collect Feature over Reference* in general uninvertible. For the special case of a single-valued reference, *Move Feature along Reference* is an unsafe inverse.

5.6 Replacement Operators

Replacement operators replace one metamodeling construct by another, equivalent construct. Thus replacement operators typically are refactorings and safely model-migrating. With the exception of the last two operators, an operator to replace the first construct by a second always comes with a safe inverse to replace the second by the first, and vice versa.

		Class.			MM			OODB				OOC		[12]		COPE				Acoda		
#	Operator Name	L	M	I	[24]	[2]	[6]	[1]	[3]	[19]	[7]	[10]	[8]	F	T	[13]	[14]	U	Q	B	R	Y
1	Subclasses to Enum.	r	s	2s								x										
2	Enum. to Subclasses	r	s	1s								x						x				
3	Reference to Class	r	s	4s	x										x				x			
4	Class to Reference	r	s	3s	x									x								
5	Inheritance to Delegation	r	s	6s						x		x		x	x		x					
6	Delegation to Inheritance	r	s	5s						x		x										
7	Reference to Identifier	c	s	8s										x								
8	Identifier to Reference	d	u	7u										x			x		x		x	

To be more flexible, empty subclasses of a class can be replaced by an attribute which has an enumeration as type, and vice versa. *Subclasses to Enumeration* deletes all subclasses of the class and creates the attribute in the class as well as the enumeration with a literal for each subclass. In a model, objects of a certain subclass are migrated to the super class, setting the attribute to the corresponding literal. Thus, the class is required to be non-abstract and to have only empty subclasses without further subclasses. *Enumeration to Subclasses* does the inverse and replaces an enumeration attribute of a class by subclasses for each literal. The following example demonstrates both directions:

```
class C { ... }
class S1 : C {}
class S2 : C {}
```

```
class C { e :: E ... }

enum E { s1, s2 }
```

From left to right, *Subclasses to Enumeration* replaces the subclasses `S1` and `S2` of class `C` by the new attribute `C.e` which has the enumeration `E` with literals `s1` and `s2` as type. In a model, objects of a subclass `S1` are migrated to class `C`, setting the attribute `e` to the appropriate literal `s1`. From right to left, *Enumeration to Subclasses* introduces a subclass to `C` for each literal of `E`. Next, it deletes the attribute `C.e` as well as the enumeration `E`. In a model, objects of class `C` are migrated to a subclass according to the value of attribute `e`.

To be able to extend a reference with features, it can be replaced by a class, and vice versa. *Reference to Class* makes the reference composite and creates the reference class as its new type. Single-valued references are created in the reference class to target the source and target class of the original reference. In a model, links conforming to the reference are replaced by objects of the reference class, setting source and target reference appropriately. *Class to Reference* does the inverse and replaces the class by a reference. To not lose expressiveness, the reference class is required to define no features other than the source and target references. The following example demonstrates both directions:

```
class S {
  r -> T (1..*) ...
}
```

```
class S { r <> R (1..*) ... }
class R { s -> S (1..1) opposite r
          t -> T (1..1) }
```

From left to right, *Reference to Class* retargets the reference `S.r` to a new reference class `R`. Source and target of the original reference can be accessed via references `R.s` and `R.t`. In a model, links conforming to the reference `r` are replaced by objects of the reference class `R`. From right to left, *Class to Reference* removes the reference class `R` and retargets the reference `S.r` directly to the target class `T`.

Inheriting features from a superclass can be replaced by delegating them to the superclass, and vice versa. *Inheritance to Delegation* removes the inheritance relationship to the superclass and creates a composite, mandatory single-valued reference to the superclass. In a model, the slots of the features inherited from the superclass are extracted to a separate object of the super class. By removing the super type relationship, links of references to the superclass are no longer allowed to target the original object, and thus have to be retargeted to the extracted object. *Delegation to Inheritance* does the inverse and replaces the delegation to a class by an inheritance link to that class. The following example demonstrates both directions:

```
class C : S { ... }
```

```
class C { s <> S (1..1), ... }
```

From left to right, *Inheritance to Delegation* replaces the inheritance link of class `C` to its superclass `S` by a composite, single-valued reference from `C` to `S`. In a model, the slots of the features inherited from the super class `S` are extracted to a separate object of the super class. From right to left, *Delegation to Inheritance* removes the reference `C.s` and makes `S` a super class of `C`.

To decouple a reference, it can be replaced by an indirect reference using an identifier, and vice versa. *Reference to Identifier* deletes the reference and creates an attribute in the source class whose value refers to an id attribute in the target class. In a model, links of the reference are replaced by setting the attribute in the source object to the identifier of the target object. *Identifier to Reference* does the inverse and replaces an indirect reference via identifier by a direct reference. Our metamodeling formalism does not provide a means to ensure that there is a target object for each identifier used by a source object. Consequently, *Reference to Identifier* is a constructor and *Identifier to Reference* a destructor, thus being an exception in the group of replacement operators.

5.7 Merge / Split Operators

Merge operators merge several metamodel elements of the same type into a single element, whereas split operators split a metamodel element into several elements of the same type. Consequently, merge operators typically are destructors and split operators constructors. In general, each merge operator has an inverse split operator. Split operators are more difficult to define, as they may require metamodel-specific information about how to split values. There are different merge and split operators for the different metamodeling constructs.

		Class.			MM			OODB				OOC		[12]		COPE				Acoda		
#	Operator Name	L	M	I	[24]	[2]	[6]	[1]	[3]	[19]	[7]	[10]	[8]	F	T	[13]	[14]	U	Q	B	R	Y
1	Merge Features	d	u							x				x					x			
2	Split Reference by Type	r	s	1s															x			
3	Merge Classes	d	u	4u					x	x				x		x	x		x			
4	Split Class	c	p	3s																		
5	Merge Enumerations	d	u														x					

Merge Features merges a number of features defined in the same class into a single feature. In the metamodel, the source features are deleted and the target feature is required to be general enough—through its type and multiplicity—so that the values of the other features can be fully moved to it in a model. Depending on the type of feature that is merged, a repeated application of *Create Attribute* or *Create Reference* provides an unsafe inverse. *Split Reference by Type* splits a reference into references for each subclass of the type of the original reference. In a model, each link instantiating the original reference is moved to the corresponding target reference according to its type. If we require that the type of the reference is abstract, this operator is a refactoring and has *Merge Features* as a safe inverse.

Merge Classes merges a number of sibling classes—i.e. classes sharing a common superclass—into a single class. In the metamodel, the sibling classes are deleted and their features are merged to the features of the target class according to name equality. Each of the sibling classes is required to define the same features so that this operator is a destructor. In a model, objects of the sibling classes are migrated to the new class. *Split Class* is an unsafe inverse and splits a class into a number of classes. A function that maps each object of the source class to one of the target classes needs to be provided to the migration.

Merge Enumerations merges a number of enumerations into a single enumeration. In the metamodel, the source enumerations are deleted and their literals are merged to the literals of the target enumeration according to name equality.

Each of the source enumerations is required to define the same literals so that this operator is a destructor. Additionally, attributes that have the source enumerations as type have to be retargeted to the target enumeration. In a model, the values of these attributes have to be migrated according to how literals are merged. A repeated application of *Create Enumeration* provides a safe inverse.

6 Discussion

Completeness. At the metamodel level, an operator catalog is complete if any source metamodel can be evolved to any target metamodel. This kind of completeness is achieved by the catalog presented in the paper. An extreme strategy would be the following [1]: In a first step, the original metamodel needs to be discarded. Therefore, we delete opposite references and features. Next, we delete data types and enumerations and collapse inheritance hierarchies by inlining subclasses. We can now delete the remaining classes. Finally, we delete packages. In a second step, the target metamodel is constructed from scratch by creating packages, enumerations, literals, data types, classes, attributes, and references. Inheritance hierarchies are constructed by extracting empty subclasses.

Completeness is much harder to achieve, when we take the model level into account. Here, an operator catalog is complete if any model migration corresponding to an evolution from a source metamodel to a target model can be expressed. In this sense, a complete catalog needs to provide a full-fledged model transformation language based on operators. A first useful restriction is Turing completeness. But reaching for this kind of completeness comes at the price of usability. Given an existing operator, one can always think of a slightly different operator having the same effect on the metamodel level but a slightly different migration. But the higher the number of coupled operators, the more difficult it is to find an operator in the catalog. And with many similar operators, it is hard to decide which one to apply.

We therefore do not target theoretical completeness—to capture all possible migrations—but rather practical completeness—to capture migrations that likely happen in practice. When we started our case studies, we found the set of operators from the literature rather incomplete. For each case study, we added frequently reoccurring operators to the catalog. The number of operators we added to the catalog declined with every new case study, thus approaching a stable catalog. Our latest studies revealed no new operators. Although, in most case studies, we found a few operators which were only applied once in a single case study. They were never reused in other case studies. Therefore, we decided not to include them in the catalog.

We expect similar special cases in practical applications where only a few evolution steps can not be modeled by operators from the catalog. These cases can effectively be handled by providing a means for overwriting a coupling [13]: The user can specify metamodel evolution by an operator application but overwrites the model migration for this particular application. This way, theoretical completeness can still be achieved.

Metamodeling Formalism. In this paper, we focus only on core metamodeling constructs that are interesting for coupled evolution of metamodels and models. But a metamodel defines not only the abstract syntax of a modeling language, but also an API to access models expressed in this language. For this purpose, concrete metamodeling formalisms like Ecore or MOF provide metamodeling constructs like interfaces, operations, derived features, volatile features, or annotations. An operator catalog will need additional operators addressing these metamodeling constructs to reach full compatibility with Ecore or MOF.

These additional operators are relevant for practical completeness. In the GMF case study [14], we found 25% of the applied operators to address changes in the API. Most of these operators do not require migration. The only exceptions were annotations containing constraints. An operator catalog accounting for constraints needs to deal with two kinds of migrations: First, the constraints need migration when the metamodel evolves. Operators need to provide this migration in addition to model migration. Second, evolving constraints might invalidate existing models and thus require model migration. Here, new coupled operators for the evolution of constraints are needed.

Things become more complicated when it comes to CMOF [18]. Concepts like package merge, feature subsetting, and visibility affect the semantics of operators in the paper and additional operators are needed to deal with these concepts. For example, we would need four different kinds of *Rename* due to the package merge: 1) Renaming an element which is not involved in a merge neither before nor after the renaming (*Rename Element*). 2) Renaming an element which is not involved in a merge in order to include it into a merge (*Include by Name*). 3) Renaming an element which is involved in a merge in order to exclude it from the merge (*Exclude by Name*). 4) Renaming all elements which are merged to the same element (*Rename Merged Element*).

Tool Support. In operator-based tools, operators are usually made available to the user through an operator browser [23,13]. Here, the organization of the catalog into groups can help to find an operator for a change at hand. The preservation properties can be used to reason about the impact on language expressiveness and on existing models. In grammarware, similar operators have been successfully used in [16] to reason about relationships between different versions of the Java grammar. To make the user aware of the impact on models, it can be shown by a traffic light in the browser: green for model-preserving, yellow for safely and red for unsafely model-migrating. Additionally, the operator browser may have different modes for restricting the presented operators in order to guarantee language- and/or model-preservation properties. Bidirectionality can be used to invert an evolution that has been specified erroneously earlier. Recorded operator applications can be automatically undone with different levels of safety by applying the inverse operators. Tools that support evolution detection should evade of destructors in favor of refactorings to increase the preservation of information by the detected evolution.

Difference-based tools [6,11] need to be able to specify the mappings underlying the operators from the catalog. When they allow to specify complex mappings, they could introduce means to specify the mappings of the operators in a straightforward manner. Introducing such first class constructs reduces the effort for specifying the migration. For instance, the declarative language presented in [17] provides patterns to specify recurrent mappings.

7 Conclusion

We presented a catalog of 61 operators for the coupled evolution of metamodels and models. These so-called coupled operators evolve a metamodel and in response are able to automatically migrate existing models. The catalog covers not only well-known operators from the literature, but also operators which have proven useful in a number of case studies we performed. The catalog is based on the widely used EMOF metamodeling formalism [18] which was stripped of the constructs that cannot be instantiated in models. When a new construct is added to the metamodeling formalism, new operators have to be added to the catalog: Primitive operators to create, delete and modify the construct as well as complex operators to perform more intricate evolutions involving the construct. The catalog not only serves as a basis for operator-based tools, but also for difference-based tools. Operator-based tools need to provide an implementation of the presented operators. Difference-based tools need to be able to specify the mappings underlying the presented operators.

Acknowledgments. The work of the first author was funded by the German Federal Ministry of Education and Research (BMBF), grants “SPES 2020, 01IS08045A” and “Quamoco, 01IS08023B”. The work of the other authors was supported by NWO/JACQUARD, project 638.001.610, MoDSE: Model-Driven Software Evolution.

References

1. Banerjee, J., Kim, W., Kim, H.J., Korth, H.F.: Semantics and implementation of schema evolution in object-oriented databases. SIGMOD Rec. 16(3), 311–322 (1987)
2. Becker, S., Goldschmidt, T., Gruschko, B., Koziolek, H.: A process model and classification scheme for semi-automatic meta-model evolution. In: Proc. 1st Workshop MDD, SOA und IT-Management (MSI 2007), pp. 35–46. GiTO-Verlag (2007)
3. Brèche, P.: Advanced primitives for changing schemas of object databases. In: Constantopoulos, P., Vassiliou, Y., Mylopoulos, J. (eds.) CAiSE 1996. LNCS, vol. 1080, pp. 476–495. Springer, Heidelberg (1996)
4. Burger, E., Gruschko, B.: A Change Metamodel for the Evolution of MOF-Based Metamodels. In: Modellierung 2010. GI-LNI, vol. P-161 (2010)
5. Casais, E.: Managing class evolution in object-oriented systems. In: Object-Oriented Software Composition, pp. 201–244. Prentice Hall, Englewood Cliffs (1995)
6. Cicchetti, A., Ruscio, D.D., Eramo, R., Pierantonio, A.: Automating co-evolution in model-driven engineering. In: EDOC 2008: 12th International Enterprise Distributed Object Computing Conference. IEEE, Los Alamitos (2008)

7. Claypool, K.T., Rundensteiner, E.A., Heineman, G.T.: ROVER: A framework for the evolution of relationships. In: Laender, A.H.F., Liddle, S.W., Storey, V.C. (eds.) ER 2000. LNCS, vol. 1920, pp. 893–917. Springer, Heidelberg (2000)
8. Dig, D., Johnson, R.: How do APIs evolve? a story of refactoring. J. Softw. Maint. Evol. 18(2), 83–107 (2006)
9. Favre, J.M.: Languages evolve too! changing the software time scale. In: IWPSE 2005: 8th International Workshop on Principles of Software Evolution, pp. 33–42 (2005)
10. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Longman Publishing Co., Inc., Amsterdam (1999)
11. Garcés, K., Jouault, F., Cointe, P., Bézivin, J.: Managing model adaptation by precise detection of metamodel changes. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 34–49. Springer, Heidelberg (2009)
12. Herrmannsdoerfer, M., Benz, S., Juergens, E.: Automatability of coupled evolution of metamodels and models in practice. In: Busch, C., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 645–659. Springer, Heidelberg (2008)
13. Herrmannsdoerfer, M., Benz, S., Juergens, E.: COPE - automating coupled evolution of metamodels and models. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 52–76. Springer, Heidelberg (2009)
14. Herrmannsdoerfer, M., Ratiu, D., Wachsmuth, G.: Language evolution in practice: The history of GMF. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 3–22. Springer, Heidelberg (2010)
15. Lämmel, R.: Grammar adaptation. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 550–570. Springer, Heidelberg (2001)
16. Lammel, R., Zaytsev, V.: Recovering grammar relationships for the Java language specification. In: SCAM 2009: Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, pp. 178–186. IEEE, Los Alamitos (2009)
17. Narayanan, A., Levendovszky, T., Balasubramanian, D., Karsai, G.: Automatic domain model migration to manage metamodel evolution. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 706–711. Springer, Heidelberg (2009)
18. Object Management Group: Meta Object Facility (MOF) core specification version 2.0 (2006), `http://www.omg.org/spec/MOF/2.0/`
19. Pons, A., Keller, R.: Schema evolution in object databases by catalogs. In: IDEAS 1997: International Database Engineering and Applications Symposium, pp. 368–376 (1997)
20. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.A.: An analysis of approaches to model migration. In: Models and Evolution (MoDSE-MCCM) Workshop (2009)
21. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0. Addison-Wesley, Reading (2009)
22. Vermolen, S.D., Visser, E.: Heterogeneous coupled evolution of software languages. In: Busch, C., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 630–644. Springer, Heidelberg (2008)
23. Wachsmuth, G.: An adaptation browser for MOF. In: WRT 2001: First Workshop on Refactoring Tools, pp. 65–66 (2007)
24. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In: Bateni, M. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 600–624. Springer, Heidelberg (2007)

JTL: A Bidirectional and Change Propagating Transformation Language

Antonio Cicchetti[1], Davide Di Ruscio[2], Romina Eramo[2], and Alfonso Pierantonio[2]

[1] School of Innovation, Design and Engineering
Mälardalen University,
SE-721 23, Västerås, Sweden
antonio.cicchetti@mdh.se
[2] Dipartimento di Informatica
Università degli Studi dell'Aquila
Via Vetoio, Coppito I-67010, L'Aquila, Italy
{davide.diruscio,romina.eramo,alfonso.pierantonio}@univaq.it

Abstract. In Model Driven Engineering bidirectional transformations are considered a core ingredient for managing both the consistency and synchronization of two or more related models. However, while non-bijectivity in bidirectional transformations is considered relevant, current languages still lack of a common understanding of its semantic implications hampering their applicability in practice.

In this paper, the Janus Transformation Language (JTL) is presented, a bidirectional model transformation language specifically designed to support non-bijective transformations and change propagation. In particular, the language propagates changes occurring in a model to one or more related models according to the specified transformation regardless of the transformation direction. Additionally, whenever manual modifications let a model be non reachable anymore by a transformation, the *closest* model which approximate the ideal source one is inferred. The language semantics is also presented and its expressivity and applicability are validated against a reference benchmark. JTL is embedded in a framework available on the Eclipse platform which aims to facilitate the use of the approach, especially in the definition of model transformations.

1 Introduction

In Model-Driven Engineering [1] (MDE) model transformations are considered as the gluing mechanism between the different abstraction layers and viewpoints by which a system is described [2,3]. Their employment includes mapping models to other models to focus on particular features of the system, operate some analysis, simulate/validate a given application, not excluding the operation of keeping them synchronized or in a consistent state. Given the variety of scenarios in which they can be employed, each transformation problem can demand for different characteristics making the expectation of a single approach suitable for all contexts not realistic.

Bidirectionality and change propagation are relevant aspects in model transformations: often it is assumed that during development only the source model of a transformation undergoes modifications, however in practice it is necessary for developers to modify both the source and the target models of a transformation and propagate changes in both directions [4,5]. There are two main approaches for realizing bidirectional transformations: by programming forward and backward transformations in any convenient unidirectional language and manually ensuring they are consistent; or by using a bidirectional transformation language where every program describes both a forward and a backward transformation simultaneously. A major advantage of the latter approach is that the consistency of the transformations can be guaranteed by construction. Moreover, source and target roles are not fixed since the transformation direction entails them. Therefore, considerations made about the mapping executed in one direction are completely equivalent to the opposite one.

The relevance of bidirectionality in model transformations has been acknowledged already in 2005 by the Object Management Group (OMG) by including a bidirectional language in their Query View Transformation (QVT) [6]. Unfortunately, as pointed out by Perdita Stevens in [7] the language definition is affected by several weaknesses. Therefore, while MDE requirements demand enough expressiveness to write non-bijective transformations [8], the QVT standard does not clarify how to deal with corresponding issues, leaving their resolution to tool implementations. Moreover, a number of approaches and languages have been proposed due to the intrinsic complexity of bidirectionality. Each one of those languages is characterized by a set of specific properties pertaining to a particular applicative domain [9].

This paper presents the Janus Transformation Language (JTL), a declarative model transformation language specifically tailored to support bidirectionality and change propagation. In particular, the distinctive characteristics of JTL are

– *non-bijectivity*, non-bijective bidirectional transformations are capable of mapping a model into a set of models, as for instance when a single change in a target model might semantically correspond to a family of related changes in more than one source model. JTL provides support to non-bijectivity and its semantics assures that all the models are computed at once independently whether they represent the outcome of the backward or forward execution of the bidirectional transformation;
– *model approximation*, generally transformations are not total which means that target models can be manually modified in such a way they are not reachable anymore by any forward transformation, then traceability information are employed to back propagate the changes from the modified targets by inferring the *closest* model that approximates the ideal source one at best.

The language expressiveness and applicability have been validated by implementing the *Collapse/Expand State Diagrams* benchmark which have been defined in [10] to compare and assess different bidirectional approaches. The JTL semantics is defined in terms of the Answer Set Programming (ASP) [11], a form of declarative programming oriented towards difficult (primarily NP-hard) search problems and based on the stable model (answer set) semantics of logic programming. Bidirectional transformations are translated via semantic anchoring [12] into search problems which are reduced to computing stable models, and the DLV solver [13] is used to perform search.

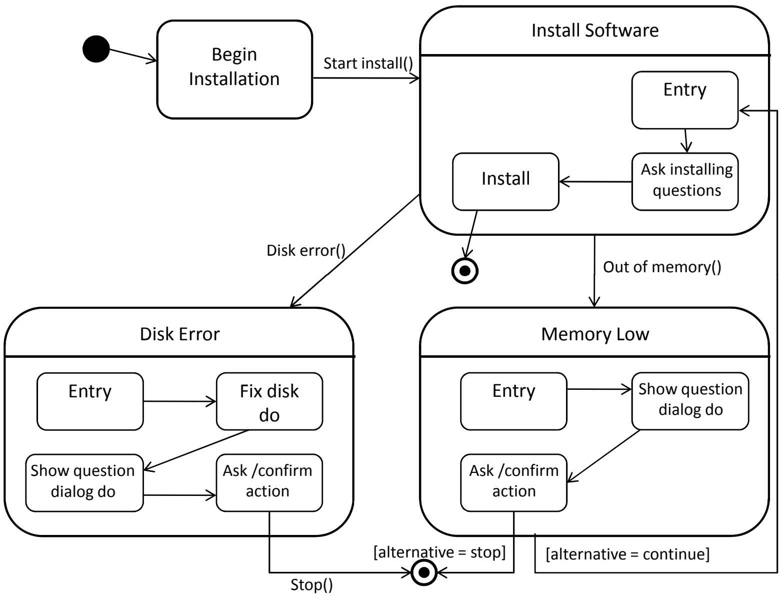
The structure of the paper is as follows: Section 2 sets the context of the paper through a motivating example that is used throughout the paper to demonstrate the approach and Section 3 discusses requirements a bidirectional and change propagating language should support. Section 4 describes conceptual and implementation aspects of the proposed approach and Section 5 applies the approach to a case study. Section 6 relates the work presented in this paper with other approaches. Finally, Section 7 draws the conclusions and presents future work.

2 Motivating Scenario

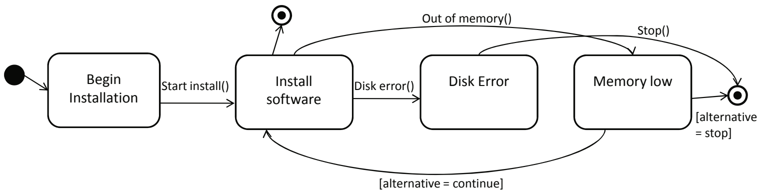
As aforesaid, bidirectionality in model transformations raises not obvious issues mainly related to non-bijectivity [7,14]. More precisely, let us consider the *Collapse/Expand State Diagrams* benchmark defined in the *GRACE International Meeting on Bidirectional Transformations* [10]: starting from a hierarchical state diagram (involving some one-level nesting) as the one reported in Figure 1.a, a flat view has to be provided as in Figure 1.b. Furthermore, any manual modifications on the (target) flat view should be back propagated and eventually reflected in the (source) hierarchical view. For instance, let us suppose the designer modifies the flat view by changing the name of the initial state from `Begin Installation` to `Start Install shield` (see Δ_1 change in Figure 2). Then, in order to persist such a refinement to new executions of the transformation, the hierarchical state machine has to be consistently updated by modifying its initial state as illustrated in Figure 3.

The flattening is a non-injective operation requiring specific support to back propagate modifications operated on the flattened state machine to the nested one. For instance, the flattened view reported in Figure 1 can be extended by adding the alternative `try again` from the state `Disk Error` to `Install software` (see Δ_2 changes in Figure 2). This gives place to an interesting situation: the new transition can be equally mapped to each one of the nested states within `Install Software` as well as to the container state itself. Consequently, more than one source model propagating the changes exists[1]. Intuitively, each time hierarchies are flattened there is a loss of information which causes ambiguities when trying to map back corresponding target revisions. Some of these problems can be alleviated by managing traceability information of the transformation executions which can be exploited later on to trace back the changes: like this each generated element can be linked with the corresponding source and contribute to the resolution of some of the ambiguities. Nonetheless, traceability is a necessary but not sufficient condition to support bidirectionality, since for instance elements discarded by the mapping may not appear in the traces, as well as new elements added on the target side. For instance, the generated flattened view in Figure 1.b can be additionally manipulated through the Δ_3 revisions which consist of adding some extra-functional information for the `Install Software` state and the transition between from `Memory low` and `Install Software` states. Because of the limited expressive power of the hierarchical state machine metamodel which does not support extra-functional annotations, the Δ_3 revisions do not have counterparts in the state machine in Figure 3.

[1] It is worth noting that the case study and examples have been kept deliberately simple since they suffice to show the relevant issues related to non-bijectivity



a) A sample Hierarchical State Machine (HSM).



b) The corresponding Non-Hierarchical State Machine (NHSM).

Fig. 1. Sample models for the *Collapse/Expand State Diagrams* benchmark

Current declarative bidirectional languages, such as QVT relations (QVT-R), are often ambivalent when discussing non-bijective transformations as already pointed out [7]; whilst other approaches, notably hybrid or graph-based transformation techniques, even if claiming the support of bidirectionality, are able to deal only with (partially) bijective mappings [4]. As a consequence, there is not a clear understanding of what non-bijectivity implies causing language implementors to adopt design decisions which differ from an implementation to another.

In order to better understand how the different languages deal with non-bijectivity, we have specified the hierarchical to non-hierarchical state machines transformation (HSM2NHSM) by means of the Medini[2] and MOFLON[3] systems. The former is an

[2] `http://projects.ikv.de/qvt/`
[3] `http://www.moflon.org`

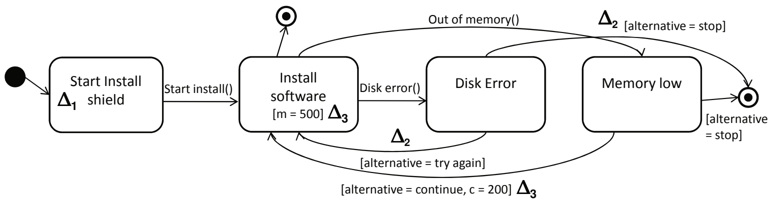


Fig. 2. A revision of the generated non-hierarchical state machine

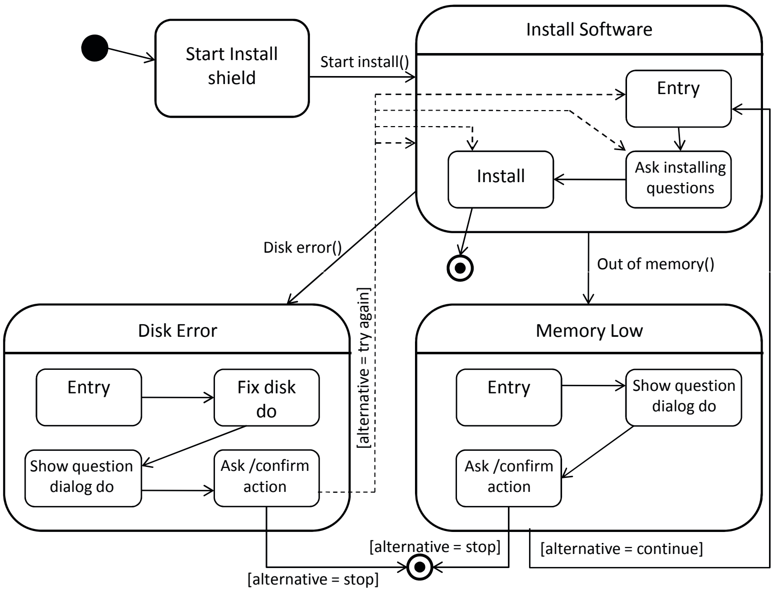


Fig. 3. The source hierarchical state machine synchronised with the target changes

implementation of the QVT-R transformation language, whereas the latter is a framework which bases on Triple Graph Grammars (TGGs) [15]: our experience with them is outlined in the following

Medini. When trying to map the generated target model back to the source without any modification, a new source model is generated which differs from the original one[4]. In particular, incoming (outgoing) transitions to (from) nested states are flattened to the corresponding parent: when going back such mapping makes the involved nested states to disappear (as `Entry` and `Install` in the `Install Software` composite in Figure 1). Moreover, the same mapping induces the creation of extra composite states for existing simple states, like `Begin Installation` and the initial and final states of the hierarchical state machine. Starting from this status, we made the modifications

[4] In this paper the details about the experiments done with Medini and TGGs can not be described in detail due to space restrictions. The interested reader can access the full implementation of both the attempts at http://www.mrtc.mdh.se/∼acicchetti/HSM2NHSM.php

on the target model as prescribed by Figure 2 and re-applied the transformation in the source direction, i.e. backward. In this case, the `Start Install shield` state is correctly mapped back by renaming the existing `Begin Installation` in the source. In the same way, the modified transition from `Disk Error` to the final state is consistently updated. However, the newly added transition outgoing from `Disk Error` to `Install software` is mapped by default to the composite state, which might not be the preferred option for the user. Finally, the manipulation of the attributes related to memory requirements and cost are not mapped back to any source element but are preserved when new executions of the transformation in the target direction are triggered.

MOFLON. The TGGs implementation offered by MOFLON is capable of generating Java programs starting from diagrammatic specifications of graph transformations. The generated code realizes two separate unidirectional transformations which as in other bidirectional languages should be consistent by construction. However, while the forward transformation implementation can be considered complete with respect to the transformation specification, the backward program restricts the change propagation to attribute updates and element deletions. In other words, the backward propagation is restricted to the contexts where the transformation can exploit trace information.

In the next sections, we firstly motivate a set of requirements a bidirectional transformation language should meet to fully achieve its potential; then, we introduce the JTL language, its support to non-bijective bidirectional transformations, and its ASP-based semantics.

3 Requirements for Bidirectionality and Change Propagation

This section refines the definition of bidirectional model transformations as proposed in [7] by explicitly considering non-bijective cases. Even if some of the existing bidirectional approaches enable the definition of non-bijective mappings [7,5], their validity is guaranteed only on bijective sub-portions of the problem. As a consequence, the forward transformation can be supposed to be an injective function, and the backward transformation its corresponding inverse; unfortunately, such requirement excludes most of the cases [16]. In general, a bidirectional transformation R between two classes of models, say M and N, and M more expressive than N, is characterized by two unidirectional transformations

$$\overrightarrow{R} : M \times N \to N$$
$$\overleftarrow{R} : M \times N \to M^*$$

where $\overrightarrow{R}$ takes a pair of models *(m, n)* and works out how to modify n so as to enforce the relation $\overrightarrow{R}$. In a similar way, $\overleftarrow{R}$ propagates changes in the opposite direction: $\overleftarrow{R}$ is a non-bijective function able to map the target model in a set of corresponding source models conforming to M[5]. Furthermore, since transformations are not total in

[5] For the sake of readability, we consider a non-bijective backward transformation assuming that only M contains elements not represented in N. However, the reasoning is completely analogous for the forward transformation and can be done by exchanging the roles of M and N.

general, bidirectionality has to be provided even in the case the generated model has been manually modified in such a way it is not reachable anymore by the considered transformation. Traceability information is employed to back propagate the changes from the modified targets by inferring the *closest*[6] model that approximates the ideal source one at best. More formally the backward transformation $\overleftarrow{R}$ is a function such that:

(i) if *R(m,n)* is a non-bijective consistency relation, $\overleftarrow{R}$ generates all the resulting models according to R;
(ii) if *R(m,n)* is a non-total consistency relation, $\overleftarrow{R}$ is able to generate a result model which approximates the ideal one.

This definition alone does not constrain much on the behavior of the reverse transformation and additional requirements are necessary in order to ensure that the propagation of changes behaves as expected.

Reachability. In case a generated model has been manually modified (n'), the backward transformation $\overleftarrow{R}$ generates models (m^*) which are exact, meaning that the original target may be reached by each of them via the transformation without additional side effects. Formally:

$$\overleftarrow{R}(m, n') = m^* \in M^*$$

$$\overrightarrow{R}(m', n') = n' \in N \text{ for each } m' \in m^*$$

Choice preservation. Let n' be the target model generated from an arbitrary model m' in m^* as above: when the user selects m' as the appropriate source pertaining to n' the backward transformation has to generate exactly m' from n' disregarding the other possible alternatives $t \in m^*$ such that $t \neq m'$. In other words, a valid round-trip process has to be guaranteed even when multiple sources are available [14]:

$$\overleftarrow{R}(m', \overrightarrow{R}(m', n')) = m' \text{ for each } m' \in m^*$$

Clearly, the above requirement in order to be met demands for adequate traceability information management.

In the rest of the paper, the proposed language is introduced and shown to satisfy the above requirements. The details of the language and its supporting development environment are presented in Section 4, whereas in Section 5 the usage of the language is demonstrated by means of the benchmark case.

4 The Janus Transformation Language

The Janus Transformation Language (JTL) is a declarative model transformation language specifically tailored to support bidirectionality and change propagation. The

[6] This concept is clarified in Sect. 4, where the transformation engine and its derivation mechanism are discussed.

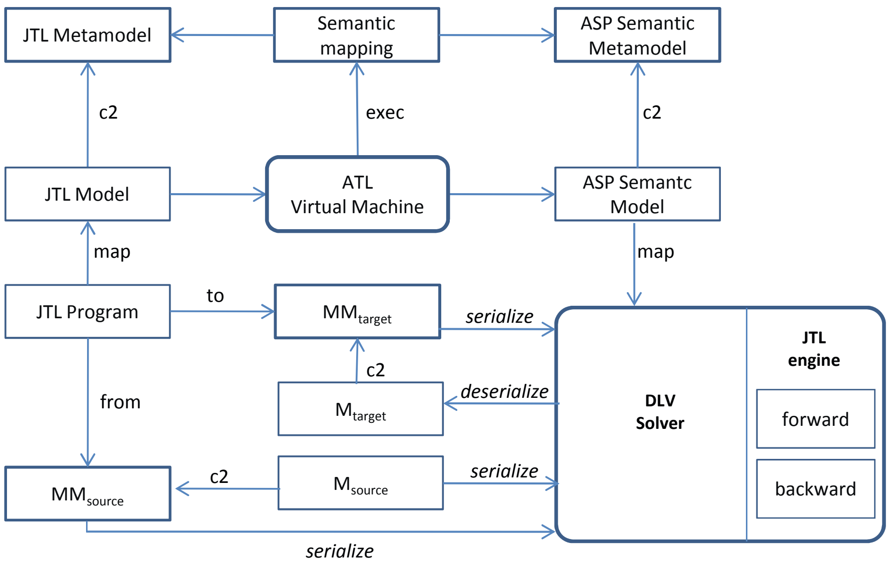


Fig. 4. Architecture overview of the JTL environment

implementation of the language relies on the Answer Set Programming (ASP) [11]. This is a form of declarative programming oriented towards difficult (primarily NP-hard) search problems and based on the stable model (answer set) semantics of logic programming. Being more precise model transformations specified in JTL are transformed into ASP programs (search problems), then an ASP solver is executed to find all the possible stable models that are sets of atoms which are consistent with the rules of the considered program and supported by a deductive process.

The overall architecture of the environment supporting the execution of JTL transformations is reported in Figure 4. The *JTL engine* is written in the ASP language and makes use of the *DLV solver* [13] to execute transformations in both forward and backward directions. The engine executes JTL transformations which have been written in a QVT-like syntax, and then automatically transformed into ASP programs. Such a semantic anchoring has been implemented in terms of an ATL [17] transformation defined on the JTL and ASP metamodels. Also the source and target metamodels of the considered transformation (MM_{source}, MM_{target}) are automatically encoded in ASP and managed by the engine during the execution of the considered transformation and to generate the output models.

The overall architecture has been implemented as a set of plug-ins of the Eclipse framework and mainly exploits the Eclipse Modelling Framework (EMF) [18] and the ATLAS Model Management Architecture (AMMA) [19]. Moreover, the DLV solver has been wrapped and integrated in the overall environment.

In the rest of the section all the components of the architecture previously outlined are presented in detail. In particular, Section 4.1 presents the JTL engine, the syntax

```
1 metanode(HSM, state).
2 metanode(HSM, transition).
3 metaprop(HSM, name, state).
4 metaprop(HSM, trigger, transition).
5 metaprop(HSM, effect, transition).
6 metaedge(HSM, association, source, transition, state).
7 metaedge(HSM, association, target, transition, state).
8 [...]
```

Listing 1.1. Fragment of the State Machine metamodel

of transformation language is described in Section 4.2 by using a running example, whereas the semantic anchoring is described in Section 4.3.

4.1 The Janus Transformation Engine

As previously said the Janus transformation engine is based on a relational and declarative approach implemented using the ASP language to specify bidirectional transformations. The approach exploits the benefits of logic programming that enables the specification of relations between source and target types by means of predicates, and intrinsically supports bidirectionality [9] in terms of unification-based matching, searching, and backtracking facilities.

Starting from the encoding of the involved metamodels and the source model (see the *serialize* arrows in the Figure 4), the representation of the target one is generated according to the JTL specification (as shown in Section 4.2). The computational process is performed by the JTL engine (as depicted in Figure 4) which is based on an ASP bidirectional transformation program executed by means of an ASP solver called DLV [13].

Encoding of models and metamodels. In the proposed approach, models and metamodels are defined in a declarative manner by means of a set of logic assertions. In particular, they are considered as graphs composed of nodes, edges and properties that qualify them. The metamodel encoding is based on a set of *terms* each characterized by the predicate symbols `metanode`, `metaedge`, and `metaprop`, respectively. A fragment of the hierarchical state machine metamodel considered in Section 2 is encoded in Listing 1.1. For instance, the `metanode(HSM,state)` in line `1` encodes the metaclass `state` belonging to the metamodel `HSM`. The `metaprop(HSM,name,state)` in line `3` encodes the attribute named `name` of the metaclass `state` belonging to the metamodel `HSM`. Finally, the `metaedge(HSM,association,source,transition,state)` in line `6` encodes the association between the metaclasses `transition` and `state`, typed `association`, named `source` and belonging to the metamodel `HSM`. The terms induced by a certain metamodel are exploited for encoding models conforming to it. In particular, models are sets of entities (represented through the predicate symbol `node`), each characterized by properties (specified by means of `prop`) and related together by relations (represented by `edge`). For instance, the state machine model in Figure 1

```
 1 node(HSM, "s1", state).
 2 node(HSM, "s2", state).
 3 node(HSM, "t1", transition).
 4 prop(HSM,"s1.1","s1",name,"begin␣installation").
 5 prop(HSM,"s2.1","s2",name,"install␣software").
 6 prop(HSM,"t1.1","t1",trigger,"install␣software").
 7 prop(HSM,"t1.2","t1",effect,"start␣install").
 8 edge(HSM,"tr1",association,source, "s1","t1").
 9 edge(HSM,"tr1",association,target, "s2","t1").
10 [...]
```

Listing 1.2. Fragment of the State Machine model in Figure 1

is encoded in the Listing 1.2. In particular, the `node(HSM,"s1",state)` in line `1` encodes the instance identified with `"s1"` of the class `state` belonging to the metamodel `HSM`. The `prop(HSM,"s1",name,"start")` in line `4` encodes the attribute `name` of the class `"s1"` with value `"start"` belonging to the metamodel `HSM`. Finally, the `edge(HSM,"tr1",association,source,"s1","t1")` in line `7` encodes the instance `"tr1"` of the association between the state `"s1"` and the transition `"t1"` belonging to the metamodel `HSM`.

Model transformation execution. After the encoding phase, the deduction of the target model is performed according to the rules defined in the ASP program. The transformation engine is composed of *i) relations* which describe correspondences among element types of the source and target metamodels, *ii) constraints* which specify restrictions on the given relations that must be satisfied in order to execute the corresponding mappings, and an *iii) execution engine* (described in the rest of the section) consisting of bidirectional rules implementing the specified relations as executable mappings. Relations and constraints are obtained from the given JTL specification, whereas the execution engine is always the same and represents the bidirectional engine able to interpret the correspondences among elements and execute the transformation.

The transformation process logically consists of the following steps:

(i) given the input (meta)models, the execution engine induces all the possible solution candidates according to the specified relations;
(ii) the set of candidates is refined by means of constraints.

The Listing 1.3 contains a fragment of the ASP code implementing relations and constraints of the HSM2NHSM transformation discussed in Section 2. In particular, the terms in lines `1-2` define the relation called `"r1"` between the metaclass `State machine` belonging to the `HSM` metamodel and the metaclass `State machine` belonging to the `NHSM` metamodel. An ASP constraint expresses an invalid condition: for example, the constraints in line `3-4` impose that each time a state machine occurs in the source model it has to be generated also in the target model. In fact, if each atoms in its body is true then the correspondent solution candidate is eliminated. In similar way, the relation between the metaclasses `State` of the involved metamodels is encoded in line `6-7`. In this case, constraints in line `8-11` impose that each time a state occurs in the HSM model, the correspondent one in the NHSM model is generated only if the source element is not a sub-state, vice versa, each state in the NHSM model is

```
 1 relation ("r1",  HSM, stateMachine).
 2 relation ("r1", NHSM, stateMachine).
 3 :- node(HSM, "sm1", stateMachine), not  node'(HSM, "sm1", stateMachine).
 4 :- node(NHSM, "sm1", stateMachine), not  node'(NHSM, "sm1", stateMachine).
 5
 6 relation ("r2", HSM, state).
 7 relation ("r2", NHSM, state).
 8 :- node(HSM, "s1", state), not edge(HSM, "ow1", owningCompositeState, "s1", "cs1
      "), not node'(NHSM, "s1", state).
 9 :- node(HSM, "s1", state), edge(HSM, "ow1", owningCompositeState, "s1", "cs1"),
      node(HSM, "cs1", compositeState), node'(NHSM, "s1", state).
10 :- node(NHSM, "s1", state), not trace_node(HSM, "s1", compositeState), not node
      '(HSM, "s1", state).
11 :- node(NHSM, "s1", state), trace_node(HSM, "s1", compositeState), node'(HSM, "
      s1", state).
12
13 relation ("r3", HSM, compositeState).
14 relation ("r3", NHSM, state).
15 :- node(HSM, "s1", compositeState), not node'(NHSM, "s1", state).
16 :- node(NHSM, "s1", state), trace_node(HSM, "s1", compositeState), not node'(HSM
      , "s1", compositeState).
17 [...]
```

Listing 1.3. Fragment of the HSM2NHSM transformation

mapped in the HSM model. Finally, the relation between the metaclasses `Composite state` and `State` is encoded in line `13-14`. Constraints in line `15-16` impose that each time a composite state occurs in the HSM model a correspondent state in the NHSM model is generated, and vice versa. Missing sub-states in a NHSM model can be generated again in the HSM model by means of trace information (see line `10-11` and `16`). Trace elements are automatically generated each time a model element is discarded by the mapping and need to be stored in order to be regenerated during the backward transformation.

Note that the specification order of the relations is not relevant as their execution is bottom-up; i.e., the final answer set is always deduced starting from the more nested facts.

Execution engine. The specified transformations are executed by a generic engine which is (partially) reported in Listing 1.4. The main goal of the transformation execution is the generation of target elements as the *node'* elements in line 11 of Listing 1.4. As previously said transformation rules may produce more than one target models, which are all the possible combinations of elements that the program is able to create. In particular, by referring to Listing 1.4 target node elements with the form `node'(MM,ID,MC)` are created if the following conditions are satisfied:

- the considered element is declared in the input source model. The lines `1-2` contain the rules for the source conformance checking related to `node` terms. In particular, the term `is_source_metamodel_conform(MM,ID,MC)` is true if the terms `node(MM,ID,MC)` and `metanode(MM,MC)` exist. Therefore, the term `bad_source` is true if the corresponding `is_source_metamodel_con- form(MM,ID,MC)` is valued to false with respect to the `node(MM,ID,MC)` source element.

```
1 is_source_metamodel_conform(MM,ID,MC) :- node(MM,ID,MC), metanode(MM,MC).
2 bad_source :- node(MM,ID,MC), not is_source_metamodel_conform(MM,ID,MC).
3 mapping(MM,ID,MC) :- relation(R,MM,MC), relation(R,MM2,MC2), node(MM2,ID,MC2), MM
      !=MM2.
4 is_target_metamodel_conform(MM,MC) :- metanode(MM,MC).
5 {is_generable(MM,ID,MC)} :- not bad_source, mapping(MM,ID,MC),
      is_target_metamodel_conform(MM,MC), MM=mmt.
6 node'(MM,ID,MC) :- is_generable(MM,ID,MC), mapping(MM,ID,MC), MM=mmt.
```

Listing 1.4. Fragment of the *Execution engine*

- at least a relation exists between a source element and the candidate target element. In particular, the term `mapping(MM,ID,MC)` in line `3` is true if exists a relation which involves elements referring to `MC` and `MC2` metaclasses and an element `node(MM2,ID,MC2)`. In other words, a mapping can be executed each time it is specified between a source and a target, and exists the appropriate source to compute the target.
- the candidate target element conforms to the target metamodel. In particular, the term `is_target_metamodel_conform(MM,MC)` in line `6` is true if the `MC` metaclass exists in the `MM` metamodel (i.e. the target metamodel).
- finally, any constraint defined in the *relations* in Listing 1.3 is valued to false.

The invertibility of transformations is obtained by means of trace information that connects source and target elements; in this way, during the transformation process, the relationships between models that are created by the transformation executions can be stored to preserve mapping information in a permanent way. Furthermore, all the source elements lost during the forward transformation execution (for example, due to the different expressive power of the metamodels) are stored in order to be generated again in the backward transformation execution.

4.2 Specifying Model Transformation with Janus

Due to the reduced usability of the ASP language, we have decided to provide support for specifying transformations by means of a more usable syntax inspired by QVT-R. In Listing 1.5 we report a fragment of the HSM2NHSM transformation specified in JTL and it transforms hierarchical state machines into flat state machines and the other way round. The forward transformation is clearly non-injective as many different hierarchical machines can be flattened to the same model and consequently transforming back a modified flat machine can give place to more than one hierarchical machine. Such a transformation consists of several relations like *StateMachine2StateMachine*, *State2State* and *CompositeState2State* which are specified in Listing 1.5. They define correspondences between *a)* state machines in the two different metamodels *b)* atomic states in the two different metamodels and *c)* composite states in hierarchical machines and atomic states in flat machines. The relation in lines 11-20 is constrained by means of the *when* clause such that only atomic states are considered. Similarly to QVT, the *checkonly* and *enforce* constructs are also provided: the former is used to check if the domain where it is applied exists in the considered model; the latter induces the modifications of those models which do not contain the domain specified as *enforce*. A JTL relation is considered bidirectional when both the contained domains are specified with the construct *enforce*.

```
 1 transformation hsm2nhsm(source : HSM, target : NHSM) {
 2
 3  top relation StateMachine2StateMachine {
 4
 5     enforce domain source sSM : HSM::StateMachine;
 6     enforce domain target tSM : NHSM::StateMachine;
 7
 8   }
 9
10  top relation State2State {
11
12     enforce domain source sourceState : HSM::State;
13     enforce domain target targetState : NHSM::State;
14
15     when {
16       sourceState.owningCompositeState.oclIsUndefined();
17     }
18
19   }
20
21  top relation CompositeState2State {
22
23    enforce domain source sourceState : HSM::CompositeState;
24    enforce domain target targetState : NHSM::State;
25
26  }
27 }
```

Listing 1.5. A non-injective JTL program

The JTL transformations specified in the QVT-like syntax are mapped to the correspondent ASP program by means of a semantic anchoring operation as described in the next section.

4.3 ASP Semantic Anchoring

According to the proposed approach, the designer task is limited to specifying relational model transformations in JTL syntax and to applying them on models and metamodels defined as EMF entities within the Eclipse framework.

Designers can take advantage of ASP and of the transformation properties discussed in the previous sections in a transparent manner since only the JTL syntax is used. In fact, ASP programs are automatically obtained from JTL specifications by means of ATL transformations as depicted in the upper part of Figure 4. Such a transformation is able to generate ASP predicates for each relation specified with JTL. For instance, the relation *State2State* in Listing 1.5 gives place to the *relation* predicates in lines 6-7 in Listing 1.3.

The JTL *when* clause is also managed and it induces the generation of further ASP constraints. For instance, the JTL clause in line 16 of Listing 1.5 gives place to a couple of ASP constraints defined on the *owningCompositeState* feature of the state machine metamodels (see lines 8-9 in Listing 1.3). Such constraints are able to filter the states and consider only those which are not nested.

To support the backward application of the specified transformation, for each JTL relation additional ASP constraints are generated in order to support the management of trace links. For instance, the *State2State* relation in Listing 1.5 induces the generation of the constraints in lines 10-11 of Listing 1.3 to deal with the non-bijectivity of the transformation. In particular, when the transformation is backward applied on a *State* element of the target model, trace links are considered to check if such a state has been previously generated from a source *CompositeState* or *State* element. If such trace information is missing all the possible alternatives are generated.

5 JTL in Practice

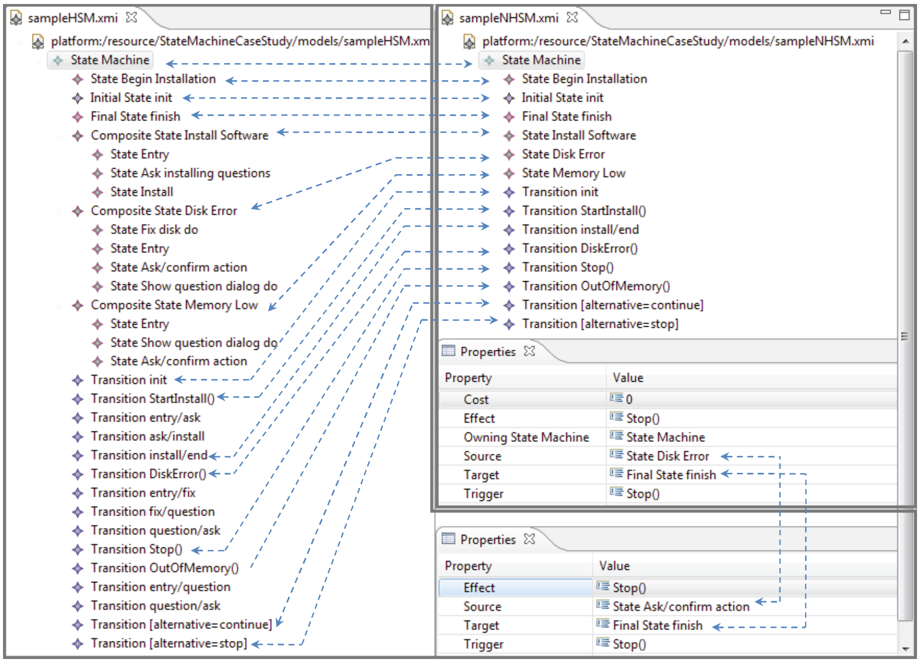
In this section we show the application of the proposed approach to the *Collapse/Expand State Diagrams* case study presented in Section 2. The objective is to illustrate the use of JTL in practice by exploiting the developed environment, and in particular to show how the approach is able to propagate changes dealing with non-bijective and non-total scenarios. The following sections present how after the definition of models and metamodels (see Section 5.1), the JTL transformation may be specified and applied over them (see Section 5.2). Finally, the approach is also applied to manage changes occurring on the target models which need to be propagated to the source ones (see Section 5.3).

5.1 Modelling State Machines

According to the scenario described in Section 2, we assume that in the software development lifecycle, the designer is interested to have a behavioral description of the system by means of hierarchical state machine, whereas a test expert produces non-hierarchical state machine models. The hierarchical and non-hierarchical state machine matamodels (respectively HSM and NHSM) are given by means of their Ecore representation within the EMF framework. Then a hierarchical state machine model conforming to the HSM metamodel can be specified as the model reported in the left-hand side of Figure 5. Models can be specified with graphical and/or concrete syntaxes depending on the tool availability for the considered modeling language. In our case, the adopted syntaxes for specifying models do not affect the overall transformation approach since models are manipulated by considering their abstract syntaxes.

5.2 Specifying and Applying the HSM2NHSM Model Transformation

Starting from the definition of the involved metamodels, the JTL transformation is specified according to the QVT-like syntax described in Section 4.2 (see Listing 1.5). By referring to the Figure 4, the *JTL program*, the *source* and *target metamodel*s and the *source model* have been created and need to be translated in their ASP encoding in order to be executed from the transformation engine. The corresponding ASP encodings are automatically produced by the mechanism illustrated in Section 4. In particular, the ASP encoding of both source model and source and target metamodels is generated according to the Listing 1.2 and 1.1, while the JTL program is translated to the corresponding ASP program (see Listing 1.3).



```
sampleHSM.xmi
  platform:/resource/StateMachineCaseStudy/models/sampleHSM.xm
    State Machine
      State Begin Installation
      Initial State init
      Final State finish
      Composite State Install Software
        State Entry
        State Ask installing questions
        State Install
      Composite State Disk Error
        State Fix disk do
        State Entry
        State Ask/confirm action
        State Show question dialog do
      Composite State Memory Low
        State Entry
        State Show question dialog do
        State Ask/confirm action
      Transition init
      Transition StartInstall()
      Transition entry/ask
      Transition ask/install
      Transition install/end
      Transition DiskError()
      Transition entry/fix
      Transition fix/question
      Transition question/ask
      Transition Stop()
      Transition OutOfMemory()
      Transition entry/question
      Transition question/ask
      Transition [alternative=continue]
      Transition [alternative=stop]
```

```
sampleNHSM.xmi
  platform:/resource/StateMachineCaseStudy/models/sampleNHSM.xmi
    State Machine
      State Begin Installation
      Initial State init
      Final State finish
      State Install Software
      State Disk Error
      State Memory Low
      Transition init
      Transition StartInstall()
      Transition install/end
      Transition DiskError()
      Transition Stop()
      Transition OutOfMemory()
      Transition [alternative=continue]
      Transition [alternative=stop]
```

Properties

Property	Value
Cost	0
Effect	Stop()
Owning State Machine	State Machine
Source	State Disk Error
Target	Final State finish
Trigger	Stop()

Properties

Property	Value
Effect	Stop()
Source	State Ask/confirm action
Target	Final State finish
Trigger	Stop()

Fig. 5. HSM source model and the correspondent NHSM target model

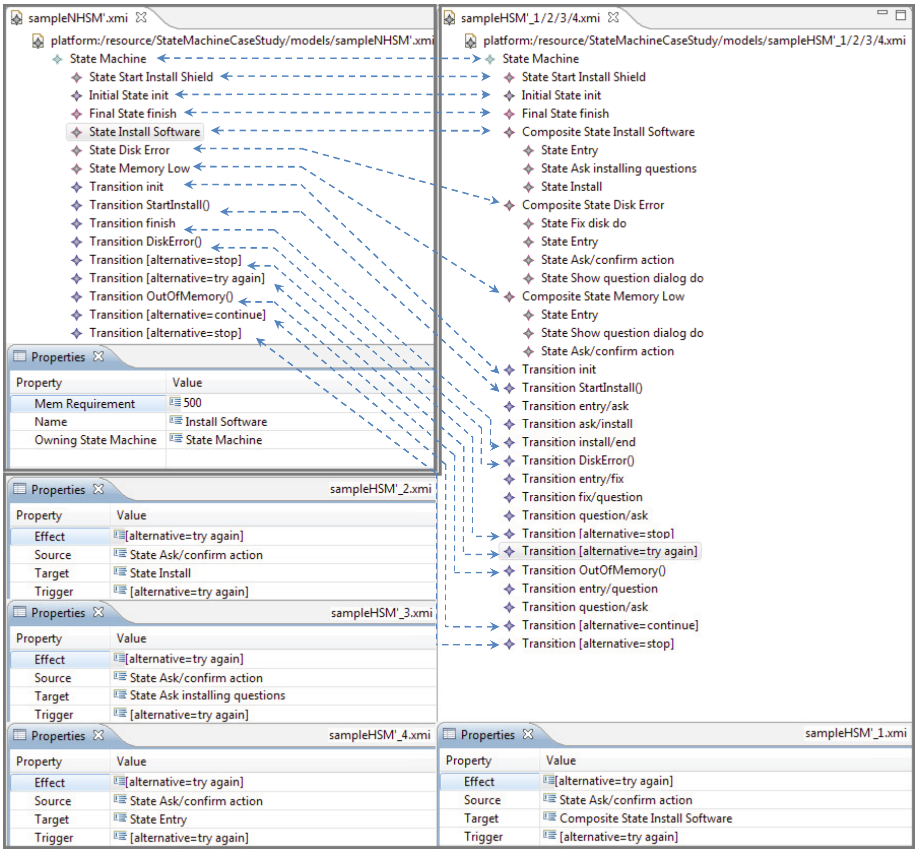
After this phase, the application of the HSM2NHSM transformation on *sampleHSM* generates the corresponding *sampleNHSM* model as depicted in the right part of Figure 4. Note that, by re-applying the transformation in the backward direction it is possible to obtain again the *sampleHSM* source model. The missing sub-states and the transitions involving them are restored by means of trace information.

5.3 Propagating Changes

Suppose that in a refinement step the designer needs to manually modify the generated target by the changes described in Section 2 (see Δ changes depicted in Figure 2), that is:

1. renaming the initial state from `Begin Installation` to `Start Install shield`;
2. adding the alternative `try again` to the state `Disk Error` to come back to `Install software`;
3. changing the attributes related to memory requirements (`m=500`) in the state `Install software` and cost (`c=200`) of the transition from `Memory low` to `Install software`.

The target model including such changes (*sampleNHSM'*) is shown in the left part of the Figure 6. If the transformation HSM2NHSM is applied on it, we expect changes to be propagated on the source model. However, due to the different expressive power of the



```
sampleNHSM'.xmi
  platform:/resource/StateMachineCaseStudy/models/sampleNHSM'.xmi
    State Machine
      State Start Install Shield
      Initial State init
      Final State finish
      State Install Software
      State Disk Error
      State Memory Low
      Transition init
      Transition StartInstall()
      Transition finish
      Transition DiskError()
      Transition [alternative=stop]
      Transition [alternative=try again]
      Transition OutOfMemory()
      Transition [alternative=continue]
      Transition [alternative=stop]
```

Properties

Property	Value
Mem Requirement	500
Name	Install Software
Owning State Machine	State Machine

```
sampleHSM'_1/2/3/4.xmi
  platform:/resource/StateMachineCaseStudy/models/sampleHSM'_1/2/3/4.xmi
    State Machine
      State Start Install Shield
      Initial State init
      Final State finish
      Composite State Install Software
        State Entry
        State Ask installing questions
        State Install
      Composite State Disk Error
        State Fix disk do
        State Entry
        State Ask/confirm action
        State Show question dialog do
      Composite State Memory Low
        State Entry
        State Show question dialog do
        State Ask/confirm action
      Transition init
      Transition StartInstall()
      Transition entry/ask
      Transition ask/install
      Transition install/end
      Transition DiskError()
      Transition entry/fix
      Transition fix/question
      Transition question/ask
      Transition [alternative=stop]
      Transition [alternative=try again]
      Transition OutOfMemory()
      Transition entry/question
      Transition question/ask
      Transition [alternative=continue]
      Transition [alternative=stop]
```

Properties — sampleHSM'_2.xmi

Property	Value
Effect	[alternative=try again]
Source	State Ask/confirm action
Target	State Install
Trigger	[alternative=try again]

Properties — sampleHSM'_3.xmi

Property	Value
Effect	[alternative=try again]
Source	State Ask/confirm action
Target	State Ask installing questions
Trigger	[alternative=try again]

Properties — sampleHSM'_4.xmi

Property	Value
Effect	[alternative=try again]
Source	State Ask/confirm action
Target	State Entry
Trigger	[alternative=try again]

Properties — sampleHSM'_1.xmi

Property	Value
Effect	[alternative=try again]
Source	State Ask/confirm action
Target	Composite State Install Software
Trigger	[alternative=try again]

Fig. 6. The modified NHSM target model and the correspondent HSM source models

involved metamodels, target changes may be propagated in a number of different ways, thus making the application of the reverse transformation to propose more solutions. The generated sources, namely *sampleHSM'_1/2/3/4* can be inspected through Figure 6: the change (1) has been propagated renaming the state to `Start Install shield`; the change (2) gives place to a non-bijective mapping and for this reason more than one model is generated. As previously said, the new transition can be equally targeted to each one of the nested states within `Install Software` as well as to the super state itself (see the properties *sampleHSM'_1/2/3/4* in Figure 6). For example, as visible in the property of the transition, *sampleHSM'_1* represents the case in which the transition is targeted to the composite state `Install Software`; finally, the change (3) is out of the domain of the transformation. In this case, the new values for memory and cost are not propagated on the generated source models.

Even in this case, if the transformation is applied on one of the derived *sampleHSM'* models, the appropriate *sampleNHSM'* models including all the changes are generated. However, this time the target will preserve information about the chosen *sampleHSM'* source model, thus causing future applications of the backward transformation to generate only *sampleHSM'*.

With regard to the performances of our approach, we performed no formal study on its complexity yet, since that goes beyond the scope of this work; however, our observations showed that the time required to execute each transformation in the illustrated case study is more than acceptable since it always took less than one second. In the general case, when there are a lot of target alternative models the overall performance of the approach may degrade.

6 Related Work

Model transformation is intrinsically difficult, and its nature poses a number of obstacles in providing adequate support for bidirectionality and change propagation [14]. As a consequence, despite a number of proposals, in general they impose relevant restrictions of the characteristics of the involved transformations. For instance, approaches like [20,21,22,23] require the mappings to be total, while [20,21,5] impose the existence of some kind of bijection between the involved source and target. Such comparison is discussed in [14].

Stevens [7] discusses bidirectional transformations focusing on basic properties which such transformations should satisfy. In particular, (i) *correctness* ensures a bidirectional transformation does something useful according to its consistency relation, (ii) *hippocraticness* prevents a transformation from does something harmful if nether model is modified by users, (iii) finally, *undoability* is about the ability whether a performed transformation can be canceled. The paper considers QVT-R as applied to the specification of bidirectional transformation and analyze requirements and open issues. Furthermore, it points out some ambiguity about whether the language is supposed to be able to specify and support non-bijective transformations.

Formal works on bidirectional transformations are based on graph grammars, especially triple graph grammars (TGGs) [15]. These approach interpret the models as graphs and the transformation is executed by using graph rewriting techniques. It is possible to specify non-bijective transformations; however, attributes are not modeled as part of the graphs.

In [5] an attempt is proposed to automate model synchronization from model transformations. It is based on QVT Relations and supports concurrent modifications on both source and target models; moreover, it propagates both sides changes in a non destructive manner. However, some issues come to light: in fact, conflicts may arise when merging models obtained by the propagation with the ones updated by the user. Moreover, it is not possible to manage manipulations that makes the models to go outside the domain of the transformation.

The formal definition of a round-trip engineering process taking into account the non-totality and non-injectivity of model transformations is presented in [14]. The valid modifications on target models are limited to the ones which do not induce backward mappings out the source metamodel and are not operated outside the transformation domain. The proposal discussed in this paper is capable also to manage target changes inducing extensions of the source metamodel by approximating the exact source as a set of models; i.e., the set of possible models which are the closest to the ideal one from which to generate the previously modified model.

In [16] the author illustrates a technique to implement a change propagating transformation language called PMT. This work supports the preservation of target changes by back propagating them toward the source. On the one hand, conflicts may arise each time the generated target should be merged with the existing one; on the other hand, the back propagation poses some problems related to the invertibility of transformations, respectively.

Declarative approaches to model transformations offer several benefits like for example implicit source model traversal, automatic traceability management, implicit target object creation, and implicit rule ordering [9,24]. A number of interesting applications is available, varying from incremental techniques [25] to the automation of transformation specifications by means of the inductive construction of first-order clausal theories from examples and background knowledge [26]. One of the closest works to our approach is xMOF [27]. It aims to provide bidirectionality and incremental transformations by means of an OCL constraint solving system which enables the specification of model transformations. However, transformation developers have no automation support to derive constraints from source/target metamodel conformance and transformation rules, which may make their task hard in case of complex mappings [16]. Moreover, it has not been clarified how such technique would deal with multiple choices and the requirements described in Sect. 3.

We already introduced in [28] an ASP based transformation engine enabling the support for partial and non injective mappings. However, the inverse transformation has to be given by the developer and a valid round-trip process is not guaranteed, as already discussed throughout the paper. For this purpose, we introduced JTL, a transformation language specifically designed for supporting change propagation with model approximation capabilities.

7 Conclusion and Future Work

Bidirectional model transformations represent at the same time an intrinsically difficult problem and a crucial mechanism for keeping consistent and synchronized a number of related models. In this paper, we have refined an existing definition of bidirectional model transformations (see [7]) in order to better accommodate non-bijectivity and model approximation. In fact, existing languages fail in many respect when dealing with non-bijectivity as in many cases its semantic implications are only partially explored, as for instance in bidirectional QVT transformations whose standard does not even clarify whether valid transformations are only bijective transformations. Naturally, non-bijective transformations can possibly map a number of source models to the same target model, therefore whenever a target model is manually modified, the changes must be back propagated to the related source models.

This paper presented the Janus Transformation Language (JTL), a declarative model transformation approach tailored to support bidirectionality and change propagation which conforms to the requirements presented in Section 3. JTL is able to map a model into a set of semantically related models in both forward and backward directions, moreover whenever modifications to a target model are making it unreachable from the transformation an approximation of the ideal source model is inferred. To the best of our knowledge these characteristics are unique and we are not aware of any other language which deals with non-bijectivity and model approximation in a similar way. The expressivity and applicability of the approach has been validated against a relevant benchmark, i.e., the transformation among hierachical and non-hierarchical state machines as prescribed by [10]. The language has been given abstract and concrete syntax and its semantics is defined in terms of Answer Set Programming; a tool is available which renders the language interoperable with EMF [7].

As future work we plan to extend the framework with a wizard helping the architect to make decisions among proposed design alternatives. The alternatives are initially partitioned, constrained, abstracted, and graphically visualized to the user. Then, when decisions are made, they are stored and used to drive subsequent decisions. Another interesting future work is to investigate about incremental bidirectional model transformations. If a developer changes one model, the effects of these changes should be propagated to other models without re-executing the entire model transformation from scratch. In the context of bidirectional transformation it should coexist with the ability to propagate changes in both the directions but preserves information in the models and, in our case, also allows the approximation of models.

References

1. Schmidt, D.: Guest Editor's Introduction: Model-Driven Engineering. Computer 39(2), 25–31 (2006)
2. Bézivin, J.: On the Unification Power of Models. Jour. on Software and Systems Modeling (SoSyM) 4(2), 171–188 (2005)
3. Sendall, S., Kozaczynski, W.: Model Transformation: The Heart and Soul of Model-Driven Software Development. IEEE Software 20(5), 42–45 (2003)
4. Stevens, P.: A Landscape of Bidirectional Model Transformations. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007. LNCS, vol. 5235, pp. 408–424. Springer, Heidelberg (2008)
5. Xiong, Y., Song, H., Hu, Z., Takeichi, M.: Supporting parallel updates with bidirectional model transformations. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 213–228. Springer, Heidelberg (2009)
6. Object Management Group (OMG): MOF QVT Final Adopted Specification, OMG Adopted Specification ptc/05-11-01 (2005)
7. Stevens, P.: Bidirectional model transformations in qvt: semantic issues and open questions. Software and Systems Modeling 8 (2009)
8. Steven Witkop: MDA users' requirements for QVT transformations. OMG document 05-02-04 (2005)
9. Czarnecki, K., Helsen, S.: Feature-based Survey of Model Transformation Approaches. IBM Systems J. 45(3) (June 2006)
10. Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.F.: Bidirectional Transformations: A Cross-Discipline Perspective. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 260–283. Springer, Heidelberg (2009)
11. Gelfond, M., Lifschitz, V.: The Stable Model Semantics for Logic Programming. In: Kowalski, R.A., Bowen, K. (eds.) Proceedings of the Fifth Int. Conf. on Logic Programming, pp. 1070–1080. The MIT Press, Cambridge (1988)

[7] `http://www.di.univaq.it/romina.eramo/JTL`

12. Chen, K., Sztipanovits, J., Abdelwalhed, S., Jackson, E.: Semantic Anchoring with Model Transformations. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 115–129. Springer, Heidelberg (2005)
13. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The dlv system for knowledge representation and reasoning (2004)
14. Hettel, T., Lawley, M., Raymond, K.: Model Synchronisation: Definitions for Round-Trip Engineering. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 31–45. Springer, Heidelberg (2008)
15. Konigs, A., Schurr, A.: Tool Integration with Triple Graph Grammars - A Survey. Electronic Notes in Theoretical Computer Science 148, 113–150 (2006)
16. Tratt, L.: A change propagating model transformation language. Journal of Object Technology 7(3), 107–126 (2008)
17. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
18. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.: Eclipse Modeling Framework. Addison-Wesley, Reading (2003)
19. Bézivin, J., Jouault, F., Rosenthal, P., Valduriez, P.: Modeling in the Large and Modeling in the Small. In: Aßmann, U., Liu, Y., Rensink, A. (eds.) MDAFA 2003. LNCS, vol. 3599, pp. 33–46. Springer, Heidelberg (2005)
20. Van Paesschen, E., Kantarcioglu, M., D'Hondt, M.: SelfSync: A Dynamic Round-Trip Engineering Environment. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 633–647. Springer, Heidelberg (2005)
21. Giese, H., Wagner, R.: Incremental Model Synchronization with Triple Graph Grammars. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 543–557. Springer, Heidelberg (2006)
22. Mu, S.-C., Hu, Z., Takeichi, M.: An Injective Language for Reversible Computation. In: Kozen, D., Shankland, C. (eds.) MPC 2004. LNCS, vol. 3125, pp. 289–313. Springer, Heidelberg (2004)
23. Foster, J., Greenwald, M., Moore, J., Pierce, B., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. 29(3) (2007)
24. Mens, T., Gorp, P.V.: A Taxonomy of Model Transformation. Electr. Notes Theor. Comput. Sci. 152, 125–142 (2006)
25. Hearnden, D., Lawley, M., Raymond, K.: Incremental Model Transformation for the Evolution of Model-Driven Systems. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 321–335. Springer, Heidelberg (2006)
26. Balogh, Z., Varró, D.: Model transformation by example using inductive logic programming. Software and Systems Modeling (2009)
27. Compuware and Sun: XMOF queries, views and transformations on models using MOF, OCL and patterns, OMG Document ad/2003-08-07 (2003)
28. Cicchetti, A., Di Ruscio, D., Eramo, R.: Towards Propagation of Changes by Model Approximations. In: 10th IEEE Int. Enterprise Distributed Object Computing Conf. Workshops (EDOCW 2006), vol. 0, p. 24 (2006)

Software Engineering and the Semantic Web: A Match Made in Heaven or in Hell?

Abraham Bernstein*

Dynamic and Distributed Information Systems Group
Department of Informatics, University of Zürich
Binzmühlestrasse 14, 8050 Zürich, Switzerland
lastname@ifi.uzh.ch
http://www.ifi.uzh.ch/ddis/bernstein.html

Abstract. The Semantic Web provides models and abstractions for the distributed processing of knowledge bases. In Software Engineering endeavors such capabilities are direly needed, for ease of implementation, maintenance, and software analysis.

Conversely, software engineering has collected decades of experience in engineering large application frameworks containing both inheritance and aggregation. This experience could be of great use when, for example, thinking about the development of ontologies.

These examples—and many others—seem to suggest that researchers from both fields should have a field day collaborating: On the surface this looks like a match made in heaven. But is that the case?

This talk will explore the opportunities for cross-fertilization of the two research fields by presenting a set of concrete examples. In addition to the opportunities it will also try to identify cases of fools gold (pyrite), where the differences in method, tradition, or semantics between the two research fields may lead to a wild goose chase.

Keywords: Software Engineering, Semantic Web, Software Analysis, Knowledge Representation, Statistics, Ontologies.

The Semantic Web at Use for Software Engineering Tasks and Its Implications

Semantic Web technologies have successfully been used in recent software engineering research. Dietrich [2], for example, proposed an OWL[1] ontology to model the domain of software design patterns [4] to automatically generate documentation about the patterns used in a software system. With the help of

* I would like to thank the participants of the Semantic Web Software Engineering Workshop (SWSE) at the International Semantic Web Conference (ISWC) 2009 for their valuable comments on an earlier version of this presentation. Partial support for some of the work presented in this talk was provided by Swiss National Science Foundation award number 200021-112330.

[1] http://www.w3.org/TR/owl2-overview/

this ontology, the presented pattern scanner inspects the abstract syntax trees (AST) of source code fragments to identify the patterns used in the code.

Highly related is the work of Hyland-Wood *et al* [6], in which the authors present an OWL ontology of *Software Engineering Concepts (SECs).* Using SEC, it is possible to enable language-neutral, relational navigation of software systems to facilitate software understanding and maintenance. The structure of SEC is very similar to the language structure of Java and includes information about classes and methods, test cases, metrics, and requirements of software systems. Information from versioning and bug-tracking systems is, however, not modeled in SEC.

Both, Mäntylä *et al* [7] and Shatnawi and Li [8] carried out an investigation of *code smells* [3] in object-oriented software source code. While the study of Mäntylä additionally presented a taxonomy (i.e., an ontology) of smells and examined its correlations, both studies provided empirical evidence that some code smells can be linked with errors in software design.

Happel *et al* [5] presented the *KOntoR* approach that aims at storing and querying metadata about software artifacts in a central repository to foster their reuse. Furthermore, various ontologies for the description of background knowledge about the artifacts such as the programming language and licensing models are presented. Also, their work includes a number of SPARQL queries a developer can execute to retrieve particular software fragments which fit a specific application development need.

More recently Tappolet *et al* [9] presented EvoOnt,[2] a set of software ontologies and data exchange formats based on OWL. EvoOnt models software design, release history information, and bug-tracking meta-data. Since OWL describes the semantics of the data, EvoOnt (1) is easily extendible, (2) can be processed with many existing tools, and (3) allows to derive assertions through its inherent Description Logic reasoning capabilities. The contribution of their work is that it introduces a novel software evolution ontology that vastly simplifies typical software evolution analysis tasks. In detail, its shows the usefulness of EvoOnt by repeating selected software evolution and analysis experiments from the 2004-2007 Mining Software Repositories Workshops (MSR). The paper demonstrates that if the data used for analysis were available in EvoOnt then the analyses in 75% of the papers at MSR could be reduced to one or at most two simple queries within off-the-shelf SPARQL[3] tools. In addition, it presents how the inherent capabilities of the Semantic Web have the potential of enabling new tasks that have not yet been addressed by software evolution researchers, e.g., due to the complexities of the data integration.

This semi-random[4] selection *examples clearly illustrate the usefulness of Semantic Web technologies for Software Enginering.*

Conversely, decades of experience in engineering large application frameworks containing both inheritance and aggregation provides a sound foundation for the usefulness of *Software Engineering techniques to Semantic Web research.*

[2] `http://www.ifi.uzh.ch/ddis/research/evoont/`

[3] `http://www.w3.org/TR/rdf-sparql-query/`

[4] All of the examples are focused on Semantic Web enabled software analysis.

These insights seem to suggest that researchers from the fields should have a field day collaborating: On the surface this looks like a match made in heaven. But is that the case?

The main purpose of this talk is to *identify the opportunities for cross-fertilization between the two research fields* by presenting a set of concrete examples.

To contrast these opportunities it will also try to *identify possible barriers and/or impediments to collaboration,* where the differences in method, tradition, or semantics between the two research fields may lead to a wild goose chase.

References

1. Demeyer, S., Tichelaar, S., Steyaert, P.: FAMIX 2.0—the FAMOOS inf. exchange model. Tech. rep., University of Berne, Switzerland (1999), `http://www.iam.unibe.ch/~famoos/FAMIX/Famix20/Html/famix20.html`
2. Dietrich, J., Elgar, C.: A formal description of design patterns using OWL. In: Proceedings of the Australian Software Engineering Conference, Brisbane, Australia (2005)
3. Fowler, M.: Refactoring. Addison-Wesley, Reading (1999)
4. Gamma, E., Helm, R., Johnson, R.E.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley Longman, Amsterdam (1995)
5. Happel, H.-J., Korthaus, A., Seedorf, S., Tomczyk, P.: KOntoR: An Ontology-enabled Approach to Software Reuse. In: Proceedings of the 18th Internatinal Conference on Software Engineering and Knowledge Engineering (SEKE), San Francisco, CA (2006)
6. Hyland-Wood, D., Carrington, D., Kapplan, S.: Toward a Software Maintenance Methodology using Semantic Web Techniques. In: Proceedings of the 2nd ICSM International Workshop on Software Evolvability (SE), pp. 23–30 (2006)
7. Mäntylä, M., Vanhanen, J., Lassenius, C.: A Taxonomy and an Initial Empirical Study of Bad Smells in Code. In: Proceedings of the International Conference on Software Maintenance (ICSM), pp. 381–384 (2003)
8. Shatnawi, R., Li, W.: A Investigation of Bad Smells in Object-Oriented Design Code. In: Proceedings of the 3rd International Conference on Information Technology: New Generations, pp. 161–165 (2006)
9. Tappolet, J., Kiefer, C., Bernstein, A.: Semantic web enabled software analysis. Journal of Web Semantics: Science, Services and Agents on the World Wide Web 8 (July 2010)

A Unified Format for Language Documents

Vadim Zaytsev and Ralf Lämmel

Software Languages Team
Universität Koblenz-Landau
Universitätsstraße 1
56072 Koblenz
Germany

Abstract. We have analyzed a substantial number of language documentation artifacts, including language standards, language specifications, language reference manuals, as well as internal documents of standardization bodies. We have reverse-engineered their intended internal structure, and compared the results. The Language Document Format (LDF), was developed to specifically support the documentation domain. We have also integrated LDF into an engineering discipline for language documents including tool support, for example, for rendering language documents, extracting grammars and samples, and migrating existing documents into LDF. The definition of LDF, tool support for LDF, and LDF applications are freely available through SourceForge.

Keywords: language documentation, language document engineering, grammar engineering, software language engineering.

1 Introduction

Language documents form an important basis for software language engineering activities because they are primary references for the development of grammar-based tools. These documents are often viewed as static, read-only artifacts. We contend that this view is outdated. Language documents contain formalized elements of knowledge such as grammars and code examples. These elements should be checked and made available for the development of grammarware. Also, language documents may contain other formal statements, e.g., assertions about backward compatibility or the applicability of parsing technology. Again, such assertions should be validated in an automated fashion. Furthermore, the maintenance of language documents should be supported by designated tools for the benefit of improved consistency and traceability. In an earlier publication, a note for ISO [KZ05], we have explained why a language standardization body needs grammar engineering (or document engineering).

This paper presents a data model (say, metamodel or grammar) for developing language documents. Upon analyzing and reverse-engineering a wide range of language documents, which included international ISO-approved standards and vendor-specific 4GL manuals, we have designed a general format for language documents, the Language Document Format (LDF), which supports the documentation of languages in a domain-specific manner. We have integrated LDF with a formalism for syntax definition that we designed and successfully utilized in previous work [LZ09, LZ10, Zay10b].

We have integrated LDF also with existing tools and methods for grammar engineering from our previous work; see the grammar life cycle at the top of Figure 1 for an illustration. Furthermore, we have added LDF-specific tools, and begun working towards a discipline of *language document engineering.* There is support for creating, rendering, testing, and transforming language documents; see the document life cycle at the bottom of Figure 1 for an illustration. Given the new format LDF, it is particularly important that there are document extractors so that one can construct consistent LDF documents from existing language documents.

In this paper, we will be mainly interested in LDF as the format for language documents, and the survey that supports the synthesis of the LDF format. The broader discussion of language document engineering is only sketched here. For instance, most aspects of rich tool support are deferred to substantial future work efforts.

LDF can be seen as an application of literate programming [Knu84] ideology to the domain of language documentation: we aim to have one artifact that is both readable and executable. By "readable" we mean its readability, understandability and information retrievability qualities. By "executable" we assume a proper environment such as a compiler compiler (for parser definitions) or a web browser (for hyperlinked grammars). LDF provides us with a data model narrowly tailored to the domain; it allows us to focus on one baseline artifact which is meant for both understanding and formal specification. Other artifacts such as grammars, test sets, web pages, language manuals and change documents are considered secondary in that they are to be generated or programmed. The full realization of this approach relies on a transformation language for language documents that we will briefly discuss.

Summary of contributions

- We have analyzed a substantial number of language documentation artifacts, including language standards, specifications and manuals of languages such as BNF dialects, C, C++, C#, Cobol dialects, Fortran, 4GLs, Haskell, Jovial, Python, SDF, XML, and other data modeling languages. Company-specific internal documents and software engineering books that document a software language (e.g., [GHJV95] with the well-known design patterns) were also researched. The objective of the analysis was to identify domain concepts and structuring principles of language documentation.
- We have designed the Language Document Format (LDF) to specifically support the documentation domain, and to make available language documents to language document engineering.

Validation

We have applied LDF to a number of language documentation problems, but a detailed discussion of such problems is not feasible in this paper for space

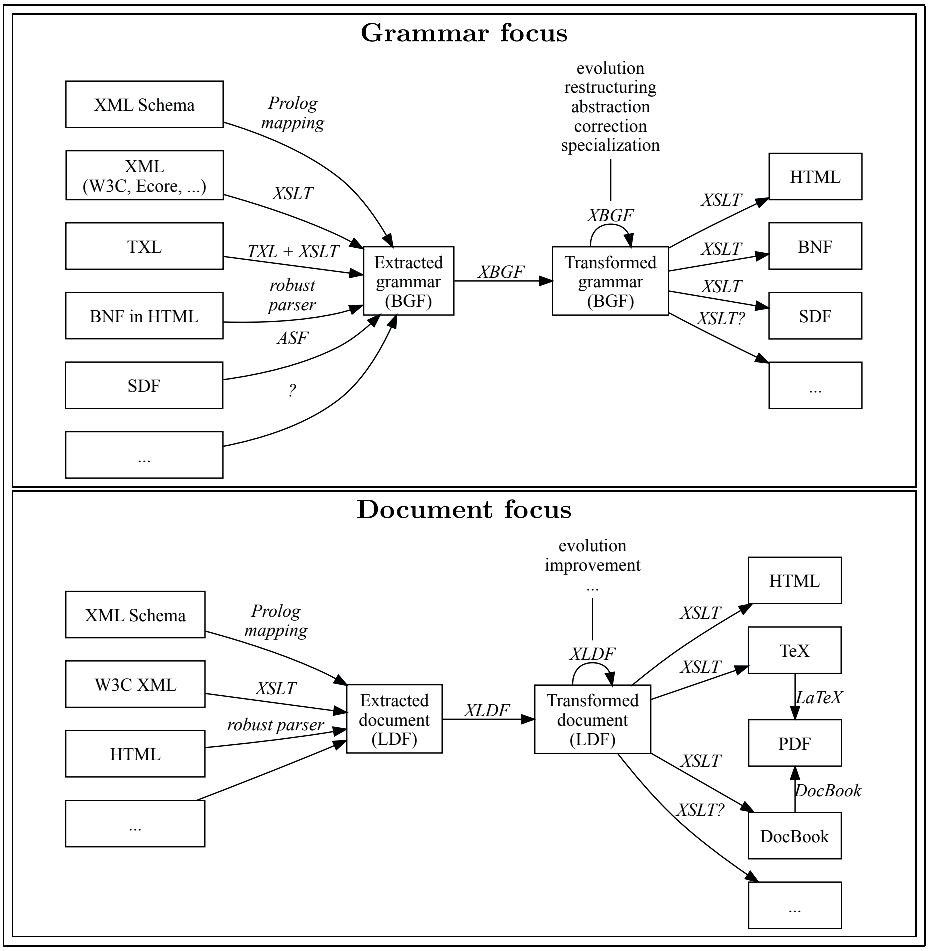


Fig. 1. Megamodels related to language document engineering. At the top, we see the life cycle of grammar extraction, recovery, and deployment. Grammars are extracted from existing software artifacts on the left, and represented in the unified format BGF. Grammars may then be subject to transformation using the XBGF transformation language. Parsers, browsable grammars, and other "executable" artifacts are delivered on the right. Such grammar engineering feeds into language document engineering. At the bottom, we see the life cycle of language document extraction, language (document) evolution, generation of end-user documents, extraction of grammars and test suites. Non-LDF documents can be converted to LDF through the extraction shown on the left. Document transformation may be needed for very different reasons, e.g., structure recovery or language evolution; see the reference to XLDF, which is the transformation language for LDF.

reasons. For instance, we have applied language document engineering systematically to the documentation of XBGF—the transformation language for BGF grammars which is used extensively in our work on grammar convergence; the outcome of this case study is available online [Zay09]. In the current paper, we briefly consider mapping W3C XML to LDF, specifically W3C's XPath standard; this case study is available online, too [W3C]. In the former case, we rely on a document extractor that processes XSD schemata in a specific manner. In the latter case, the extractor maps W3C's XML Spec Schema to LDF.

More generally, the SourceForge project "Software Processing Language Suite" (SLPS)[1] hosts the abovementioned two case studies, the LDF definition, tool support for LDF, other LDF applications, and all other grammars and tools mentioned in this paper and our referenced, previous work. For instance, we refer to the SLPS Zoo[2], which contains a collection of grammars that we extracted from diverse language documents. The next step would be to properly LDF-enable all these documents.

Road-map

The rest of the paper is organized as follows. §2 discusses the state of the art in language documentation as far as it affects our focus on a format for language documents and its role in language document engineering. §3 identifies the concepts of language documentation as they are to be supported by a unified format for language documents, and as they can be inferred, to some extent, from existing language documents. §4 describes the Language Document Format (LDF) in terms of the definitional grammar for LDF. It also provides a small scenario for language document transformation. §5 discusses related work (beyond the state of the art section). §6 concludes the paper.

2 State of the Art in Language Documentation

As a means of motivation for our research on a unified format for language documents, let us study the state of the art in this area. The bottom line of this discussion is that real-world language documents are engineered at a relatively low level of support for the *language* documentation domain.

2.1 Background on Language Standardization

In practice, all mainstream languages are somehow standardized; the standard of a mainstream language would need to be considered the primary language document. For instance, the typical standard for a programming language entails grammar knowledge and substantial textual parts for the benefit of understanding the language.

Let us provide some background on language standardization. In particular, we list standardization bodies, and we discuss some of the characteristics of

[1] SLPS project, `slps.sf.net`
[2] SLPS Zoo, `slps.sf.net/zoo`

language standards. Standardization bodies that produce, maintain and distribute language standards, are, among others:

- American National Standards Institute (ANSI, since 1918), `ansi.org`
- European Computer Manufacturers Association (ECMA, since 1961), `ecma-international.org`
- Institute of Electrical and Electronics Engineers Standards Association (IEEE-SA, since 1884), `standards.ieee.org`
- International Electrotechnical Commission (IEC, since 1906), `iec.ch`
- International Organization for Standardization (ISO, since 1947), `open-std.org`
- International Telecommunication Union (ITU, since 1865), `itu.int`
- Internet Engineering Task Force (IETF, since 1986), `ietf.org`
- Object Management Group (OMG, since 1989), `omg.org`
- Organization for the Advancement of Structured Information Standards (OASIS, since 1993), `oasis-open.org`
- Website Standards Association (WSA, since 2006), `websitestandards.org`
- World Wide Web Consortium (W3C, since 1994), `w3.org`

A language specification (programming language standard) is a complex document that may consist of hundreds of pages: the latest COBOL standard, ISO/IEC 1989:2002 [ISO02], has more than 800 pages; the latest C [ISO05] and C# [ECM06] standards contain over 500 pages each, C++ draft is already well over 1100 pages [ISO07]. It has not always been like that. For example, the Algol 60 standard [BBG$^+$63] is not much longer than 30 pages, and yet, it claimed to contain a complete definition of the language. However, as programming languages evolve, their specifications grow in size. Also, the complicated structure of modern language documents reflects the complicated structure of modern programming languages and the associated ecosystems.

2.2 The Language Documentation Challenge

Writing and maintaining a quality language document and keeping it consistent is as complex as writing and maintaining a large software system—these processes have a lot in common.

Defining a programming language in a standardized specification is often considered as a process that is executed just once. The dynamic and evolving nature of programming languages is frequently underestimated and overlooked [Fav05]. Not only software itself, but programming languages that are used to make it, evolve over time. This process usually comes naturally in the sense that the first version of a language does not have all the features desired by its creator. Also, new requirements may be discovered for a language, and hence, the language needs to be extended or revised. However, it is desirable for that process to be guided and controlled for the sake of the quality of resulting specifications.

There are tools like parsers and compilers whose development is based on a language specification. Inconsistencies in the language documents may lead to non-conformant language tools; such inconsistencies certainly challenge the effective use of the language documents. Languages need to evolve, and hence, it should be easy enough to evolve language documents. However, with the current practice of language standardization, evolution of language documents may be too ad-hoc, error-prone and labor-intensive; see, for example, our previous study on the language documentation for the Java Language Specification [LZ10].

Overall, it is difficult to support language evolution for programming languages or software languages that are widely used. We contend that a systematic approach to language documents is an important contribution to a reliable and scalable approach to language evolution in practice.

2.3 Language Documentation Approaches

In practice, language documents are created and maintained with various *technologies*, e.g., LaTeX [ISO08], HTML [BBC+07], Framemaker [ISO02], home-grown DSLs based on the language being defined in the document [Bru05], XML Schema [Zay09], DITA, DocBook. The creation and maintenance of language document is also regulated by *practices* of design committees and standardization bodies or simply language document editors. The practices are often constrained by the technologies (or vice versa). We make an attempt to organize technologies and practices. To this end, we identify different language documentation approaches.

The text- and presentation-oriented approach considers a language specification as a text document subject to text editing. The editor manually adds text to the document, manages section structure, moves around paragraphs and other units of text, performs layout and formatting operations. Typically, the text is meant to be immediately ready for presentation—perhaps even based on WYSIWYG.

The course of action for an editor of a language document is often described in a separate "change document" that is created before the actual change takes place or directly after it. The change document comprises a list of intended modifications. Once the editing process reaches a certain milestone, a new "revision" is delivered and stored in the repository. Once all the modifications approved by the language design committee are brought upon the main document, a new "version" is delivered and officially distributed within the terms of its license. This approach tends to utilize programs like Adobe Framemaker (ISO/IEC JTC1/SC22/WG4[3]), Microsoft Word (Microsoft version of C# [Mic03]), etc. It is also possible to use HTML (early W3C [Rag97]) in such a way that the main document is edited manually and the changes are discussed and/or documented elsewhere.

This approach involves significant low-level text editing. The links between the change documents and the main document revisions often remain unverified. (Versioning and change tracking facilities can be too constraining.) Any structured content that is a part of a language document must be formatted in a way dictated by the medium: e.g., the formulæ can only use the symbols available in the font. It is also common to have several differently organized layers in the infrastructure: e.g., the main document is edited by one person following the

[3] ISO/IEC JTC1/SC22/WG4 — COBOL Standardization Working Group, `http://www.cobolstandard.info/wg4/wg4.html`

instructions in the change document, but the change documents circulate in the form of co-authored Word documents.

The structure-oriented approach operates on documentation domain concepts such as “sections” or “divisions”. The approach may leverage existing editing software to support maintenance activities at the central repository of structured data. The approach also leverages backend tools that produce PDF, LaTeX, and other types of deliverables. An example of such a documentation support system is DocBook [WM99]. It is a mature, well-document, actively used technology. For instance, Microsoft uses DocBook to generate help files for Windows applications.

The separation between the content and its presentation can be sufficient in DocBook and similar systems. However, their orientation on books does not anticipate documents that have several intertwined hierarchies. In a language document, for example, a grammar production that is a part of the corresponding section, is also a part of the complete grammar in the appendix, and should appear there automatically (as opposed to being manually cloned). In principle, one could leverage transformations (such as XSLT for DocBook) for the representation of the evolution of a (language) document. We are not aware of related work of this kind.

The topic-oriented approach operates in terms of “topics” that should be covered in order for the documentation to be complete. The DocBook counterpart in this group of approaches is Darwin Information Typing Architecture (DITA) [OAS07] which was designed specifically for authoring, producing and delivering technical information. IBM uses DITA for their hardware documentation. PDF, HTML, Windows help files and other output formats are possible. DITA is a relatively modern technology (2004 versus 1991 for DocBook), its support is growing, but is not as mature as for DocBook. A more lightweight approach is wiki technology that allows for topics to be left uncovered, showing explicitly which parts of the documentation are intended to be written in the future.

Language documentation is not naturally organized in topics and tasks, and thus is not anticipated by DITA. In principle, it is possible to use DITA to represent our proposed model (LDF). In order to do that, necessary element types—like grammar productions, code examples, notes concerning version differences, optional feature descriptions, possible implementation remarks, language engineering explanations—would need to be defined. Designated backends will also be required. There is no apparent benefit of using DITA, when compared to the XML/XSD-based approach that we chose for LDF’s description.

The XML Spec Schema, available from `http://www.w3.org/2002/xmlspec`, combines elements of structure and topic orientation in a manner that brings us closer to the domain of *language* documentation. The XML Spec Schema is a DTD that is used for some W3C recommendations. It is based on the literate programming tag set SWEB and the text encoding tag set TEI Lite. The Spec Schema covers some elements of the language documentation domain such as tagging facilities for grammar fragments; it does not capture LDF’s rich classification of sections in language documents.

3 Concepts of Language Documentation

As a preparatory step towards introducing LDF, we identify the concepts of the language documentation domain. We set up a control group to this end, and we also illustrate several concepts specifically for one member of the control group: the XPath W3C Recommendation.

3.1 Control Group for the Domain Model

As we have indicated in the introduction, we have consulted a large set of language documents to eventually synthesize a unified format. For reasons of scalability, we have selected a smaller set of documents which we use here to present the results of our reverse-engineering efforts and to prepare the synthesis of a unified format for language documents. The control group of documents has been chosen for its diversity. Table 1 shows some basic metadata about the language documents for the control group. Here is a short description of the control group:

Table 1. Some basic metadata of the standards chosen for the survey

Property	IAL [Bac60]	Jovial [MIL84]	Patterns [GHJV95]	Smalltalk [Sha97]	Informix [IBM03]	C# [ECM06]	MOF [MOF06]	XPath [BBC$^+$07]
Body	ACM	DoD	—	ANSI	IBM	ECMA, ISO	OMG	W3C
Company	IBM	—	Pearson	—	IBM	Microsoft	—	—
Year	1960	1984	1995	1997	2003	2006	2006	2007
Pages	21	158	395	304	1344	548	88	111
Notation	BNF	BNF	UML	BNF	RT	BNF	UML	EBNF

- **IAL** stands for International Algebraic Language that later became known as Algol-58 [Bac60]. It is historically the first programming language document, and as such, it is the first time that the notation for specifying grammar productions was explicitly defined. The majority of all other standards produced over the following decades re-used this notation and extended it.
- **JOVIAL**, or J73 [MIL84] is a Military Standard of 1984, which "has been reviewed and determined to be valid" in 1994. It is approved for use by the Department of the Air Force and is available for use by all other Departments and Agencies of the Department of Defense of USA. The version that was examined in this survey is a result of a second upgrade of the original language. It is less than 200 pages and very strictly composed: basically every section has a syntax, semantics and constraints subsections, with rare notes or examples. A traditional BNF is used for syntax, plain English for semantics.
- ***Design Patterns:*** *Elements of Reusable Object-Oriented Software* [GHJV95] is a well-known book by Erich Gamma et al., which defines 23 well-known design patterns. Since design patterns can be considered a special language, their definition can be considered a language document—and Table 2 only proves that, letting the 400 pages long book's structure fit in the general data model perfectly.

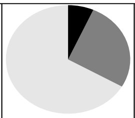
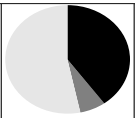
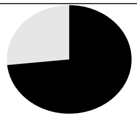
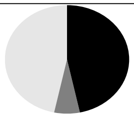
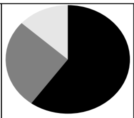
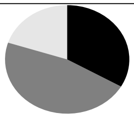
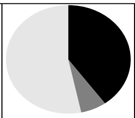
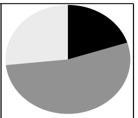
- **ANSI Smalltalk** [Sha97] is an NCITS J20 draft of 1997, 300+ pages long, it describes both the language (ANSI Smalltalk is derived from Smalltalk-80) and the Standard Class Library.
- **Informix** [IBM03] is an **IBM** manual for a proprietary fourth generation language. It exemplifies industrial standards, which are extensively strictly structured, contain minimum extra sections and have impressive volume. Informix specification utilizes "railroad track" syntax diagrams, which can be mapped more or less directly to EBNF.
- **C#** specification [Mic03, ECM06] is both an **ISO** and an **ECMA** standard, yet it was developed entirely within **Microsoft** and only approved by standardization bodies. The ECMA version used for this survey is 550 pages long and very loosely structured, explaining a lot of issues in running text and using liberal sub-sectioning.
- **MOF Core Specification** [MOF06] is a 90-pages long document describing Meta Object Facility. It uses UML and presents the information in a significantly different way, being oriented on diagrams, properties, operations and constraints. However, structuring overall turns out to be similar to conventional (E)BNF-based standards.
- The structure of **XPath W3C Recommendation** [BBC^{+}07] is rather volatile, following the tradition of all other W3C recommendations. Each section contains one or several EBNF formulæ, the definition for a domain concept modeled by it and a body of text organized arbitrarily in lists and subsections.

3.2 Identification of Concepts

The core domain concepts of LDF are these: **synopsis**, **description** (an extended textual definition), **syntax** (associated grammar productions), **constraints** (restricting the use of the construct), **references** (to other language constructs), **relationship** (with other language constructs), **semantics**, **rationale**, **example**, **update** (from the previous language version), **default** (for absent parts). Four additional concepts can occur multiple times: **value** (associated named piece of metadata), **list** (itemized data), **section** (volatile textual content), **subtopic** (structured section).

Table 2 compares the documents from the control group in terms of the domain concepts. The cells in the table are filled with names of the sections, subsections or otherwise identifiable paragraphs in the corresponding documents, unless noted otherwise. The coverage graph shows fully covered parts of LDF in black (represented by section names in table cells), partially covered in gray ("∼" in a table cell means that the information is given but lacks any specific markup) and not covered in white ("—" in a cell means that this kind of information is absent from the language document). Gray concepts are interesting in so far that we face instances of implicit structure which can only be recovered with human intervention or advanced information retrieval techniques in the extraction tool.

Table 2. Mapping language definitions to domain concepts for language documentation

Domain concept	IAL [Bac60]	Jovial [MIL84]	Design Patterns [GHJV95]	Smalltalk [Sha97]	Informix [IBM03]	C# [ECM06]	MOF [MOF06]	XPath [BBC$^+$07]
synopsis	—	~	intent	synopsis	~	~	~	—
description	~	—	motivation	definition	usage	~	—	~
syntax	—[a]	syntax	structure	~	~	~	—	[NN][b]
constraints	—	constraints	applicability	errors	restrictions	~	constraints	~
references	—	—	related patterns	—	references	~	—	~
relationship	—	—	consequences	return value, refinement	related	return type	—	~
semantics	—	semantics	collaborations	—	important	~	semantics	~
rationale	~	notes	implementation	rationale	GLS, ES[c]	note	rationale	note
example	examples	examples	sample code, known uses	—	~	example	—	~
update	—	—	—	—	—	—[d]	changes	—
default	—	—	—	—	note	default values	—	—
value	—	—	also known as	conforms to	—	—	—	—
list	~	—	—	messages, parameters	*terminals*	—	properties	~
plain-section	~	—	—	—	~	~	—	~
subtopic	—	types	participants	—	fields	parameters, methods	operations	functions
Coverage of LDF								

[a] The absence of **syntax** elements means that grammar productions only occur within the designated part of a standard.
[b] All productions in XPath standard are numbered and marked as [1], [2], etc.
[c] GLS — Global Language Support, ES — an IBM Informix database type.
[d] For every version of C#, there is a separate document that summarizes the changes brought to the language.

3.3 Example: The XPath Language Document

The discovery of a language document's structure and underlying domain concepts is a genuine process which we would like to sketch here for one example. We have chosen XPath 1.0 for this purpose—mainly because of its modest size.

The XML Path Language 1.0 specification [CD99] is one of the small standards, it contains only 32 pages in the printed version. We perform a cursory examination of it, trying to locate the domain concepts identified in the previous section:

Synopsis — is not automatically retrievable. We contend that, in some sections, the first sentence seems to serve as a synopsis (e.g., "Every axis has a principal node type.").
Description — if no specific structure can be recovered, we will treat all section content as a description.
Syntax — when we use the XML version of the specification as a source, all grammar productions are easily identifiable by the `<scrap>` tag. A specific parser had to be developed in XSLT to deal with the mix of plain text (e.g., for EBNF metasymbols) and XML tags (e.g., for nonterminal symbols).
Constraints — some of the Notes are mentioning constraints (e.g., "The number function should not be used..."), but they are not automatically distinguishable from other Notes.
References — since all nonterminal names are always annotated with hyperlinks to the corresponding sections, no explicit references are required.
Relationship — there are mentions of relationships, some of which are even inter-documentary (e.g., the `mod` operator is being compared to the `%` operator in ECMAScript and the IEEE 754 remainder operation, but it is impossible to derive them naturally during recovery.
Semantics — is defined in plain English in running text.
Rationale — almost all Notes can be classified as providing rationales. We map them all to rationales at the extraction step. Exceptions would need to be handled by programmed transformations.
Example — as typical for a W3C document, examples sections are inlined, but preceded by the sentences like "for example," or "here are some examples".
Update — XPath 1.0 is the first specification of its kind, which means that it contains no updates.
Default, Value — not found in this standard.
List — found inside the `<ulist>` and `<slist>` tags in the XML version of the document.
Section — Data Model section contains simple subsections.
Subtopic — every function description (the `<proto>` tag) can be treated as a subtopic. They are never long, but still can contain structured information such as lists and examples.

The global structure of the XPath specification is mapped to LDF in a straightforward fashion: for example, specific sections within the `<header>` such as Abstract and Status form a front matter part; `<body>` subsections populate the core part; `<back>` subsections become the back matter part. The mapping is mainly terminological: i.e., Status becomes “scope”, “Introduction” becomes “foreword”, etc. The extracted LDF for XPath 1.0 is available as [W3C].

4 A Unified Format for Language Documents

We will now describe the Language Document Format (LDF)—a unified format for language documents (say, language documentation). Given the motivation of LDF in previous sections, we will focus here on the actual language description for LDF. LDF’s description is available online through the SLPS SourceForge project as `shared/xsd/ldf.xsd`. (LDF’s primary description leverages XML Schema.) This section presents and discusses a full grammar for (current) LDF. The grammar notation we use here is a pretty-printed EBNF dialect called BGF [LZ10, Zay10b, Zay10a], for BNF-like Grammar Format, which should be intuitively comprehensible. For brevity’s sake, some more routine (obvious) format elements are skipped in the discussion.

4.1 Language Document Partitioning

Consider the following productions concerning the **document** top sort and top level sections. For example, a **document** always contains one **document metadata**, and one or more **parts**. Each part also contains a portion of metadata, and consists of sections of various types.

```
document:
       document-metadata part+
document-metadata:
       body? number::string? author::person* topic::string status
         version-or-edition previous::named-link* date::time-stamp
body:
       ansi::ε | ecma::ε | ieee::ε | iso::ε | iso/iet::ε | itu::ε | iec::ε
         | ietf::ε | oasis::ε | omg::ε | wsa::ε | w3c::ε
person:
       name::string affiliation::string? email::string?
status:
       unknown::ε | draft::ε | candidate::ε | proposed::ε | approved::ε
         | revised::ε | obsolete::ε | withdrawn::ε | collection::ε
         | trial::ε | errata::ε | report::ε
version-or-edition:
       version::string | edition::string
named-link:
       title::string version-or-edition? uri::any-uri?
part:
       part-metadata section+
part-metadata:
       id::id? part-role title::string? author::person*
part-role:
       front-matter::ε | core-part::ε | back-matter::ε | annex::ε
```

Most of the structural facets and elements should be self-explanatory. Let us highlight here the mandatory division of each language **document** into **part**s. In this manner, we encourage more structure than a simple list of top-level chapters. Existing documents vary greatly in the order of sections and their presentation. For instance, "conformance" and "references" sections are usually found in the **front matter** between the title page and the core chapters, but in the XPath 1.0 standard [CD99], conformance is the last core chapter, and references form an appendix. LDF's emphasis on parts encourages some grouping among the many sections.

4.2 Top Sections

On the top level, several types of sections can be found. First, there are simple sections that have no or minimal subdivisioning and dedicate themselves to one specific (side) issue—they are commonly found in the front or back matter. They can also describe lexical details, in such case we identify several commonly encountered section roles. Finally, there can be container sections—each of them explains one language construct and presents information in a specifically structured way.

```
section:
        section-metadata section-structure
section-metadata:
        id::id? section-role type? title::string? author::person*
section-role:
        abstract::ε | conformance::ε | compatibility::ε | design-goals::ε
         | outline::ε | foreword::ε | references::ε | scope::ε | index::ε
         | notation::ε | what-is-new::ε | full-grammar::ε | tables-list::ε
         | authors-list::ε | contents::ε | overview::ε | lexical-issue::ε
         | line-continuations::ε | literals::ε | preprocessor::ε
         | tokens::ε | whitespace::ε | glossary::ε | container::ε
type:
        normative::ε | informative::ε
section-structure:
        content::(content-content+)
        placeholder::ε
        subsection+
```

As shown above, the **metadata** of a top section contains a possible **id** that is used to refer to it from elsewhere; the **role** of the section; possibly its **type**; a possible specific **title** (if absent, assumed to be determined by the role); and a possible list of authors (if absent, assumed to be equal to the list of the document authors). If the **id** is missing, one can still use an XPath expression over the document structure to access the section at hand (by its position, title or other distinctive features). However, explicit ids are potentially preferred because of their greater robustness with regard to document evolution.

A **type** relates to the way the content of the section should be treated: for example, an informative code sample is a way to tell the reader how a piece of code would look like, but a normative one can serve as an official test case. **Placeholder**s can be used if the content can be generated automatically: for example, a references section can be written manually in a verbose way with lots of useful annotations, but it can also be just a list of all references occurring in the rest of the specification.

The list of **section role**s was synthesized from the reverse-engineered language documents. The roles should be intuitively understandable, but the concrete wording may vary: a particular **foreword** can be called "introduction" the same way that an **obsolete** standard can be called "rescinded". Furthermore, roles are not exclusively one per document: for example, there can be one "glossary" for the list of definitions and one for the list of abbreviations.

4.3 Inner Sections

As discussed above, the inner sections of the simple top sections are rather unsophisticated, but a container section explaining one syntactic category consistently shows the same set of possible subsections across many manuals and standards that we analyzed. The exact set of container sections depends on the set of categories and is therefore very language dependent.

```
subsection:
        subsection-metadata section-structure
subsection-metadata:
        id::id? subsection-role type? title::string? author::person*
subsection-role:
        synopsis::ε | description::ε | syntax::ε | constraints::ε
         | references::ε | relationship::ε | semantics::ε | rationale::ε
         | example::ε | update::ε | default::ε | value::ε | list::ε
         | plain-section::ε | subtopic::ε
```

Subsections have roles that map directly to the domain concepts already been seen in Table 2.

4.4 Detailed Content

Language documents, especially modern standards, have structured content even at the textual level of a section: hyperlinks, other references, tables, figures, formulæ, lists, inline code fragments are among the most commonly used formatting elements.

```
content-content:
        empty::ε | code::string | para::mixed | list | figure | table
         | formula | sample::(string source::string) | production
list:
        item::mixed+
figure:
        figure-metadata figure-source+
figure-metadata:
        id::id? short-caption::string? caption::string author::person*
```

```
figure-source:
        type::figure-type location::any-uri
figure-type:
        PDF::ε | PostScript::ε | SVG::ε | PNG::ε | GIF::ε | JPEG::ε
table:
        header::table-row* row::table-row+
table-row:
        table-cell::content-content
```

For formulæ we reuse MathML[ABC+01], which definition is omitted here. For productions we reuse BGF [LZ10, Zay10b, Zay10a]—the same notation we use in this paper.

We allow multiple **figure source**s so that the rendering tools for LDF can pick the source that is most convenient for the desired output format. For instance, a bitmap (**PNG**, **GIF**, **JPEG**) picture can be easily inserted into a web page, but a **PDF** file cannot be used in this manner. However, PDF may be preferred when LDF is rendered with pdfLaTeX.

4.5 Transformation of LDF Documents

In the introduction, we mentioned the pivotal role of transformations for enabling the life cycle of language documents. In this section, we want to briefly illustrate such document transformations on top of LDF.

Let us set up a challenge for document transformation. Consider the two standards of XPath: versions 1.0 [CD99] and 2.0 [BBC+07]. They are vastly different documents, the one being three times the size of the other; with different author teams, and generally following different structure. Thus, there is no correspondence (neither explicitly defined nor easily conceived) between the two versions, except for the backwards compatibility section in the latter, which statements cannot be validated explicitly. However, using language document engineering—including document transformations—we should be able to represent the delta between the two versions through a script of appropriate transformation steps.

We are working on a transformation language for LDF, i.e., XLDF, which should be ultimately sufficient in addressing conveniently the above challenge. We refer to [Zay10a] for a more extensive discussion of the XLDF effort, and we sketch XLDF in the sequel. Our current XLDF design and implementation has been useful already for simple problems. For [Zay09], we extracted a complete XBGF manual from the corresponding XML Schema, improved it with a few XLDF transformation steps and delivered a browsable version at the end. Such steps were needed because the assumed profile of XML Schema does not cover all LDF functionality.

XLDF is to LDF what XBGF [Zay09] is to BGF [LZ09]. That is, in the same sense as grammars can be adapted programmatically with XBGF, language documents would be adapted with XLDF. Apart from `xldf:transform-grammar` operator that lifts grammar transformations, XLDF also contains operators for introducing and moving content. Consider the following illustration where a number of operators are applied in a transformation sequence.

```
xldf:add-section(section:((title:"For Expressions",
                   id:"id-for-expressions"),
                   ...));
xldf:move-section(id:"section-Function-Calls",
                   inside:"id-primary-expressions");
xldf:rename-id(from:"section-Function-Calls",
                 to:"id-function-calls");
```

One could even think of meta-level transformations that affect the grammar notation used in LDF. For instance, XPath 1.0's grammar notation uses single quotes, while XPath 2.0's grammar notation uses double quotes:

```
xldf:change-grammar-notation(start-terminal,");
xldf:change-grammar-notation(end-terminal,");
```

Executing such XLDF commands would have to involve transforming the transformations that pretty-print BGF productions. Such higher-order transformations will be studied in future work on XLDF.

5 Additional Related Work

A discussion about general documentation approaches was already included in §2. Below we will discuss related work more broadly.

We have carried out a previously published case study for Cobol [Läm05] where the grammar of Cobol is extracted from the Cobol standard; it is then refactored, made consistent and finally put back into the standard without detailed parsing of the standard's structure. This is a limited case of document engineering where only grammar parts are affected, but it goes beyond grammar extraction due to the persistent link between the grammar and the manual.

In [Wai02, Wei02], respected experts in the field of technical documentation advocate the engineering approach to documentation, as opposed to the artistic one—without though covering the kind of domain support or life cycle that is enabled by LDF.

Original verification techniques on language documentation are presented in [SWJF09]. Checks include formulae like "for all reading paths, a term X must be defined before it is used". These ideas are complementary to ours.

The use of highly interactive eBooks for technical documentation is proposed in [DMW05]. In our domain, we use "browsable grammars" to enable interaction with language documentation.

Extraction for documentation is not necessarily restricted to text; extraction in [TL08] operates on graphic-rich documents. We could think of visual languages, UML-like models and "railroad tracks" kind of syntax diagrams.

One may also use Natural Language Generation (NLG) in deriving readable documents. For instance, in [RML98], the text is automatically generated with NLG when creating a final PDF output of the domain knowledge stored in a well-structured way. On a related account, several OMG technologies such as Knowledge Discovery Metamodel [KDM09] and Semantics of Business Vocabulary and Business Rules [SBV08] try to capture ontological concepts of the software domain and a means to make formal statements about them.

In [HR07], an industrial (Hewlett Packard) case study on documentation is presented. It involves user guides, man pages, context-sensitive help and white papers. The approach caters for a primary artifact from which a heterogeneous set of deliverables is generated with XMLware. To this end, disparate pieces of related information are positioned into final documents. A conceptually similar relationship between different documentation artifacts is considered in [BM06], where a view-based approach to software documentation is proposed.

6 Concluding Remarks

We have described the Language Document Format (LDF)—a unified format for language documents (say, language documentation). The unique characteristics of LDF are that i) it is derived by abstracting over a substantial and diverse body of actual language documents, and ii) it is integrated well with our previous research and infrastructure for grammar extraction, grammar recovery, grammar convergence, and grammar transformation.

Language document engineering with LDF brings us a step closer to the technical and methodological feasibility of life-cycle-enabled language documents so that state of the art documents could be migrated to a more structured setup of language documentation that is amenable to i) continuous validation, ii) systematic reuse of all embedded formal parts (grammars, examples, keywords, normative sections) in other grammar engineering activities, and iii) transformational support for evolution.

There are these major areas for future work on the subject. First, we will further improve our infrastructure for engineering language documents so that we serve a number of input and output formats with sufficient quality, for example, in terms of "recall" for extraction or "roundtripping" for re-exporting to legacy formats. Second, our current approach to supporting evolution of language documents is not fully developed. More language design work and possibly tool support is needed for the transformation language XLDF. Third, a case study bigger than XPath is required where an important language document (say, Cobol's or Java's standard) is converted into LDF, and the various benefits of our approach to language document engineering are properly illustrated, with language evolution as one of the most important issues.

References on language documentation

[ABC$^+$01] Ausbrooks, R., Buswell, S., Carlisle, D., Dalmas, S., Devitt, S., Diaz, A., Froumentin, M., Hunter, R., Ion, P., Kohlhase, M., Miner, R., Poppelier, N., Smith, B., Soiffer, N., Sutor, R., Watt, S.: Mathematical Markup Language (MathML) Version 2.0. W3C Recommendation, 2nd edn (2001)

[Bac60] Backus, J.W.: The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference. In: de Picciotto, S. (ed.) Proceedings of the International Conference on Information Processing, Unesco, Paris, pp. 125–131 (1960)

[BBC$^+$07] Berglund, A., Boag, S., Chamberlin, D., Fernández, M., Kay, M., Robie, J., Siméon, J.: XML Path Language (XPath) 2.0. W3C Recommendation (January 23, 2007), `www.w3.org/TR/2007/REC-xpath20-20070123`

[BBG$^+$63] Backus, J.W., Bauer, F.L., Green, J., Katz, C., McCarthy, J., Naur, P., Perlis, A.J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J.H., van Wijngaarden, A., Woodger, M.: Revised Report on the Algorithmic Language ALGOL 60. Numerische Mathematik 4, 420–453 (1963)

[Bru05] Brukardt, R.: ISO/IEC JTC1/SC22/WG9 Document N465—Report on “Grammar Engineering” (2005)

[CD99] Clark, J., DeRose, S.: XML Path Language (XPath) 1.0. W3C Recommendation (November 16, 1999), `http://www.w3.org/TR/1999/REC-xpath-19991116`

[ECM06] ECMA-334 Standard. C# Language Specification, 4th edn. (June 2006), `http://www.ecma-international.org/publications/standards/Ecma-334.htm`

[GHJV95] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)

[IBM03] IBM. Informix 4GL Reference Manual, 7.32 edn. (March 2003)

[ISO02] ISO/IEC 1989:2002. Information Technology—Programming Languages—COBOL (2002)

[ISO05] ISO/IEC 9899:TC2. Information Technology—Programming Languages—C, Committee Draft WG14/N1124 (2005)

[ISO07] ISO/IEC 14882. Information Technology—Programming Languages—C++, Committee Draft WG21/N2315, 2007. Accessed in (September 2010), `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2315.pdf`

[ISO08] ISO/IEC N2723=08-0233. Working Draft, Standard for Programming Language C++ (2008)

[KDM09] Object Management Group. Knowledge Discovery Metamodel (KDM), 1.1 edn. (January 2009), `http://www.omg.org/spec/KDM/1.1`

[Mic03] Microsoft .NET Framework Developer Center. ECMA and ISO/IEC C# and Common Language Infrastructure Standards and mirror sites (2003), `http://msdn.microsoft.com/netframework/ecma`

[MIL84] MIL–STD–1589C. Military Standard Jovial (J73) (July 1984)

[MOF06] Object Management Group. Meta-Object Facility (MOF$^{\text{TM}}$) Core Specification, 2.0 edn. (January 2006), `http://omg.org/spec/MOF/2.0`

[OAS07] OASIS. DITA Version 1.1 Language Specification Approved as an OASIS Standard. Committee Specification 01 (May 31, 2007), `http://docs.oasis-open.org/dita/v1.1/CS01/langspec`

[Rag97] Raggett, D.: HTML 3.2 Reference Specification. W3C Recommendation (January 14, 1997), `http://www.w3.org/TR/REC-html32`

[SBV08] Object Management Group. Semantics of Business Vocabulary and Rules (SBVR), 1.0 edn. (January 2008), `http://www.omg.org/spec/SBVR/1.0/`

[Sha97] Shan, Y.-P., et al.: NCITS J20 DRAFT of ANSI Smalltalk Standard, Revision 1.9 (December 1997), http://wiki.squeak.org/squeak/uploads/172/standard_v1_9-indexed.pdf (accessed in June 2007)

[W3C] Software Language Processing Suite: Mapping Spec Schema to LDF, http://slps.sf.net/w3c

[WM99] Walsh, N., Meullner, L.: DocBook: The Definitive Guide. O'Reilly, Sebastopol (1999)

[Zay09] Zaytsev, V.: XBGF Manual: BGF Transformation Operator Suite v.1.0 (August 2009), http://slps.sf.net/xbgf

Other References

[BM06] Bayer, J., Muthig, D.: A View-Based Approach for Improving Software Documentation Practices. In: Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS), Washington, DC, USA, 2006, pp. 269–278. IEEE Computer Society, Los Alamitos (2006)

[DMW05] Davison, G., Murphy, S., Wong, R.: The use of eBooks and interactive multimedia as alternative forms of technical documentation. In: Proceedings of the 23rd Annual International Conference on Design of Communication (SIGDOC), pp. 108–115. ACM, New York (2005)

[Fav05] Favre, J.-M.: Languages Evolve Too! Changing the Software Time Scale. In: IEEE (ed.) 8th Interntational Workshop on Principles of Software Evolution, IWPSE (2005)

[HR07] Haramundanis, K., Rowland, L.: Experience Paper: a Content Reuse Documentation Design Experience. In: Proceedings of the 25th Annual ACM International Conference on Design of Communication (SIGDOC), pp. 229–233. ACM, New York (2007)

[Knu84] Knuth, D.E.: Literate Programming. The Computer Journal 27(2), 97–111 (1984)

[KZ05] Klusener, S., Zaytsev, V.: ISO/IEC JTC1/SC22 Document N3977—Language Standardization Needs Grammarware (2005), http://www.open-std.org/jtc1/sc22/open/n3977.pdf

[Läm05] Lämmel, R.: The Amsterdam Toolkit for Language Archaeology. Electronic Notes in Theoretical Computer Science (ENTCS) 137(3), 43–55 (2005); Workshop on Metamodels, Schemas and Grammars for Reverse Engineering (ATEM 2004)

[LZ09] Lämmel, R., Zaytsev, V.: An introduction to grammar convergence. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 246–260. Springer, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-00255-7_17

[LZ10] Lämmel, R., Zaytsev, V.: Recovering Grammar Relationships for the Java Language Specification. Software Quality Journal, SCAM Special Issue (November 2010), http://dx.doi.org/10.1007/s11219-010-9116-5

[RML98] Reiter, E., Mellish, C., Levine, J.: Automatic Generation of Technical Documentation. Readings in Intelligent User Interfaces, 141–156 (1998)

[SWJF09] Schönberg, C., Weitl, F., Jakšić, M., Freitag, B.: Logic-based Verification of Technical Documentation. In: Proceedings of the 9th ACM Symposium on Document Engineering, pp. 251–252. ACM, New York (2009)

[TL08] Tombre, K., Lamiroy, B.: Pattern Recognition Methods for Querying and Browsing Technical Documentation. In: Ruiz-Shulcloper, J., Kropatsch, W.G. (eds.) CIARP 2008. LNCS, vol. 5197, pp. 504–518. Springer, Heidelberg (2008)

[Wai02] Waite, B.: Consequences of the Engineering Approach to Technical Writing. ACM Journal of Computer Documentation (JCD) 26(1), 22–26 (2002)

[Wei02] Weiss, E.H.: Egoless Writing: Improving Quality by Replacing Artistic Impulse with Engineering Discipline. ACM Journal of Computer Documentation (JCD) 26(1), 3–10 (2002)

[Zay10a] Zaytsev, V.: Recovery, Convergence and Documentation of Languages. PhD thesis, Vrije Universiteit, Amsterdam, The Netherlands (October 2010), `http://grammarware.net/text/2010/zaytsev-thesis.pdf`

[Zay10b] Zaytsev, V.: Language convergence infrastructure. In: Lämmel, R. (ed.) GTTSE 2009. LNCS, vol. 6491, pp. 481–497. Springer, Heidelberg (2011)

Canonical Method Names for Java

Using Implementation Semantics to Identify Synonymous Verbs

Einar W. Høst[1] and Bjarte M. Østvold[2]

[1] Computas AS
eih@computas.com
[2] Norwegian Computing Center
bjarte@nr.no

Abstract. Programmers rely on the conventional meanings of method names when writing programs. However, these conventional meanings are implicit and vague, leading to various forms of ambiguity. This is problematic since it hurts the readability and maintainability of programs. Java programmers would benefit greatly from a more well-defined vocabulary. Identifying synonyms in the vocabulary of verbs used in method names is a step towards this goal. By rooting the meaning of verbs in the semantics of a large number of methods taken from real-world Java applications, we find that such synonyms can readily be identified. To support our claims, we demonstrate automatic identification of synonym candidates. This could be used as a starting point for a manual canonicalisation process, where redundant verbs are eliminated from the vocabulary.

1 Introduction

Abelson and Sussman [1] contend that "programs must be written for people to read, and only incidentally for machines to execute". This is sound advice backed by the hard reality of economics: maintainability drives the cost of software systems [2], and readability drives the cost of maintenance [3,4]. Studies indicate some factors that influence readability, such as the presence or absence of abbreviations in identifiers [5]. Voices in the industry would have programmers using "good names" [6,7], typically meaning very explicit names. A different approach with the same goal is *spartan programming*[1]. "Geared at achieving the programming equivalent of laconic speech", spartan programming suggests conventions and practical techniques to reduce the complexity of program texts.

We contend that both approaches attempt to fight *ambiguity*. The natural language dimension of program texts, that is, the expressions encoded in the identifiers of the program, is inherently ambiguous. There are no enforced rules regarding the meaning of the identifiers, and hence we get ambiguity in the form of synonyms (several words are used for a single meaning) and polysemes

[1] `http://ssdl-wiki.cs.technion.ac.il/wiki/index.php/Spartan_programming`

(a single word has multiple meanings). This ambiguity could be reduced if we managed to establish a more well-defined vocabulary for programmers to use.

We restrict our attention to the first lexical token found in method names. For simplicity, we refer to all these tokens as "verbs", though they need not actually be verbs in English: `to` and `size` are two examples of this. We focus on verbs because they form a central, stable part of the vocabulary of programmers; whereas nouns tend to vary greatly by the domain of the program, the core set of verbs stays more or less intact.

We have shown before [8,9,10] that the meaning of verbs in method names can be modelled by abstracting over the bytecode of the method implementations. This allows us to 1) identify what is typical of implementations that share the same verb, and 2) compare the set of implementations for different verbs. In this paper, we aim at improving the core vocabulary of verbs for Java programmers by identifying potential synonyms that could be unified.

The contributions of this paper are:

- The introduction of *nominal entropy* as a way to measure how "nameable" a method is (Section 3.1).
- A technique to identify methods with "unnameable semantics" based on nominal entropy (Section 4.3).
- A technique to mechanically identify likely instances of code generation in a corpus of methods (Section 4.1).
- A formula to guide the identification of synonymous verbs in method names (Section 3.3).
- A mechanically generated graph showing synonym candidates for the most commonly used verbs in Java (Section 5.1).
- A mechanically generated list of suggestions for canonicalisation of verbs through unsupervised synonym elimination (Section 5.2).

2 Problem Description

To help the readability and learnability of the scripting language PowerShell, Microsoft has defined a standardised set of verbs to use. The verbs and their definitions can be found online[2], and PowerShell programmers are strongly encouraged to follow the conventions. The benefits to readability and learnability are obvious.

By contrast, the set of verbs used in method names in Java has emerged organically, as a mixture of verbs inherited from similar preceding languages, emulation of verbs used in the Java API, and so forth. A similar organic process occurs in natural languages. Steels argues that language "can be viewed as a complex adaptive system that adapts to exploit the available physiological and cognitive resources of its community of users in order to handle their communicative challenges" [11].

[2] `http://msdn.microsoft.com/en-us/library/ms714428%28VS.85%29.aspx`

We have seen before that Java programmers have a fairly homogenous, shared understanding of many of the most prevalent verbs used in Java programs [8]. Yet the organic evolution of conventional verb meaning has some obvious limitations:

– **Redundancy.** There are concepts that evolution has not selected a single verb to represent. This leads to superfluous synonymous verbs for the programmer to learn. Even worse, some programmers may use what are conventional synonyms in subtly different meanings.
– **Coarseness.** It is hard to organically grow verbs with precise meanings. To make sure that a verb is understood, it may be tempting to default to a very general and coarse verb. This results in "bagging" of different meanings into a small set of polysemous verbs.
– **Vagueness.** Evolution in Java has produced some common verbs that are almost devoid of meaning (such as `process` or `handle`), yet are lent a sense of legitimacy simply because they are common and shared among programmers.

Redundancy is the problem of *synonyms*, and can be addressed by identifying verbs with near-identical uses, and choosing a single, canonical verb among them. Coarseness is the problem of *polysemes*, and could be addressed by using data mining to identify common polysemous uses of a verb, and coming up with more precise names for these uses. Vagueness is hard to combat directly, as it is a result of the combination of a lack of a well-defined vocabulary with the programmer's lacking ability or will to create a clear, unambiguous and nameable abstraction. In this paper, we primarily address the problem of redundancy.

3 Analysis of Methods

The meaning of verbs in method names stems from the implementations they represent. That is, the meaning of a verb is simply the collection of observed uses of that verb (Section 3.1). Further, we hold that the verbs become more meaningful when they are consistently used to represent similar implementations. To make it easier to compare method implementations, we employ a coarse-grained semantic model for methods, based on predicates defined on Java bytecode (Section 3.2). We apply entropy considerations to measure both how consistently methods with the same verb are implemented, and how consistently the same implementation is named. We refer to this as semantic and nominal entropy, respectively. These two metrics are combined in a formula that we use to identify synonymous verbs (Section 3.3). Figure 1 presents an overview of the approach.

3.1 Definitions

We define a *method* m as a tuple consisting of three *components*: a unique *fingerprint* u, a *name* n, and a *semantics* s. Intuitively m is an idealised method, a model of a real method in Java bytecode. The unique fingerprints are a technicality that prevents set elements from collapsing into one; hence, a set made

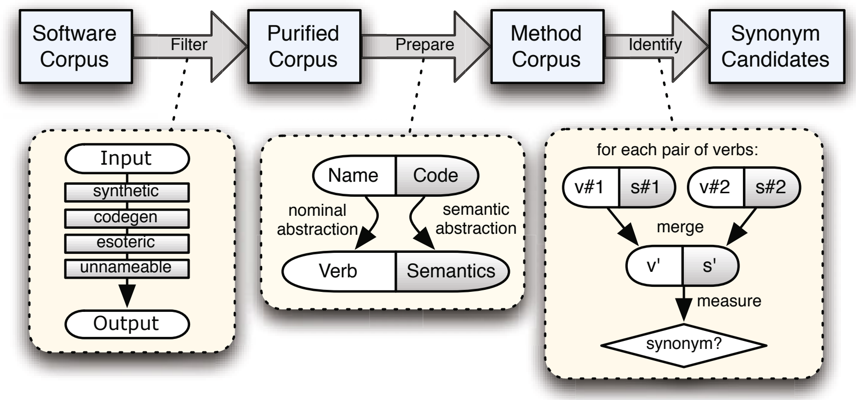


Fig. 1. Overview of the approach

from arbitrary methods $\{m_1, \ldots, m_k\}$ will always have k elements.[3] Often, we elide u from method tuples, writing just $m = (n, s)$.

We need two kinds of languages to reason about methods and their semantics. First, a concrete language where the semantics of a method is simply the string of Java bytecodes in m's implementation. Thus, bytecode is the canonical *concrete language*, denoted as $\mathcal{L}_{\text{Java}}$. For convenience, we define a labelling function f_{MD5} that maps from the bytecode to the MD5 digest of the opcodes in the bytecode. This allows us to easily apply uniform labels to the various implementations.

Second, we need an abstract language consisting of bit-vectors $[b_1, \ldots, b_k]$ where each b_i represents the result of evaluating a logical predicate q_i on a method's implementation. In the context of a concrete method m and its implementation, we refer to the vector $[b_1, \ldots, b_k]$ as *profile* of m. Different choices of predicates $q_1, \ldots, q_k$, leads to different *abstract languages*. Note that with the concrete language there is no limit on the size of a method's semantics; hence there is in principle an unlimited number of semantic objects. With an abstract language there is a fixed number of semantic objects, since s is a k-bit vector for some fixed number k, regardless of the choice of predicates.

A *corpus* $\mathcal{C}$ is a finite set of methods. We use the notation $\mathcal{C}/n$ to denote the set of methods in $\mathcal{C}$ that have name n, but where the semantics generally differs; and similarly $\mathcal{C}/s$ denotes the subcorpus of $\mathcal{C}$ where all methods have semantics s irrespective of their name. $\mathcal{C}/n$ is called a *nominal* corpus, $\mathcal{C}/s$ a *semantic* corpus.

Let x denote either a name component n or a semantics component s of some method. If $x_1, \ldots, x_k$ are all values occurring in $\mathcal{C}$ for a component, then we can view $\mathcal{C}$ as factored into disjoint subcorpora based on these values,

$$\mathcal{C} = \mathcal{C}/x_1 \cup \cdots \cup \mathcal{C}/x_k. \tag{1}$$

[3] The fingerprints models the mechanisms that the run-time system has for identifying distinct callable methods.

Corpus semantics and entropy. We repeat some information-theoretical concepts [12]. A *probability mass function* $p(x)$ is such that a) for all $i = 1, \ldots, k$ it holds that $0 \leq p(x_i) \leq 1$; and b) $\sum_{i=1}^{k} p(x_i) = 1$. Then $p(x_1), \ldots, p(x_k)$ is a *probability distribution.* From Equation (1) we observe that the following defines a probability mass function:

$$p(\mathcal{C}/x) \stackrel{\text{def}}{=} \frac{|\mathcal{C}/x|}{|\mathcal{C}|}$$

We write p^N for the nominal probability mass function based on name factoring $\mathcal{C}/n$ and Equation (1), and p^S for the semantic version.

We define the semantics $[\![\mathcal{C}]\!]$ of a corpus $\mathcal{C}$ in terms of the distribution defined by p^S:

$$[\![\mathcal{C}]\!] \stackrel{\text{def}}{=} p(\mathcal{C}/s_1) \ldots p^n(\mathcal{C}/s_k)$$

where we assume that $s_1, \ldots, s_k$ are all possible semantic objects in $\mathcal{C}$ as in Equation (1). Of particular interest is the semantics of a nominal corpus; we therefore write $[\![n]\!]$ as a shorthand for $[\![\mathcal{C}/n]\!]$ when $\mathcal{C}$ is obvious from the context. This is what we intuitively refer to as "the meaning of n".

Using the probability mass function, we introduce a notion of entropy for corpora—similar to Shannon entropy [12].

$$H(\mathcal{C}) \stackrel{\text{def}}{=} -\sum_{x \in \chi} p(\mathcal{C}/x) \log_2 p(\mathcal{C}/x)$$

where we assume $0 \log_2 0 = 0$. We write $H^N(\mathcal{C})$ for the *nominal entropy* of $\mathcal{C}$, in which case χ denotes the set of all names in $\mathcal{C}$; and $H^S(\mathcal{C})$ for *semantic entropy* of $\mathcal{C}$, where χ denotes the set of all semantics. The entropy $H^S(\mathcal{C})$ is a measure of the semantic diversity of $\mathcal{C}$: High entropy means high diversity, low entropy means few different method implementations. Entropy $H^N(\mathcal{C})$ has the dual interpretation.

Entropy is particularly interesting on subcorpora of $\mathcal{C}$. The nominal entropy of a semantic subcorpus, $H^N(\mathcal{C}/s)$, measures the consistency in the naming methods with profile s in $\mathcal{C}$. The semantic entropy of a nominal subcorpus, $H^S(\mathcal{C}/n)$ measures the consistency in the implementation of name n. The nominal entropy of a nominal subcorpus is not interesting as it is always 0. The same holds for the dual concept. When there can be no confusion about $\mathcal{C}$, we speak of the nominal entropy of a profile s,

$$H^N(s) \stackrel{\text{def}}{=} H^N(\mathcal{C}/s)$$

and similarly for the dual concept $H^S(n)$.

Nominal entropy can be used to compare profiles. A profile with comparatively low nominal entropy indicates an implementation that tends to be consistently named. A profile with comparatively high nominal entropy indicates an ambiguous implementation. An obvious example of the latter is the empty method.

We can also compare the semantic entropy of names. A name with comparatively low semantic entropy implies that methods with that name tend to be implemented using a few, well-understood "cliches". A name with comparatively high semantic entropy implies that programmers cannot agree on what to call such method implementations (or that the semantics are particularly ill-suited at capturing the nature of the name).

We define aggregated entropy of corpus $\mathcal{C}$ as follows.

$$H_{agg}(\mathcal{C}) \stackrel{\text{def}}{=} \frac{\sum_{x \in \mathcal{X}} |\mathcal{C}/x| H(\mathcal{C}/x)}{|\mathcal{C}|}$$

again leading to nominal and semantic notions of aggregated entropy, $H^N_{agg}(\mathcal{C})$ and $H^S_{agg}(\mathcal{C})$. These notions lets us quantify the overall entropy of subcorpora in $\mathcal{C}$, weighing the entropy of each subcorpus by its size.

Semantic cliches. When a method semantics is frequent in a corpus we call the semantics a semantic cliche, or simply a cliche, for that corpus. When a cliche has many different names we call it an unnameable cliche. Formally, a method semantics s is a *semantic cliche* for a corpus $\mathcal{C}$ if the prevalence of s in $\mathcal{C}$ is above some threshold value ϕ_{cl},

$$\frac{|\mathcal{C}/s|}{|\mathcal{C}|} > \phi_{cl}. \qquad (2)$$

Furthermore, s is an *unnameable semantic cliche* if it satisfies the above, and in addition the nominal entropy of corpus $\mathcal{C}/s$ is above some threshold value H^N_{cl}, $H^N(\mathcal{C}/s) > H^N_{cl}$.

3.2 Semantic Model

There are many ways of modelling the semantics of Java methods. For the purpose of comparing method names to implementations, we note one desirable property in particular. While the set of possible method implementations is practically unlimited, the set of different semantics in the model should both be finite and treat implementations that are essentially the same as having the same semantics. This is important, since each $\mathcal{C}/s$ should be large enough so that it is meaningful to speak of consistent or inconsistent naming of the methods in $\mathcal{C}/s$. This ensures that we can judge whether or not methods with semantics s are consistently named.

Some candidates for modelling method semantics are opcode sequences, abstract syntax trees and execution trace sets. However, we find these to be ill suited for our analysis: they do not provide a radical enough abstraction over the implementation. Therefore, we choose to model method semantics using an abstract language of bit vectors, as defined in Section 3.1.

Attributes. The abstract language relies on a set of predicates defined on Java bytecode. We refer to such predicates as *attributes* of the method implementation. Here we select and discuss the attributes we use, which yield a particular abstract language.

Individual attributes cannot distinguish perfectly between verbs. Rather, we expect to see trends when considering the probability that methods in each nominal corpus $\mathcal{C}/n$ satisfy each attribute. Furthermore, we note that 1) there might be names that are practically indistinguishable using bytecode predicates alone, and 2) some names are synonyms, and so *should* be indistinguishable.

Useful attributes. Intuitively, an attribute is *useful* if it helps distinguish between verbs. In Section 3.1, we noted that a verb might influence the probability that the predicate of an attribute is satisfied. Useful attributes have the property that this influence is significant. Attributes can be *broad* or *narrow* in scope. A broad attribute lets us identify larger groups of verbs that are aligned according to the attribute. A narrow attribute lets us identify smaller groups of verbs (sometimes consisting of a single verb). Both can be useful. The goal is to find a collection of attributes that together provides a good distinction between verbs.

Chosen attributes. We hand-craft a list of attributes for the abstract method semantics. An alternative would be to generate all possible simple predicates on bytecode instructions, and provide a selection mechanism to choose the "best" attributes according to some criterion. However, we find it useful to define predicates that involve a combination of bytecodes, for instance to describe control flow or subtleties in object creation. We deem it impractical to attempt a brute force search to find such combinations, and therefore resort to subjective judgement in defining attributes. To ensure a reasonable span of attributes, we pick attributes from the following categories: *method signature*, *object creation*, *data flow*, *control flow*, *exception handling* and *method calls*. The resulting attributes are listed in Table 1.

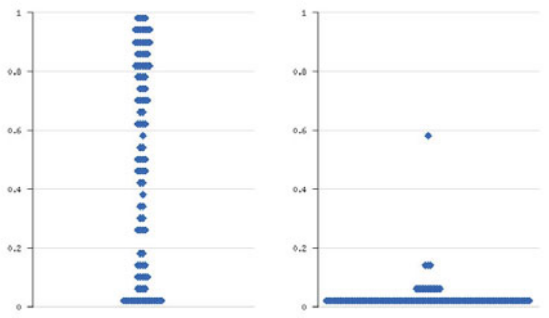
Probability distribution. The probability distribution for an attribute indicates if and how an attribute distinguishes between verbs. To illustrate, Figure 2 shows the probability distribution for two attributes: **Returns void** and **Writes parameter value to field**. Each dot represents the p^v for a given verb v, where v is a "common verb", as defined in Section 4.2. **Returns void** is a broad attribute, that distinguishes well between larger groups of verbs. However, there

Table 1. Attributes

Returns void	Returns field value
Returns boolean	Returns created object
Returns string	Runtime type check
No parameters	Creates custom objects[a]
Reads field	Contains loop
Writes field	Method call
Writes parameter value to field	Returns call result
Throws exceptions	Same verb call
Parameter value passed to method call on field value	

[a] A custom object is an instance of a type not in the `java.*` or `javax.*` namespaces.

are also verbs that are ambiguous with respect to the attribute. By contrast, **Writes parameter value to field** is a narrow attribute. Most verbs have a very low probability for this attribute, but there is a single verb which stands out with a fairly high probability: this verb is `set`, which is rather unsurprising.



(a) *Broad attribute:* **Returns void.**

(b) *Narrow attribute:* **Writes parameter value to field.**

Fig. 2. Probability distribution for some attributes

Critique. We have chosen a set of attributes for the semantic model based on our knowledge of commonly used method verbs in Java and how they are implemented. While all the attributes in the set are *useful* in the sense outlined above, we have no evidence that our set is "optimal" for the task at hand. There are two main problems with this.

First, we might have created an "unbalanced" set of attributes, meaning that we can have too many attributes capturing some kind of behaviour, such as object creation, and too few attributes capturing some other behaviour, such as exception handling. There might even be relevant behaviours that we have omitted altogether.

Second, we can construct many other attributes that could be used to distinguish between names; **Inverted method call**[4] and **Recursive call** are two candidates that we considered but rejected. The former is a narrow attribute that would help characterise `visit` methods, for instance. However, it turns out that `visit` is not ubiquitous enough to be included in our analysis (see Section 4.2); hence the attribute does not help in practise. The latter is simply too rarely satisfied to be very helpful.

The underlying problem is that there is no obvious metric by which to measure the quality of our attribute set. Arguably, the quality — or lack thereof — reveals itself in the results of our analysis.

[4] By "inverted method call", we mean that the calling object is passed as a parameter to the method call.

3.3 Identifying Synonyms

Intuitively, a verb n_1 is redundant if there exists another, more prevalent verb n_2 with *the same meaning*. It is somewhat fuzzy what "the same meaning" means. We define the meaning $[\![n]\!]$ of a verb n as the distribution of profiles in $\mathcal{C}/n$ (see Section 3.1). It is unlikely that the distributions for two verbs will be identical; however, some will be more similar than others. Hence we say that n_1 and n_2 have the same meaning if they are associated with sufficiently similar profile distributions.

We identify synonyms by investigating what happens when we merge the nominal corpora of two verbs. In other words, we attempt to eliminate one of the verbs, and investigate the effects on nominal and semantic entropy. If the effects are beneficial, we have identified a possible synonym.

The effects of synonym elimination. Elimination has two observable effects. First, there is a likely reduction in the aggregated nominal entropy H^N_{agg} of semantic corpora. The reason is that the nominal entropy of an individual semantic corpus is either *unaffected* by the elimination (if the eliminated verb is not used for any of the methods in the corpus), or it is *lowered*. Second, there is a likely increase in the aggregated semantic entropy H^S_{agg} of the nominal corpora — except for the unlikely event that the distribution of profiles is identical for the original corpora $\mathcal{C}/n_1$ and $\mathcal{C}/n_2$. How much H^S_{agg} increases depends on how semantically similar or different the eliminated verb is from the replacement verb. The increase in semantic entropy for the combined nominal corpus will be much lower for synonyms than for non-synonyms.

Optimisation strategy. When identifying synonyms, we must balance the positive effect on nominal entropy with the negative effect on semantic entropy. If we were to ignore the effect on semantic entropy, we would not be considering synonyms at all: simply to combine the two largest nominal corpora would yield the best effect. If we were to ignore the effect on nominal entropy, we would lose sight of the number of methods that are renamed. To combine a very large nominal corpus with a very small one would yield the best effect.

With this in mind, we devise a formula to guide us when identifying synonyms. A naive approach would be to demand that the positive effect on nominal entropy should simply be larger than the negative effect on semantic entropy. From practical experiments, we have found it necessary to emphasise semantic entropy over nominal entropy. That way, we avoid falsely identifying verbs with very large nominal corpora as synonyms. We therefore employ the following optimisation formula, which emphasises balance and avoids extremes, yet is particularly sensitive to increases in semantic entropy:

$$opt(\mathcal{C}) \stackrel{\text{def}}{=} \sqrt{4H^S_{agg}(\mathcal{C})^2 + H^N_{agg}(\mathcal{C})^2}$$

4 Software Corpus

We have gathered a corpus of Java programs of all sizes, from a wide variety of domains. We assume that the corpus is large and varied enough for the code

Table 2. The corpus of Java applications and libraries

Desktop applications			
ArgoUML 0.24	Azureus 2.5.0	BlueJ 2.1.3	Eclipse 3.2.1
JEdit 4.3	LimeWire 4.12.11	NetBeans 5.5	Poseidon CE 5.0.1
Programmer tools			
Ant 1.7.0	Cactus 1.7.2	Checkstyle 4.3	Cobertura 1.8
CruiseControl 2.6	Emma 2.0.5312	FitNesse	JUnit 4.2
Javassist 3.4	Maven 2.0.4	Velocity 1.4	
Languages and language tools			
ANTLR 2.7.6	ASM 2.2.3	AspectJ 1.5.3	BSF 2.4.0
BeanShell 2.0b	Groovy 1.0	JRuby 0.9.2	JavaCC 4.0
Jython 2.2b1	Kawa 1.9.1	MJC 1.3.2	Polyglot 2.1.0
Rhino 1.6r5			
Middleware, frameworks and toolkits			
AXIS 1.4	Avalon 4.1.5	Google Web Toolkit 1.3.3	JXTA 2.4.1
JacORB 2.3.0	Java 5 EE SDK	Java 6 SDK	Jini 2.1
Mule 1.3.3	OpenJMS 0.7.7a	PicoContainer 1.3	Spring 2.0.2
Sun WTK 2.5	Struts 2.0.1	Tapestry 4.0.2	WSDL4J 1.6.2
Servers and databases			
DB Derby 10.2.2.0	Geronimo 1.1.1	HSQLDB	JBoss 4.0.5
JOnAS 4.8.4	James 2.3.0	Jetty 6.1.1	Tomcat 6.0.7b
XML tools			
Castor 1.1	Dom4J 1.6.1	JDOM 1.0	Piccolo 1.04
Saxon 8.8	XBean 2.0.0	XOM 1.1	XPP 1.1.3.4
XStream 1.2.1	Xalan-J 2.7.0	Xerces-J 2.9.0	
Utilities and libraries			
Batik 1.6	BluePrints UI 1.4	c3p0 0.9.1	CGLib 2.1.03
Ganymed ssh b209	Genericra	HOWL 1.0.2	Hibernate 3.2.1
JGroups 2.2.8	JarJar Links 0.7	Log4J 1.2.14	MOF
MX4J 3.0.2	OGNL 2.6.9	OpenSAML 1.0.1	Shale Remoting
TranQL 1.3	Trove	XML Security 1.3.0	
Jakarta commons utilities			
Codec 1.3	Collections 3.2	DBCP 1.2.1	Digester 1.8
Discovery 0.4	EL 1.0	FileUpload 1.2	HttpClient 3.0.1
IO 1.3.1	Lang 2.3	Modeler 2.0	Net 1.4.1
Pool 1.3	Validator 1.3.1		

to be representative of Java programming in general. Table 2 lists the 100 Java applications, frameworks and libraries that constitute our corpus.

We filter the corpus in various ways to "purify" it:

- Omit compiler-generated methods (marked as **synthetic** in the bytecode).
- Omit methods that appear to have been code-generated.
- Omit methods without a common verb-name.
- Omit methods with unnameable semantics.

Table 3. The effects of corpus filtering

Total methods	1.226.611
Non-synthetic	1.090.982
Hand-written	1.050.707
Common-verb name	818.503
Nameable semantics	778.715

The purpose of the filtering is to reduce the amount of noise affecting our analysis. Table 3 presents some numbers indicating the size of the corpus and the impact of each filtering step.

4.1 Source Code Generation

Generation of source code represents a challenge for our analysis, since it can lead to a skewed impression of the semantics of a verb. In our context, the problem is this: a single application may contain a large number of near-identical methods, with identical verb and identical profile. The result is that the nominal corpus corresponding to the verb in question is "flooded" by methods with a specific profile, skewing the semantics of that corpus. Conversely, the semantic corpus corresponding to the profile in question is also "flooded" by methods with a specific verb, giving us a wrong impression of how methods with that profile are named.

To diminish the influence of code generation, we impose limits on the number of method instances contributed by a single application. By comparing the contribution from individual applications to that of all others, we can calculate an *expected* contribution for the application. We compare this with the *actual* contribution, and truncate the contribution if the ratio between the two numbers is unreasonable.

If the actual contribution is above some threshold T, then we truncate it to:

$$\max(T, \min(\frac{|\mathcal{C}_a/v|}{|\mathcal{C}/v|}, L\frac{|\mathcal{C}/v| - |\mathcal{C}_a/v|}{|\mathcal{C}| - |\mathcal{C}_a|}))$$

where L acts as a constraint on how much the contribution may exceed expectations.

Determining T and L is a subjective judgement, since we have no way of identifying false positives or false negatives among the method instances we eliminate. Our goal is to diminish the influence of code generation on our analysis rather than eliminate it. Therefore, we opt to be fairly lax, erring more on the side of false negatives than false positives. In our analysis, $T = 50$ and $L = 25$; that is, if some application contains more than 50 identical methods (n, s), we check that the number of identical methods does not exceed 25 times that of the average application. This nevertheless captures quite a few instances of evident code generation.

Table 4. Vocabularies

Vocabulary	*% Apps*	*Verbs*	*Methods*	*Example verbs*
essential	⟨ 90, 100]	7	50.73%	get, set, create
core	⟨ 75, 90]	21	13.26%	find, equals, parse
extended	⟨ 50, 75]	74	13.92%	handle, match, save
specific	⟨ 25, 50]	220	11.72%	sort, visit, refresh
narrow	⟨ 10, 25]	555	5.95%	render, shift, purge
marginal	⟨ 0, 10]	5722	4.43%	squeeze, unhook, animate

4.2 Common Verbs

Some verbs are common, such as `get` and `set`, whereas others are esoteric, such as `unproxy` and `scavenge`. In this paper, we focus on the former and ignore the latter. There are several possible interpretations of *common*; two obvious candidates are *ubiquity* (percentage of applications) and *volume* (number of methods).

We choose ubiquity as our interpretation of *common*. Rudimentary grouping of verbs according to ubiquity is shown in Table 4. Since we are interested in the shared vocabulary of programmers, we restrict our analysis to the top three groups: *essential*, *core* and *extended*. The 102 verbs in these three groups cover nearly 77% of all methods (after filtering of generated code). Figure 3 shows a "word cloud" visualisation[5] of the common verbs.

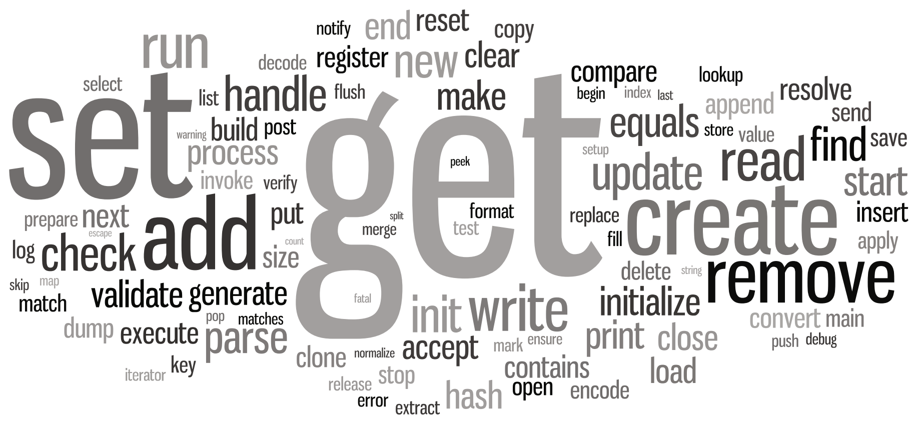


Fig. 3. The 102 most common verbs

4.3 Unnameable Cliches

Unnameable cliches, that is, method implementations that are common, yet inherently ambiguous, constitute noise for our analysis. We aim to reduce the

[5] Generated by Wordle.net.

impact of this noise by omitting methods whose implementations are unnameable cliches. The rationale is that the semantics of each verb will be more distinct without the noise, making it easier to compare and contrast the verbs.

In some cases, an implementation cliche may appear to be unnameable without being inherently ambiguous: rather, no generally accepted name for it has emerged. By applying canonicalisation through synonym elimination, the naming ambiguity can be reduced to normal levels. We must therefore distinguish between cliches that are *seemingly* and *genuinely* unnameable.

To identify implementation cliches, we use the concrete language $\mathcal{L}_{\text{Java}}$ (see Section 3.1). Table 5 shows unnameable cliches identified using Equation (2), with $\phi_{cl} = 500$ and $H^N_{cl} = 1.75$. We also include a reverse engineered example in a stylized Java source code-like syntax for each cliche.

To label the method implementations we apply f_{MD5}, which yields the MD5 digest of the opcodes for each implementation. Note that we include only the opcodes in the digest. We omit the operands to avoid distinguishing between implementations based on constants, text strings, the names of types and methods, and so forth. Hence f_{MD5} does abstract over the implementation somewhat. As a consequence, we cannot distinguish between, say, `this.`$m(p)$ and p`.`m`(this)`: these are considered instances of the same cliche. Also, some cliches may yield the same example, since there are opcode sequences that cannot be distinguished when written as stylized source code.

Most of the cliches in Table 5 seem genuinely unnameable. Unsurprisingly, variations over delegation to other methods dominate. We cannot reasonably

Table 5. Semantic cliches with unstable naming

Cliche	*# Methods*	H^N	*Retain*	*Top names*
{ super.m(); }	539	3.50		remove [10.6%], set [8.7%], insert [6.5%]
{ }	14566	3.40		set [18.1%], initialize [8.4%], end [7.8%]
{ this.m(); }	794	3.22		set [18.5%], close [7.2%], do [6.2%]
{ this.f.m(); }	2007	2.94		clear [20.7%], close [13.2%], run [11.7%]
{ return p; }	532	2.94		get [34.4%], convert [7.1%], create [4.7%]
{ super.$m(p)$; }	742	2.69		set [32.2%], end [12.0%], add [9.4%]
{ throw new E(); }	3511	2.68		get [25.4%], remove [17.2%], set [14.8%]
{ this.f.m(); }	900	2.65		clear [28.2%], remove [16.1%], close [9.9%]
{ throw new $E(s)$; }	5666	2.59		get [25.9%], set [22.3%], create [10.7%]
{ this.f.$m(p)$; }	1062	2.48		set [39.2%], add [14.8%], remove [12.0%]
{ this.$m(p)$; }	1476	2.45		set [24.4%], end [21.7%], add [14.2%]
{ return this.f.$m(p)$; }	954	2.38		contains [25.9%], is [20.8%], equals [11.1%]
{ this.f.m(p_1, p_2); }	522	2.34		set [33.0%], add [17.2%], remove [13.0%]
{ return this.f.m(p); }	929	2.14		contains [28.3%], is [25.0%], get [11.1%]
{ return this.m(p); }	618	2.14		get [52.8%], post [8.4%], create [6.3%]
{ this.f = true; }	631	2.08	✓	set [48.5%], mark [12.8%], start [6.7%]
{ C.m(this.f); }	544	1.96		run [46.9%], handle [14.3%], insert [9.9%]
{ this.f.$m(p)$; }	3906	1.92		set [36.8%], add [29.7%], remove [16.8%]
{ return new C(this); }	1540	1.87	✓	create [34.6%], get [25.7%], new [11.9%]
{ return this.m(); }	520	1.83		get [45.0%], is [20.0%], has [12.5%]
{ return false; }	6322	1.83	✓	is [52.8%], get [20.1%], has [7.3%]

provide a name for such methods without considering the names of the methods being delegated to. There are also some examples of "unimplemented" methods; for instance `{ throw new E(); }` or the empty method `{ }`. We believe that in many cases, the presence of these methods will be required by the compiler (for instance to satisfy some interface), but in practice, they will never be invoked.

Table 5 also contains three cliches that we deem only seemingly unnameable. This is based on a subjective judgement that they could be relatively consistently named, given a more well-defined vocabulary. These have been marked as being *retained*, meaning they are included in the analysis. The others are omitted.

5 Addressing Synonyms

To address the problem of synonyms, we employ the formula *opt* from Section 3.3. We use *opt* to mechanically identify likely synonyms in the corpus described in Section 4, and then to attempt unsupervised elimination of synonyms.

5.1 Identifying Synonyms

Compared to each of the common verbs in the corpus, the other verbs will range from synonyms or "semantic siblings" to the opposite or unrelated. To find the verbs that are semantically most similar to each verb, we calculate the value for *opt* when merging the nominal corpus of each verb with the nominal corpus of each of the other verbs. The verbs that yield the lowest value for *opt* are considered synonym candidates.

It is more likely that two verbs are genuine synonyms if they reciprocally hold each other to be synonym candidates. When we identify such pairs of synonym candidates, we find that clusters emerge among the verbs, as shown in Figure 4.

Several of the clusters could be labelled, for instance as *questions*, *initialisers*, *factories*, *runners*, *checkers* and *terminators*. This suggests that these clusters have a "topic". It does not imply that all the verbs in each cluster could be replaced by a single verb, however. For instance, note that in the *factory* cluster, `create` and `make` are indicated as synonym candidates, as are `create` and `new`, but `new` and `make` are not. An explanation could be that `create` has a broader use than `new` and `make`.

We also see that there are two large clusters that appear to have more than one topic. We offer two possible explanations. First, polysemous verbs will tie together otherwise unrelated topics (see Section 2). In the largest cluster, for instance, we find a mix of verbs associated with I/O and verbs that handle collections. In this case, `append` is an example of a polysemous verb used in both contexts. Second, we may lack attributes to distinguish appropriately between the verbs.

5.2 Eliminating Synonyms

To eliminate synonyms, we iterate over the collection of verbs. We greedily select the elimination that yields the best immediate benefit for *opt* in each iteration.

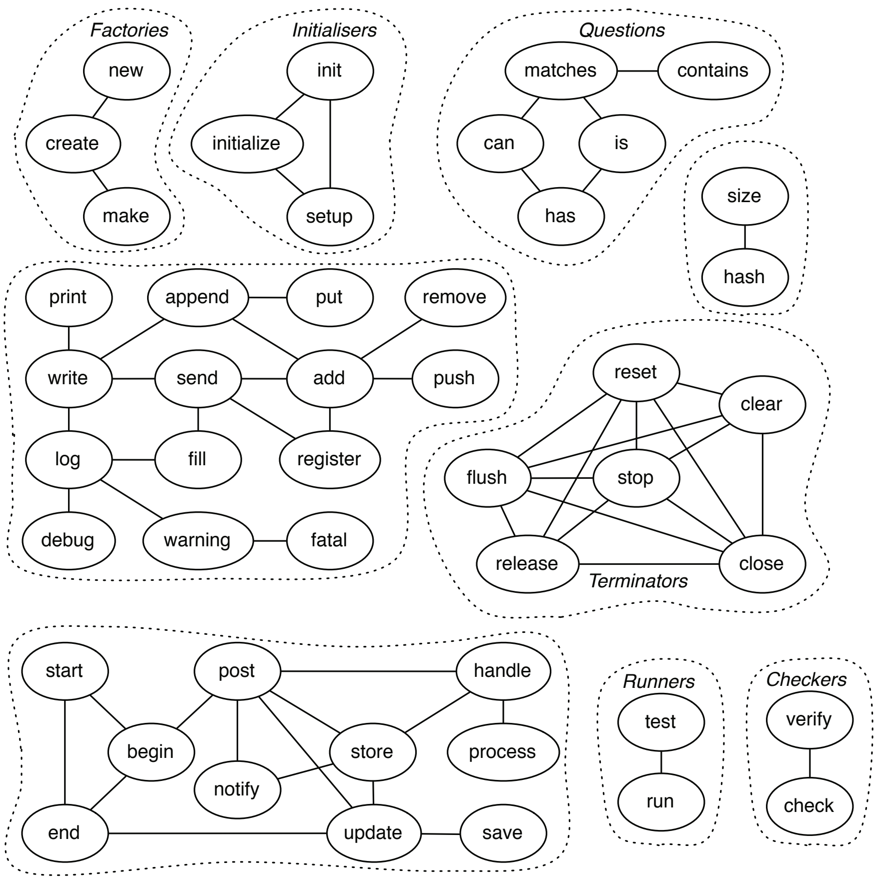


Fig. 4. Clusters of synonym candidates. Clusters with a single topic are labelled.

Table 6. Mechanical elimination of synonyms

Run	*Canonical verb (cv)*	*Old verbs*	$\lvert\mathcal{C}/cv\rvert$	*Sum*	ΔH^S_{agg}	ΔH^N_{agg}	Δopt
1	is	has+is	49041	6820+42221	0.00269	-0.02270	-0.01152
2	is	can+is	51649	2608+49041	0.00178	-0.01148	-0.00409
3	add	remove+add	43241	16172+27069	0.00667	-0.03004	-0.00237
4	init	initialize+init	11026	3568+7458	0.00149	-0.00743	-0.00126
5	close	stop+close	5025	1810+3215	0.00074	-0.00348	-0.00040
6	create	make+create	38140	4940+33200	0.00363	-0.01525	-0.00021
7	close	flush+close	5936	911+5025	0.00061	-0.00266	-0.00014
8	reset	clear+reset	5849	2901+2948	0.00100	-0.00421	-0.00007
9	write	log+write	13659	1775+11884	0.00131	-0.00547	-0.00004

Table 7. Manual elimination of synonyms

Canonical verb (cv)	*Old verbs*	$\|\mathcal{C}/cv\|$	*Sum*	ΔH^S_{agg}	ΔH^N_{agg}	Δopt
clone	clone+copy	4732	2595+2137	0.00271	-0.00147	0.00979
execute	execute+invoke	4947	2997+1950	0.00197	-0.00229	0.00589
verify	check+verify	8550	7440+1110	0.00126	-0.00298	0.00223
stop	stop+end	4814	1810+3004	0.00126	-0.00283	0.00242
write	write+log+dump	15987	11884+1775+2328	0.00420	-0.01109	0.00635
start	start+begin	5485	4735+750	0.00081	-0.00200	0.00135
init	init+initialize	11026	7458+3568	0.00149	-0.00743	-0.00126
error	error+fatal	1531	1116+415	0.00027	-0.00088	0.00023
create	create+new+make	45565	33200+7425+4940	0.00901	-0.03588	0.00152

We assume that beneficial eliminations will occur eventually, and that the order of eliminations is not important. We only label a synonym candidate as "genuine" if the value for *opt* decreases; the iteration stops when no more genuine candidates can be found. For comparison, we also perform manual elimination of synonyms, based on a hand-crafted list of synonym candidates.

The results of mechanical synonym elimination are shown in Table 6. Note that the input to the elimination algorithm is the output given by the preceding run of the algorithm. For the first run, the input is the original "purified" corpus described in Section 4, whereas for the second, the verb `has` has been eliminated, and the original nominal corpora for `has` and `is` have been merged.

The elimination of `has` is interesting: it is considered the most beneficial elimination by *opt*, yet as Java programmers, we would hesitate to eliminate it. The subtle differences in meaning between all "boolean queries" (`is`, `has`, `can`, `supports` and so forth) are hard to discern at the implementation level. Indeed, we would often accept method names with different verbs for the same implementation: `hasChildren` and `isParent` could be equally valid names. This kind of nominal flexibility is arguably beneficial for the readability of code.

It is easier to see that either `init` or `initialize` should be eliminated: there is no reason for the duplication. Eliminating `make` and using `create` as a canonical verb for factory methods also seems reasonable. Similarly, the suggestion to use `write` instead of `log` is understandable — however, one could argue that `log` is useful because it is more precise than the generic `write`.

There seems to be quite a few verbs for "termination code"; some of these verbs might be redundant. The unsupervised elimination process identifies `flush`, `stop` and `close` as candidates for synonym elimination. However, we find it unacceptable: certainly, `flush` and `close` cannot always be used interchangeably. In our coarse-grained semantic model, we lack the "semantic clues" to distinguish between these related, yet distinct verbs.

The suggestion to combine `add` and `remove` is also problematic, again showing that the approach has some limitations. Both `add` and `remove` typically involve collections of items, perhaps including iteration (which is captured by the **Contains loop** attribute). The crucial distinction between the two operations will often be hidden inside a call to a method in the Java API. Even if we were to observe the actual adding or removing of an item, this might involve incrementing or decrementing a counter, which is not captured by our model.

Table 7 shows the result of the manual elimination of synonyms. We note that only the elimination of `initialize` yields a decreased value for *opt* — apparently, we are not very good at manual synonym identification! However, it may be that the requirement that *opt* should decrease is too strict. Indeed, we find that many of our candidates are present in the clusters shown in Figure 4. This indicates that there is no deep conflict between our suggestions and the underlying data.

5.3 Canonicalisation

Overall, we note that our approach succeeds in finding relevant candidates for synonym elimination. However, it is also clear that the elimination must be supervised by a programmer. We therefore suggest using Figure 4 as a starting point for manual canonicalisation of verbs in method names. Canonicalisation should entail both eliminating synonyms and providing a precise definition, rationale and use cases for each verb.

6 Related Work

Gil and Maman [13] introduce the notion of machine-traceable patterns, in order to identify so-called micro patterns; machine-traceable implementation patterns at the class level. When we model the semantics of method implementations using hand-crafted bytecode predicates, we could in principle discern "nano patterns" at the method implementation level. According to Gamma et al. [14], however, a pattern has four essential elements: name, problem, solution and consequences. Though we do identify some very commonly used implementation cliches, we do not attempt to interpret and structure these cliches. Still, Singer et al. [15] present their own expanded set of bytecode predicates under the label "fundamental nano patterns", where the term "pattern" must be understood in a broader, more colloquial sense.

Collberg et al. [16] present a large set of low-level statistics from a corpus of Java applications, similar in size to ours. Most interesting to us are the statistics showing k-grams of opcodes, highlighting the most commonly found opcode sequences. This is similar to the implementation cliches we find in our work. Unfortunately, the k-grams are not considered as logical entities, so a common 2-gram will often appear as part of a common 3-gram as well.

Similar in spirit to our work, Singer and Kirkham [17] find a correlation between certain commonly used type name suffixes and some of Gil and Maman's micro patterns. Pollock et al. [18] propose using "natural language program analysis", where natural language clues found in comments and identifiers are used to augment and guide program analyses. Tools for program navigation and aspect mining have been developed [19,20] based on this idea. Ma et al. [21] exploit the fact that programmers usually choose appropriate names in their code to guide searches for software artefacts.

The quality of identifiers is widely recognised as important. Deißenböck and Pizka [22] seek to formalise two quality metrics, *conciseness* and *consistency*, based on a bijective mapping between identifers and concepts. Unfortunately, the mapping must be constructed by a human expert. Lawrie et al. [23] seek to overcome this problem by deriving syntactic rules for conciseness and consistency from the identifiers themselves. This makes the approach much more applicable, but introduces the potential for false positives and negatives.

7 Conclusion and Further Work

The ambiguous vocabulary of verbs used in method names makes Java programs less readable than they could be. We have identified *redundancy*, *coarseness* and *vagueness* as the problems to address. In this paper, we focussed on *redundancy*, where more than one verb is used in the same meaning. We looked at the identification and elimination of synonymous verbs as a means towards this goal.

We found that we were indeed able to identify reasonable synonym candidates for many verbs. To select the genuine synonyms among the candidates without human supervision is more problematic. The abstract semantics we use for method implementations is sometimes insufficient to capture important nuances between verbs. A more sophisticated model that takes into account invoked methods, either semantically (by interprocedural analysis of bytecode) or nominally (by noting the names of the invoked methods) might overcome some of these problems. Realistically, however, the perspective of a programmer will probably still be needed. A more fruitful way forward may be to use the identified synonym candidates as a starting point for a manual process where a canonical set of verbs is given precise definitions, and the rest are discouraged from use.

Addressing the problem of *coarseness* is a natural counterpart to the topic of this paper. Coarseness manifests itself in polysemous verbs, that is, verbs that have more than a single meaning. Polysemous verbs could be addressed by investigating the semantics of the methods that constitute a nominal corpus $\mathcal{C}/n$. The intuition is that polysemous uses of n will reveal itself as clusters of semantically similar methods. Standard data clustering techniques could be applied to identify such polysemous clusters. If a nominal corpus were found to contain polysemous clusters, we could investigate the effect of renaming the methods in one of the clusters. This would entail splitting the original nominal corpus $\mathcal{C}/n$ in two, $\mathcal{C}/n$ and $\mathcal{C}/n'$. The effect of splitting the corpus could be measured, for instance by applying the formula *opt* from Section 3.3.

References

1. Abelson, H., Sussman, G.J.: Structure and Interpretation of Computer Programs, 2nd edn. MIT Electrical Engineering and Computer Science. MIT Press, Cambridge (1996)
2. Eierman, M.A., Dishaw, M.T.: The process of software maintenance: a comparison of object-oriented and third-generation development languages. Journal of Software Maintenance and Evolution: Research and Practice 19(1), 33–47 (2007)

3. Collar, E., Valerdi, R.: Role of software readability on software development cost. In: Proceedings of the 21st Forum on COCOMO and Software Cost Modeling, Herndon, VA (October 2006)
4. von Mayrhauser, A., Vans, A.M.: Program comprehension during software maintenance and evolution. Computer 28(8), 44–55 (1995)
5. Lawrie, D., Morrell, C., Feild, H., Binkley, D.: Effective identifier names for comprehension and memory. ISSE 3(4), 303–318 (2007)
6. Martin, R.C.: Clean Code. Prentice-Hall, Englewood Cliffs (2008)
7. Beck, K.: Implementation Patterns. Addison-Wesley Professional, Reading (2007)
8. Høst, E.W., Østvold, B.M.: The programmer's lexicon, volume I: The verbs. In: Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007), pp. 193–202. IEEE Computer Society, Los Alamitos (2007)
9. Høst, E.W., Østvold, B.M.: The java programmer's phrase book. In: Gašević, D., Lämmel, R., Van Wyk, E. (eds.) SLE 2008. LNCS, vol. 5452, pp. 322–341. Springer, Heidelberg (2009)
10. Høst, E.W., Østvold, B.M.: Debugging method names. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 294–317. Springer, Heidelberg (2009)
11. Steels, L.: The recruitment theory of language origins. In: Lyon, C., Nehaniv, C.L., Cangelosi, A. (eds.) Emergence of Language and Communication, pp. 129–151. Springer, Heidelberg (2007)
12. Cover, T.M., Thomas, J.A.: Elements of Information Theory, 2nd edn. Wiley Series in Telecommunications. Wiley, Chichester (2006)
13. Gil, J., Maman, I.: Micro patterns in Java code. In: Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005), pp. 97–116. ACM, New York (2005)
14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley, Boston (1995)
15. Singer, J., Brown, G., Lujan, M., Pocock, A., Yiapanis, P.: Fundamental nano-patterns to characterize and classify Java methods. In: Proceedings of the 9th Workshop on Language Descriptions, Tools and Applications (LDTA 2009), pp. 204–218 (2009)
16. Collberg, C., Myles, G., Stepp, M.: An empirical study of Java bytecode programs. Software Practice and Experience 37(6), 581–641 (2007)
17. Singer, J., Kirkham, C.: Exploiting the correspondence between micro patterns and class names. In: Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2008), pp. 67–76. IEEE Computer Society, Los Alamitos (2008)
18. Pollock, L.L., Vijay-Shanker, K., Shepherd, D., Hill, E., Fry, Z.P., Maloor, K.: Introducing natural language program analysis. In: Das, M., Grossman, D. (eds.) Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2007), pp. 15–16. ACM, New York (2007)
19. Shepherd, D., Pollock, L.L., Vijay-Shanker, K.: Towards supporting on-demand virtual remodularization using program graphs. In: Filman, R.E. (ed.) Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD 2006), pp. 3–14. ACM, New York (2006)
20. Shepherd, D., Fry, Z.P., Hill, E., Pollock, L., Vijay-Shanker, K.: Using natural language program analysis to locate and understand action-oriented concerns. In: Proceedings of the 6th International Conference on Aspect-Oriented Software Development (AOSD 2007), pp. 212–224. ACM, New York (2007)

21. Ma, H., Amor, R., Tempero, E.D.: Indexing the Java API using source code. In: Proceedings of the 19th Australian Software Engineering Conference (ASWEC 2008), pp. 451–460. IEEE Computer Society, Los Alamitos (2008)
22. Deißenböck, F., Pizka, M.: Concise and consistent naming. In: Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC 2005), pp. 97–106. IEEE Computer Society, Los Alamitos (2005)
23. Lawrie, D., Feild, H., Binkley, D.: Syntactic identifier conciseness and consistency. In: 6th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006), pp. 139–148. IEEE Computer Society, Los Alamitos (2006)

Subjective-C
Bringing Context to Mobile Platform Programming

Sebastián González, Nicolás Cardozo, Kim Mens,
Alfredo Cádiz, Jean-Christophe Libbrecht, and Julien Goffaux

Computing Science Engineering Pole, ICTEAM, UCLouvain
Place Sainte-Barbe 2, 1348 Louvain-la-Neuve, Belgium

Abstract. Thanks to steady advances in hardware, mobile computing platforms are nowadays much more connected to their physical and logical environment than ever before. To ease the construction of adaptable applications that are smarter with respect to their execution environment, the context-oriented programming paradigm has emerged. However, up until now there has been no proof that this emerging paradigm can be implemented and used effectively on mobile devices, probably the kind of platform which is most subject to dynamically changing contexts. In this paper we study how to effectively realise core context-oriented abstractions on top of Objective-C, a mainstream language for mobile device programming. The result is Subjective-C, a language which goes beyond existing context-oriented languages by providing a rich encoding of context interdependencies. Our initial validation cases and efficiency benchmarks make us confident that context-oriented programming can become mainstream in mobile application development.

1 Introduction

New computing platforms are interrelated to their physical execution environment through all kinds of sensors that are able to measure location, orientation, movement, light, sound, temperature, network signal strength, battery charge level, and so forth. At the logical level, even traditional desktop and server platforms are getting exposed to richer environments in which they can find network services of all sorts. Both at the physical and logical levels, the live environment in which applications execute is acquiring a central role. If equipped with higher levels of context-driven adaptability, software systems can become smarter with respect to their environment and to user needs, exhibit emergent properties, be resilient in the face of perturbations, and generally fit better in the technical ecosystem in which they are used.

Unfortunately, most software systems do not meet the high adaptability expectations that stem naturally from their connectedness to their environment. Most applications exhibit fixed functionality and are seldom aware of changing contexts to adapt their behaviour accordingly. Many chances of delivering improved services to users and network peers are thus missed. We hypothesise that a major reason for this lack of adaptability is the unavailability of appropriate context-aware programming languages and related tool sets. Current programming technology does not put programmers in the right state of mind, nor does it provide adequate abstractions, to program context-aware applications.

Starting from this observation, Context-Oriented Programming (COP) has been introduced as a novel programming paradigm which eases the development of adaptable behaviour according to changing contexts [4,9]. COP offers an alternative to hard-coded conditional statements and special design patterns to encode context-dependent behaviour. COP thereby renders code more reusable and maintainable. Unfortunately, current COP languages do not run on mobile platforms —probably the kind of platform for which context is most relevant, thus offering the most promising possibilities for development of context-aware applications. Furthermore, COP languages still lack dedicated facilities to model the knowledge of the situation in which applications execute.

With the aim of having a COP language that runs on a mobile platform, we set out to develop an extension of Objective-C, one of the most widespread programming languages for mobile systems. The result is Subjective-C, a new COP language extension aimed at easing the construction of context-aware mobile applications. In Subjective-C, object behaviour depends on the context in which it executes. Hence, observed behaviour is not absolute or objective, but rather of a more relative or *subjective* nature [17]. A minimum amount of computational reflection available in Objective-C suffices to add the necessary abstractions which allow the straightforward expression of context-specific behaviour.

Subjective-C is not a mere reimplementation of the concepts behind main COP languages like Ambience [8] and ContextL [4]. Subjective-C goes beyond the simple inheritance relationships that are possible between context objects in Ambience, and between layer classes in ContextL, by providing explicit means to encode more advanced interdependencies between contexts. Not only does Subjective-C allow for more kinds of dependencies, but also they can be expressed in a domain-specific language developed especially for this purpose, making the declaration of such dependencies more readable.

To validate Subjective-C, we implemented three proof-of-concept applications that run on actual smartphones. These case studies showed the feasibility of programming context-aware applications using subjective programming abstractions, with a noticeable increase in software understandability, maintainability and extensibility. Furthermore, efficiency benchmarks show that the performance impact of COP abstractions in Subjective-C is negligible, to the point that in some cases it can even improve performance.

The remainder of this paper is organised as follows. Section 2 introduces the basics of context-oriented programming in Subjective-C. Section 3 goes on to explain context relations in detail. Section 4 presents the reflective implementation technique we used to add a subjective layer on top of Objective-C. Section 5 briefly presents three validation cases we conducted to assess the advantages and disadvantages of Subjective-C. Section 6 reports on the efficiency benchmarks we carried out. Section 7 discusses limitations and future work. We present related work in Section 8, and draw the paper to a close in Section 9.

2 Context-Oriented Programming in Subjective-C

Context-Oriented Programming (COP) is an emerging programming paradigm in which contextual information plays a central role in the definition of application behaviour [4,8,18]. The essence of COP is the ability to overlay adapted behaviour on top of existing default behaviour, according to the circumstances in which the software is being used. Such adaptations are meant to gracefully adjust the service level of the application, following detected changes in the execution environment. COP languages provide dedicated programming abstractions to enable this behavioural adaptability to changing contexts. This section presents the COP core on which Subjective-C is based. The fundamental language constructs are introduced progressively, as core mechanisms are explained.

2.1 General System Architecture

Subjective-C has been conceived for a fairly straightforward system architecture, illustrated in Fig. 1. Context information is received mainly from two sources. Firstly, there is a context discovery module which collects sensor data to make sense of the *physical* world in which the system is running, and also monitors network services to make sense of the *logical* environment. Logical context changes can also be signalled internally by the running application, for instance when switching to secure or logging mode. Having all context changes at hand, the context management module analyses the current situation and might chose to prioritise some of the context changes, defer others for later application, drop context changes that have become outdated, and solve possible conflicts stemming from contradictory information and adaptation policies (for instance, *increasing* fan speed due to overheating, versus *reducing* fan speed due to low battery charge). It then commits a coherent set of changes to the active context representation, which directly affects application behaviour.

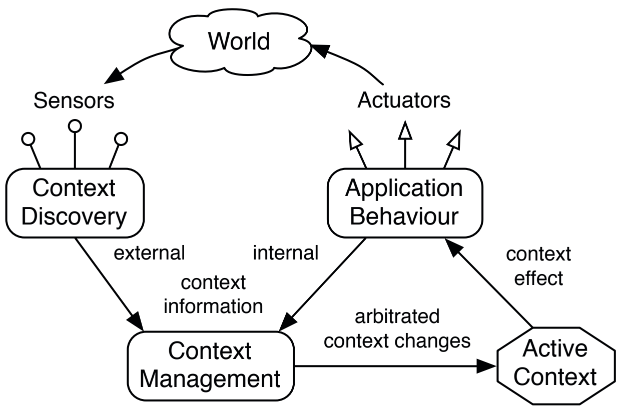


Fig. 1. General architecture for context-aware systems

```
Context* landscape = [[Context alloc] initWithName: @"Landscape"];
[CONTEXT addContext: landscape];
```

Snippet 1: Subjective-C context definition.

The global architecture proposed here is compatible with more detailed ones such as Rainbow [6], meaning that the more refined subsystems of those architectures can be accommodated within ours. However, the presented level of detail suffices as frame of reference for the explanations that follow.

2.2 Contexts

We define *context* as an abstraction of the particular state of affairs or set of circumstances for which specialised application behaviour can be defined. The context discovery module shown in Fig. 1 is in charge of making sense of perceived data and assigning it a higher-level meaning as contexts. Table 1 shows a few examples. This mapping of data into meaningful contexts is not explored further in this paper.

Table 1. Environmental data vs. contexts as semantically-rich situations

Sensed data	Contexts
Coordinates = 50°50'N 4°21'E	In Brussels
Battery charge = 220 mAh	Low battery charge
Idle cycles = 11 MHz	High CPU load
Z axis = 0.03	Landscape orientation

The notion of context put forward by Subjective-C is in line with dictionary definitions such as "the situation within which something exists or happens, and that can help explain it"[1] and "the interrelated conditions in which something exists or occurs".[2] This is in contrast to the more general definition of context by Hirschfeld et al. [10] as "any information which is computationally accessible".

In Subjective-C, contexts are reified as first-class objects. A typical context definition is shown in Snippet 1. The landscape context is allocated and given a name. Contexts are declared to the system's context manager by means of the `addContext:` call. As exemplified in Sections 2.3 and 2.4, the `Landscape` context can be used by a smartphone application whose behaviour depends on the spatial orientation of the device.

At any given time, contexts can be either *active* or *inactive*. Active contexts represent the currently perceived circumstances in which the system is running, and only these active contexts have an effect on system behaviour. Snippet 2 shows the way a context can be activated and deactivated. We call such changes from one state to the other *context switches*. Context switches are carried out by the context manager in response to incoming context changes.

[1] `http://dictionary.cambridge.org/dictionary/british/context_1`
[2] `http://merriam-webster.com/dictionary/context`

```
[CONTEXT activateContextWithName: @"Landscape"];
[CONTEXT deactivateContextWithName: @"Landscape"];
```

Snippet 2: Context activation and deactivation.

```
#context Landscape
- (NSString*)getText() {
  return [NSString stringWithString:@"Landscape view"];
}
```

Snippet 3: Context-specific method definition.

2.3 Contextual Behaviour

Subjective-C concentrates on algorithmic adaptation, allowing the definition of behaviour that is specific to certain execution contexts. Programmers can thereby define behaviour that is more appropriate to those particular contexts than the application's default behaviour.

In Subjective-C, defining context-dependent behaviour is straightforward. Adapted behaviour can be defined at a very fine granularity level, namely on a per-method basis. To define methods that are specific to a context, a simple annotation suffices. Snippet 3 illustrates a typical context-specific method definition. The `#context` annotation lets Subjective-C know that the `getText` method definition is specific to the given named context. This version of `getText` should be invoked only when the device is in `Landscape` position. The method is not to be applied under any other circumstances. The general EBNF syntax for context-specific method definitions is as follows:

```
#context ([!]contextName)+
methodDefiniton
```

This is one of the two syntactic extensions Subjective-C lays over Objective-C; the other one is explained in Section 2.4. As can be observed in this general form, it is possible to specialise a method on more than one context. It suffices to provide multiple context names after the `#context` keyword that precedes the method definition. The method is applicable only when all its corresponding contexts are active simultaneously.

As a convenience, Subjective-C introduces method specialisation on the *complement* of a context by means of the negation symbol (`!`). Such complementary method specialisations mean that the method is applicable only when the given context is inactive. Complementary specialisations serve as a shortcut to explicitly defining a complementary context object and associated management policies.

2.4 Behaviour Reuse

For most cases it is of little use to provide a means to define context-specific behaviour but no means to invoke at some point the original default behaviour

```
@implementation UILabel (color)
#context Landscape
- (void)drawTextInRect:(CGRect)rect{
  self.textColor = [UIColor greenColor];
  [superContext drawTextInRect:rect];
}
@end
```

Snippet 4: Sample use of `superContext` construct.

as part of the adaptation. The absence of such a mechanism would lead to the reimplementation of default behaviour in overriding context-specific methods, resulting in code duplication. Subjective-C therefore permits the invocation of overridden behaviour through the `superContext` construct, which has two general forms:

```
[superContext selector];
[superContext keyword: argument ...];
```

Next to the `#context` construct explained in Section 2.3, `superContext` is the second syntactic extension of Subjective-C over Objective-C.

Snippet 4 shows an example in which the colour of a label widget changes to green when the orientation of the host device is horizontal (i.e., when the context `Landscape` is active). This example shows in passing that it is possible to modify the behaviour of stock library objects such as UILabel, which have been developed independently, and for which adaptations such as the one in Snippet 4 were not foreseen. It is possible to layer adaptations on top of *any* existing object, without access to its source code. This is made possible by Objective-C's open classes and categories.

3 Context Relations

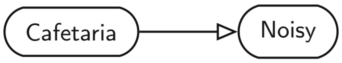
Subjective-C allows the explicit encoding of context relationships. These relationships impose constraints among contexts, which either impede activation or cause cascaded activations and deactivations of related contexts. A failure to respect the natural relationships between contexts could lead to unexpected, undesired, or erroneous application behaviour. All behaviour described in this section is part of the context management subsystem illustrated in Fig. 1.

When a context is switched, the system has to inspect all relations involving the context, checking if the change is consistent with imposed constraints, and performing other switches triggered by the requested one. This section concisely specifies the different relation types and their effect on context switching through four main methods that any context must implement: `canActivate`, `canDeactivate`, `activate` and `deactivate`.

To deal with multiple activations, every context has an *activation counter*, which the `activate` method increases, and `deactivate` decreases (if positive). Only when the counter falls down to zero is the context actually removed from the active context representation.

3.1 Weak Inclusion Relation

Sometimes the activation of a context implies the activation of a related one. For example, if domain analysis yields that cafeterias are usually noisy, then the activation of the `Cafeteria` context should induce the activation of the `Noisy` context. We say that the former context *includes* the latter one. However, the inclusion is a weak one in the sense that the contrapositive does not necessarily hold. Even though there might be no noise, the device might still be located in a cafeteria. The key here is that cafeterias are usually, though not always, noisy. Fig. 2 shows the definition of weak inclusion relations. Activating (resp. deactivating) the source context `Cafeteria` implies activating (resp. deactivating) the target context `Noisy`. The source context can always be activated and deactivated. Conversely, because it is a weak inclusion relation, the target context `Noisy` is not constrained at all by the source context `Cafeteria`. The target context can be activated and deactivated anytime, without consequences on the activation status of the source context.

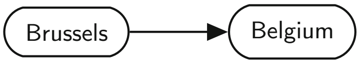


message	source behaviour	target behaviour
canActivate	YES	YES
canDeactivate	YES	YES
activate	target activate	—
deactivate	target deactivate	—

Fig. 2. Weak inclusion relation specification

3.2 Strong Inclusion Relation

In a strong inclusion relation, the activation of the source context implies the activation of the target context, as in weak inclusions. Additionally, the contrapositive holds: deactivation of the target implies automatically a deactivation of the source. For example, if the current location is `Brussels`, then necessarily the device is also located in `Belgium`. If the current location is not `Belgium`, then it is certainly also not `Brussels`. Fig. 3 shows the definition of such strong inclusion relations. As illustrated by the example, this kind of relation can be used to signal that a specific context is a particular case of a more general one.

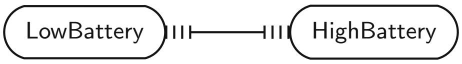


message	source behaviour	target behaviour
canActivate	target canActivate	YES
canDeactivate	YES	source canDeactivate
activate	target activate	—
deactivate	target deactivate	source deactivate

Fig. 3. Strong inclusion relation specification

3.3 Exclusion Relation

Some contexts are mutually exclusive. For instance, a network connection status cannot be `Online` and `Offline` simultaneously, and the battery charge level cannot be high and low at the same time (note however that it makes sense for two exclusive contexts such as `LowBattery` and `HighBattery` to be simultaneously inactive). This motivates the introduction of *exclusion* relations between contexts, specified in Fig. 4. Note that the exclusion relation is symmetrical.

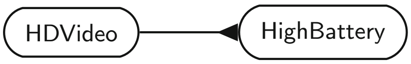


message	source behaviour	target behaviour
canActivate	target isInactive	source isInactive
canDeactivate	YES	YES
activate	—	—
deactivate	—	—

Fig. 4. Exclusion relation specification

3.4 Requirement Relation

Sometimes certain contexts require other contexts to function properly. For instance, a high-definition video decoding context `HDVideo` might work only when `HighBattery` is active. If `HighBattery` is inactive, then `HDVideo` cannot be activated either. Fig. 5 specifies this *requirement* relation between contexts.



message	source behaviour	target behaviour
canActivate	target isActive	YES
canDeactivate	YES	source canDeactivate
activate	—	—
deactivate	—	source deactivate

Fig. 5. Requirement relation specification

3.5 Context Declaration Language

For non-trivial scenarios, the programmatic definition of contexts and their relations in Subjective-C can become verbose. As an example, consider the relatively complex code to create just two contexts and an exclusion relation between them, shown in Snippet 5 (left).

Observing that it is cumbersome to describe context settings programmatically, we developed a small Domain-Specific Language (DSL) for this purpose.

`Context* on = [[Context alloc] initWithName:@"Online"];`
`Context* off = [[Context alloc] initWithName:@"Offline"];`
`[CONTEXT addContext:on];`
`[CONTEXT addContext:off];`
`[on addExclusionLinkWith:off];`
	`Contexts:`
	`Online`
	`Offline`
	`Links:`
	`Online >< Offline`

Snippet 5: Manual creation of contexts and their relations in Subjective-C (left) versus equivalent code using the context declaration language (right).

In this DSL, contexts are declared simply by naming them, and their relations established by means of the following textual notation:

– Weak Inclusion: `A -> B`
– Strong Inclusion: `A => B`
– Exclusion: `A >< B`
– Requirement: `A =< B`

The right side of Snippet 5 shows how the context set-up on the left side is obtained using the context declaration language. This language permits the edition of contexts and their relations with all the advantages brought by a DSL: it has a more intuitive notation that can be understood even by non-programmers, it results in more succinct code, and eases rapid prototyping.[3]

In its current version, Subjective-C does not check inconsistencies among context relationships at the moment they are created (for example if `A => B` on the one hand but `A >< B` on the other); as mentioned earlier, it checks for inconsistencies when contexts are switched, preventing any contradictory change. Support for earlier checks is part of our future work.

4 Implementation

Most existing COP implementations exploit meta-programming facilities such as syntactic macros and computational reflection provided by the host object model to modify method dispatch semantics, thereby achieving dynamic behaviour selection. It is no surprise that these implementations have been laid on top of dynamic languages that permit such level of flexibility.

For approaches based on more static languages such as ContextJ for Java [1], existing implementations use a dedicated compiler. This is also the case of Subjective-C, in which the compiler is just a small language transformer to plain Objective-C. However, Subjective-C does not intercept method dispatch as other approaches do. Rather, it precomputes the most specific methods that become active right after every context switch. This original implementation technique, explained in this section, is possible thanks to some of the dynamic features offered by Objective-C. Section 6 presents an efficiency comparison of the two approaches (method precomputation versus method lookup modification).

[3] A script integrated in the build process of the IDE parses the context declaration files written in the DSL and translates them into equivalent Objective-C code. The result is compiled together with regular source code files.

```
@implementation UILabel (color)
- (void)Context_Landscape_drawTextInRect:(CGRect)rect {
  self.textColor = [UIColor greenColor];
  SUPERCONTEXT(@selector(drawTextInRect:), [self drawTextInRect:rect]);
}
@end
```

Snippet 6: Context-specific version of `UILabel`'s `drawTextInRect:` method translated to plain Objective-C.

```
[MANAGER
  addMethod:@selector(Context_Landscape_drawTextInRect:)
  forClass:[UILabel class]
  forContextNames: [NSSet setWithObjects: @"Landscape", nil]
  withDefautSelector:@selector(drawTextInRect:)
  withPriority:0];
```

Snippet 7: Registration of a context-specific method.

4.1 Method Translation

Context-specific methods, explained in Section 2.3, have the same signature as the original method containing the default implementation. For instance, the `drawTextInRect:` method from Snippet 4 has the same signature as the standard method furnished by Apple's UIKit framework.[4] The intention of adapted methods is precisely to match the same messages the original method matches, but then exhibit different behaviour in response to the message.

To disambiguate method identifiers that have been overloaded for multiple contexts, and thus distinguish between the different context-dependent implementations sharing a same signature, Subjective-C uses name mangling. Name mangling is a well-known technique in which identifiers are decorated with additional information from the method's signature, class, namespace, and possibly others pieces of information to render the decorated name unique. In Subjective-C, the selector of any context-specific method is mangled by prefixing the `Context` keyword, followed by the name of all contexts on which the method has been specialised. The name of complementary contexts (explained in Section 2.3) is prefixed with `NOT`. The different name parts are separated by underscores. As an example, Snippet 6 shows how the name of the `drawTextInRect:` method from Snippet 4 is mangled. The snippet also shows the translation of the `superContext` construct, discussed further in Section 4.3.

The different method versions are registered to the context manager by automatically generated code, shown in Snippet 7. The priority index lets the context manager order method implementations to avoid ambiguities. This ordering is discussed further in Section 7.

[4] UIKit provides the classes needed to manage an application's user interface in iOS.

```
Method current_method =
 class_getInstanceMethod(affectedClass, defaultMethodName);
Method selected_method =
 class_getInstanceMethod(affectedClass, mangledMethodName);
method_setImplementation
 (current_method, method_getImplementation(selected_method));
```

Snippet 8: Reflective method replacement to achieve predispatching.

4.2 Method Predispatch

Contrary to existing COP implementations, Subjective-C does not modify the method lookup process of its host language. Rather, it determines the method implementations that should be invoked according to the currently active context at context-switching time. The chosen methods become the *active methods* in the system. The set of active methods is recalculated for every change of the active context. We call this process *method predispatch*.

Method predispatch is made possible by the ability to dynamically replace method implementations in Objective-C. As sketched in Snippet 8, the predispatcher uses the reflective layer of Objective-C to exchange method implementations.[5] The currently active implementation is replaced by a version that is most specific for the active context. It can very well be that the old and new versions are the same, in which case the method switching operation has no effect.

This implementation technique would be less easy to achieve in other members of the C language family such as C++, due to the lack of a standard reflective API that enables the manipulation of virtual method tables. A non-reflective implementation would probably involve compiler– and platform-specific pointer manipulations to patch such tables manually.

Finally, from Snippet 8 it can be observed that the default method and its context-dependent adaptations belong to the same class. Since Objective-C features open classes, methods can be added to any existing class. Open classes make it possible for Subjective-C to add context-specific methods to any user-defined, standard or built-in class, without access to its source code. Adaptability of third-party code is one of the strongest advantages brought by Subjective-C, and is another area in which other members of the C language family would fall short in implementing a similar mechanism (because they lack open classes).

4.3 Super-Context Calls

Snippet 6 shows how the `superContext` construct from Snippet 4 is translated to plain Objective-C. Snippet 9 shows the definition of the `SUPERCONTEXT` preprocessor macro used by the translated code. This macro replaces the current method implementation by the next one in the method ordering corresponding to the given class, default selector and currently active context, invokes the newly set implementation, and reverts the change to leave the system in its original state.

[5] Besides instance methods, it is also possible to manipulate class methods through `class_getClassMethod`, but the details are inessential to the discussion.

```
#define SUPERCONTEXT(_defaultSelector, _message) \
  [MANAGER setSuperContextMethod:_defaultSelector forClass:[self class]]; \
  _message; \
  [MANAGER restoreContextMethod:_defaultSelector forClass:[self class]];
```

Snippet 9: Macro definition used to translate `superContext` constructs.

5 Validation

This section summarises three case studies we developed to assess the qualities of Subjective-C to respectively create a new context-aware application from scratch, extend an existing application so that it becomes context-aware, and refactor an existing application by exploiting its internal modes of operation (i.e. logical contexts, as opposed to making it adaptable to physical changes).

Home Automation System. The goal of this case study is to build a home automation system using Subjective-C from the ground up. The system permits the use of a smartphone as remote control to regulate climatic factors such as temperature, ventilation and lighting, and to command household appliances such as televisions. The remote control communicates through the local network with a server system, which simulates these factors and appliances, in a home with a kitchen, bathroom, bedroom and living room. Each room is equipped with a different combination of windows (for ventilation regulation), heating, air conditioning and illumination systems. The remote control application adapts its user interface and behaviour dynamically according to the simulated context changes coming from the server.

This case study heavily uses the context declaration language introduced in Section 3.5. Fig. 6 shows a graphical overview for the home server implementation. The relatively complex relations for this proof-of-concept system show that the definition of a dedicated context declaration language is justified. In a full-fledged home automation system the contexts and their relations could be even more intricate, and defining all entities programmatically would result in complex code.

Device Orientation. The goal of this case study is to extend an existing Objective-C application with context-oriented constructs. The original Device Orientation application is a proof-of-concept whose basic functionality is to display a text label which dynamically adjusts its display angle so that it remains parallel to the ground, regardless of the physical device orientation. The application extension consists in changing the text and colour of the label according to orientation changes in the x and z axes. The guideline is to be able to introduce said extensions with minimal intervention of the original source code. Several code snippets from this case study are used throughout this paper.

Regarding efficiency, Device Orientation switches contexts as frequently as every 100 milliseconds to keep the label constantly parallel to the ground. We observed no apparent slowdown with respect to the original application: the label adapts swiftly to orientation changes.

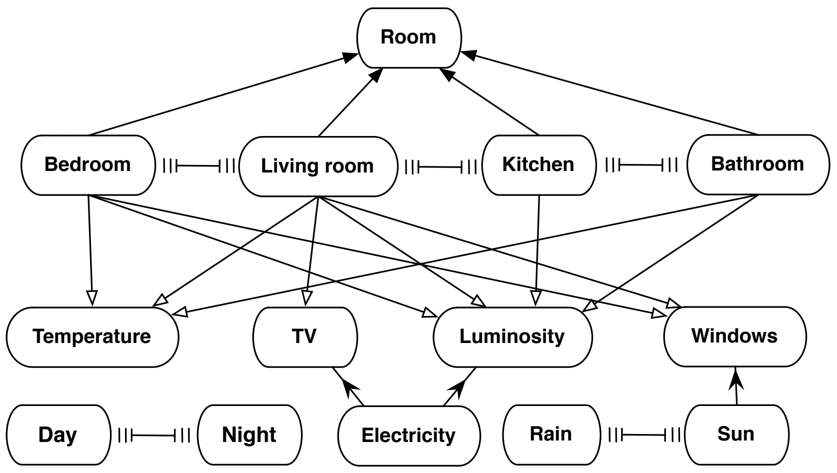


Fig. 6. Context and relations in the Home Automation case study

Accelerometer Graph. The goal of this case study is to perform a behaviour-preserving refactoring of an existing application using contexts, to assess the impact on source code quality. The chosen application is Accelerometer Graph, developed by Apple to illustrate the use of filters to smooth the data obtained from an iPhone's accelerometer. The application presents a graph of read accelerometer data against time. It can work in standard (default) or adaptive mode. Independently from these modes, it can work in low-pass or high-pass mode. Due to these operation modes, the original application presents some cases of conditional statements related to the operation mode, and code duplication. The refactored version avoids the conditionals and the duplication by modelling the different operation modes as contexts.

From the experience gathered in the described case studies, we have observed that extensibility and maintainability are particularly strong points of Subjective-C. These main strengths come from the separation of concerns between the base application and its context-specific adaptations. Subjective-C allows the adaptation of any method of the application, and all such method adaptations that correspond to a given context can be modularised and furnished as a single unit. Further details and in-depth discussion of the case studies are provided by Libbrecht and Goffaux [13].

6 Benchmarks

As mentioned in Section 1, one of the main advantages of COP is that it offers an alternative to hard-coded conditional statements. By helping to avoid such statements, COP renders code more reusable and maintainable. However, this advantage would be nullified if the penalty in performance would be prohibitively high. Therefore, to assess the cost of using COP abstractions, we measured the difference in execution time between an application that uses contexts

```
-(void) test:(int) mode {
  if (mode == 1)
    result = 1;
  else if (mode == 2)
    result = 2;
  ...
  else if (mode == N)
    result = N;
  else
    result = 0;
}
```

```
#context C1
-(void) test {
  result = 1;
}
...
#context CN
-(void) test {
  result = N;
}
-(void) test {
  result = 0;
}
```

Snippet 10: Dummy test methods in Objective-C (left) and Subjective-C (right) with $N + 1$ behavioural variants. The choice of the variant depends on the application's current operating mode and the application's current execution context, respectively.

in Subjective-C and an equivalent application that uses conditional statements in Objective-C. Our benchmark consists of a dummy application that runs in $1+N$ possible operation modes (the default one plus N variants). In Objective-C these modes are encoded as integers stored in a global variable, on which application behaviour depends. In Subjective-C the alternatives are represented as contexts. Snippet 10 illustrates the two approaches. For the sake of the benchmark, the `test` method merely produces a side effect by assigning the `result` global variable. Since the execution cost of such an assignment is negligible, the cost of `test` is dominated by the cost of method invocation. Additionally, the Objective-C solution incurs the cost of testing the branches in the conditional statement. For sufficiently high values of N, this cost becomes considerable. In Subjective-C there is no additional cost associated to the choice of a behavioural variant during method invocation, because such choice has been precomputed at context-switching time.

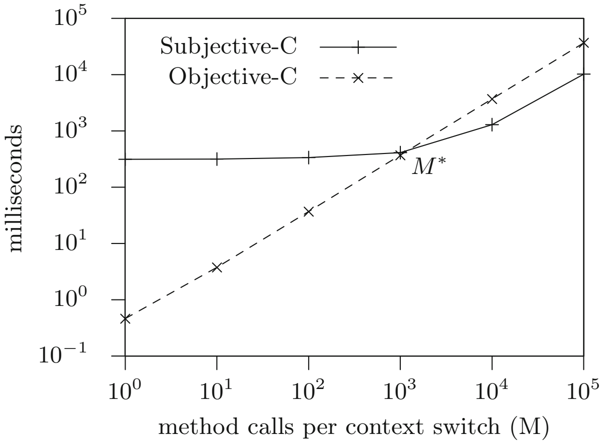
Naturally, the question is how the costs of conditional statement execution in Objective-C and context switching in Subjective-C compare. To measure the difference, we invoke the `test` method M times for every context change, as shown in Snippet 11. In Objective-C, `test` execution time depends on the number of branches K that need to be evaluated in the conditional statement. In Subjective-C, `test` execution time is constant, but at context-switching time it is necessary to iterate over the first K possible methods to find the one that needs to be activated. The results of the comparison between these two approaches are shown graphically in Fig. 7a for $N = 50$ and $K = 50$. The test application was run in debugging mode on an iPhone 3GS with iOS 4.0. In the case illustrated in Fig. 7a, context switching reaches the efficiency of conditional statement execution at about 1150 method calls per context switch. Beyond this point, Subjective-C is more efficient than Objective-C; the execution time in

```
for (int i = 0; i < 1000; i++) {
  if (i % 2)
    [CONTEXT activateContextWithName:@"CK"]; // mode = K;
  else
    [CONTEXT deactivateContextWithName:@"CK"]; // mode = 0;
  for (int j = 0; j < M; j++)
    [self test];
}
```

Snippet 11: Code to measure the relative cost of context changes with respect to context-dependent method invocation in Subjective-C; the Objective-C counterpart is analogous and is therefore just suggested as comments.

both approaches will tend to grow linearly, although Objective-C will have a considerably higher slope due to conditional statement execution,[6] besides the cost of method invocation which is incurred in both approaches. Fig. 7b summarises the intersection points for various values of N and K, including the case of Fig. 7a.

(a)



(b)

N	K	M^*
5	1	8230
10	1	4708
50	1	3405
5	5	4521
10	10	2653
50	50	1148

Fig. 7. Performance comparison of Objective-C and Subjective-C; (a) illustration with logarithmic scale for the case $N = 50$, $K = 50$, and (b) summary of efficiency meeting points M^* for various values of N and K

The benchmarks just discussed use contexts that are not linked through any of the relations introduced in Section 3. Though not shown here for space limitations, we have carried out a few benchmarks to assess the impact of relations on context activation [13]. The presence of exclusion relations increases slightly the time of activation (i.e. an extra check for every excluded context); in inclusion

[6] This difference is not apparent in Fig. 7a because of the logarithmic scale.

relations, the (de)activation must be not only accepted but also propagated on the chain of included contexts. Processing inclusion relations is about 4 times more costly than processing exclusion relations.

In another benchmark we assessed the cost of activation according to the number of methods associated to the switched context. As can be expected, the activation time increases linearly with the number of methods (e.g. if a context has twice as many methods, it takes twice as much time to switch).

Yet in one more benchmark we evaluated the activation time according to number of contexts that specialise a given method. In our implementation, the execution time grows linearly with the number of contexts that implement the same method, because finding the most specific method involves a linear search in a list of available methods.[7]

The efficiency of Subjective-C depends highly on its usage. Most benefits are obtained for contexts that are switched infrequently with respect to the rate of usage of affected methods. Fortunately, this is the case for most common scenarios, because context changes are usually linked to physical phenomena such as orientation changes, temperature changes, battery charge changes, network connections, and so forth. In particular, in each of the three case studies described in Section 5, the Subjective-C implementation did not give rise to apparent performance penalties. This being said, we can think of a few kinds of contexts which could be switched very rapidly, for instance software memory transactions implemented as contexts [7], used in tight loops. For these cases the penalty incurred by Subjective-C could become detrimental to overall performance. However, for most practical cases we can conclude that COP abstractions do not incur a performance penalty that would bar them from mobile platform programming.

7 Limitations and Future Work

Even though Subjective-C is usable for application development on mobile devices as suggested in Section 5, it still has rough edges we need to iron out. This section describes the most salient ones, starting with the more technical and going over to the more conceptual.

Super-Context Translation. A caveat of the implementation presented in Section 4.3 is the impossibility to retrieve the return value from a `superContext` message. Our current solution consists in having a different syntax when the return value is needed, which complements the definition of the `superContext` construct given in Section 2.4:

```
[superContext selector] => variable;
[superContext keyword: argument ...] => variable;
```

This syntax is translated to a variant of the `SUPERCONTEXT` macro which expands the message as `variable = _message` instead of just `_message`.

[7] We have invested no effort yet in improving this straightforward implementation.

The additional syntax shown here is due to a particularity of our current implementation, but we see no fundamental reason why it could not be avoided in a more sophisticated version of the compiler.

Context Scope and Concurrency. Context activation in Subjective-C is global. All threads share the same active context and see the effects of context switching performed by other threads. This can give rise to race conditions and behavioural inconsistencies if concurrent context switches occur at inappropriate execution points [9]. However, note that this issue does not stem from a conceptual error. As discussed in Section 2.1, any computing device is embedded in a physical and logical execution environment, which all applications running on the device share. For instance, it is natural that all applications become aware of a `LowBattery` condition, or a reorientation of the device to `Landscape` position. Rather than avoiding the problem of shared contexts altogether by limiting the scope of context effects to individual threads, our open research challenge and line of future work consists in detecting the execution points at which shared context changes are safe to perform, whether automatically or with some sort of assistance such as source code annotations.

Nevertheless, having made a case for global contexts, we do believe that adding support for local contexts, representing the running conditions of particular threads, would be a useful addition to Subjective-C. For instance, one thread could run in `Debug` or `Tracing` mode simultaneously with other threads running in default mode.

Behaviour Disambiguation. Whenever multiple methods are applicable for a given message and active context configuration, the context manager should be able to deterministically define which of the methods is to be invoked. For example, suppose there are two versions of `UILabel`'s `drawTextInRect:` method, respectively specialised on the `Landscape` and `LowBattery` contexts. If both contexts are active at any given time, it is unclear which of the two versions of `drawTextInRect:` should be applied first.

Currently, the choice is based on a priority assigned to every method. Default methods have always less priority than context-dependent methods. For two context-dependent methods, the priority is given by the order in which the compiler comes across the method definitions.[8] Hence, the later a method is found, the higher its priority. Clearly this ad hoc mechanism to automatically determine priorities is insufficient. A better solution is to define a version of the ambiguous method specialised on the set of conflicting contexts (in the example, `Landscape` and `LowBattery`), and resolve the ambiguity manually in that specific method implementation. This solution cannot be used for every possible combination as this would result in a combinatorial explosion. A line of research is to help predicting which ambiguities arise in practice by analysing context relations, and provide declarative rules to resolve remaining ambiguities based on domain-specific criteria.

[8] Note that Objective-C open classes can be defined across multiple files.

8 Related Work

COP-like ideas for object-oriented systems can be traced back as far as 1996. The Us language, an extension of the Self language with subjective object behaviour [17], inspired our work since the early stages. In Us, subjectivity is obtained by allowing multiple perspectives from which different object behaviour might be observed. These perspectives are reified as *layer* objects, and hence, Us layers are akin to Subjective-C contexts.

The contemporary notion of COP has been realised through a few implementations, in particular ContextL [4] which extends CLOS [2], Ambience [9] which is based on AmOS [8], and further extensions of Smalltalk [10], Python [14] and Ruby,[9] among others. Most existing approaches, with the exception of Ambience, seem to be conceptual descendants of ContextL, and therefore share similar characteristics. None of these COP languages is similar to Subjective-C in that they affect method dispatch semantics to achieve dynamic behaviour selection, whereas Subjective-C uses method predispatching, introduced in Section 4.2.

Subjective-C is inspired on our previous work with Ambience. In particular, both languages use the notion of contexts as objects representing particular run-time situations, described in Section 2.2. Further, contexts in Ambience are also global and shared by all running threads, an issue discussed in Section 7. Whereas in Ambience it would not be difficult to adapt the underlying AmOS object model to support thread-local contexts, in Subjective-C we do not control the underlying object system inherited from Objective-C.

An issue barely tackled by existing COP approaches is the high-level modelling of contexts and their conditions of activation. Subjective-C makes a step forward in this direction by introducing different types of relations between contexts, explained in Section 3. This system of relations bears a strong resemblance to some of the models found in Software Product Line Engineering (SPLE). Unfortunately, thus far SPLE has focused mostly on systems with variability in static contexts [15]. Variability models such as Feature Diagrams (FDs) [11] and their extensions have not been geared towards capturing the dynamism of context-dependent behavior. More recent work on variability models acknowledges the concept of dynamic variability in SPLE [3,5,12].

Also related to COP in general, and Subjective-C in particular, is the family of dynamic Aspect-Oriented Programming approaches. PROSE [16] for instance is a Java-based system using dynamic Aspect-Oriented Programming (AOP) for run-time adaptability. Since dynamic aspects can be woven and unwoven according to context, dynamic AOP can be a suitable option to obtain dynamic behaviour adaptation to context. Dynamic AOP buys flexibility (for instance, the ability to express join points that capture only certain invocations of a given method, instead of every invocation) at the expense of more conceptual and technical complexity (e.g. additional join point language and abstractions for aspect definition).

[9] http://contextr.rubyforge.org

9 Conclusions

The field of Context-Oriented Programming (COP) was born in response to a lack of adequate programming abstractions to develop adaptable applications that are sensible to their changing execution conditions. Observing that no existing COP language allowed us to experiment with context-oriented mobile application programming, we set out to develop an extension of one of the most widely used languages for mobile devices, namely Objective-C. The result is the Subjective-C language,which furnishes dedicated language abstractions to deal with context-specific method definitions and thus permits run-time behavioural adaptation to context. Subjective-C objects are less "objective" than those of Objective-C in that their expressed behaviour does not depend entirely on the messages they receive, but also on the current execution context.

Subjective-C goes beyond existing COP approaches by providing an explicit means to express complex context interrelations. The different relations have a corresponding graphical depiction and textual representation, to ease their description and communication among developers and domain experts. Although we cannot guarantee that the set of supported relation types is complete enough to express all relevant context settings, we have found it to be sufficient in practice for now.

Subjective-C introduces an original implementation technique that trades context switching efficiency for method execution efficiency. Our experience with the validation cases shows that this technique results in no noticeable efficiency penalties, despite the relatively resource-constrained host platforms on which these applications run. Even more, under some circumstances efficiency is improved as compared to using plain Objective-C. From the same experience we have observed that Subjective-C seems to achieve its ultimate software engineering goals, which are improved modularisation, increased readability, reusability, maintainability and extensibility of context-aware software.

Acknowledgements

This work has been supported by the ICT Impulse Programme of the Institute for the encouragement of Scientific Research and Innovation of Brussels, and by the Interuniversity Attraction Poles Programme, Belgian State, Belgian Science Policy. We thank the anonymous reviewers for their comments on an earlier version of this paper.

References

1. Appeltauer, M., Hirschfeld, R., Haupt, M., Masuhara, H.: ContextJ: Context-oriented programming for Java. Computer Software of The Japan Society for Software Science and Technology (June 2010)
2. Bobrow, D., DeMichiel, L., Gabriel, R., Keene, S., Kiczales, G., Moon, D.: Common Lisp Object System specification. Lisp and Symbolic Computation 1(3/4), 245–394 (1989)

3. Cetina, C., Haugen, O., Zhang, X., Fleurey, F., Pelechano, V.: Strategies for variability transformation at run time. In: Proceedings of the International Software Product Line Conference, pp. 61–70. Carnegie Mellon University, Pittsburgh (2009)
4. Costanza, P., Hirschfeld, R.: Language constructs for context-oriented programming: an overview of ContextL. In: Proceedings of the Dynamic Languages Symposium, pp. 1–10. ACM Press, New York (2005); co-located with OOPSLA 2005
5. Desmet, B., Vallejos, J., Costanza, P., Kantarcioglu, M., D'Hondt, T.: Context-oriented domain analysis. In: Kokinov, B., Richardson, D.C., Roth-Berghofer, T.R., Vieu, L. (eds.) CONTEXT 2007. LNCS (LNAI), vol. 4635, pp. 178–191. Springer, Heidelberg (2007)
6. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer 37(10), 46–54 (2004)
7. González, S., Denker, M., Mens, K.: Transactional contexts: Harnessing the power of context-oriented reflection. In: International Workshop on Context-Oriented Programming, July 7, pp. 1–6. ACM Press, New York (2009)
8. González, S., Mens, K., Cádiz, A.: Context-Oriented Programming with the Ambient Object System. Journal of Universal Computer Science 14(20), 3307–3332 (2008)
9. González, S., Mens, K., Heymans, P.: Highly dynamic behaviour adaptability through prototypes with subjective multimethods. In: Proceedings of the Dynamic Languages Symposium, pp. 77–88. ACM Press, New York (2007)
10. Hirschfeld, R., Costanza, P., Haupt, M.: An introduction to context-oriented programming with contextS. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007. LNCS, vol. 5235, pp. 396–407. Springer, Heidelberg (2008)
11. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (foda) feasibility study. Tech. Rep. CMU/SEI-90-TR-21, Carnegie-Mellon University Software Engineering Institute (November 1990)
12. Lee, J., Kang, K.C.: A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. In: Proceedings of the International Software Product Line Conference, pp. 131–140. IEEE Computer Society Press, Los Alamitos (2006)
13. Libbrecht, J.C., Goffaux, J.: Subjective-C: Enabling Context-Aware Programming on iPhones. Master's thesis, Ecole Polytechnique de Louvain, UCLouvain (June 2010)
14. von Löwis, M., Denker, M., Nierstrasz, O.: Context-oriented programming: Beyond layers. In: Proceedings of the 2007 International Conference on Dynamic languages, pp. 143–156. ACM Press, New York (2007)
15. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, Heidelberg (2005)
16. Popovici, A., Gross, T., Alonso, G.: Dynamic weaving for aspect-oriented programming. In: Proceedings of the International Conference on Aspect-Oriented Software Development, pp. 141–147. ACM Press, New York (2002)
17. Smith, R.B., Ungar, D.: A simple and unifying approach to subjective objects. Theory and Practice of Object Systems 2(3), 161–178 (1996)
18. Vallejos, J., González, S., Costanza, P., De Meuter, W., D'Hondt, T., Mens, K.: Predicated generic functions. In: Baudry, B., Wohlstadter, E. (eds.) SC 2010. LNCS, vol. 6144, pp. 66–81. Springer, Heidelberg (2010)

The Level-Agnostic Modeling Language

Colin Atkinson, Bastian Kennel, and Björn Goß

Chair of Software Technology, University Mannheim
A5, 6 B 68131 Mannheim, Germany
{Colin.Atkinson,Bastian.Kennel,
Bjoern.Goss}@informatik.uni-mannheim.de

Abstract. As an alternative modeling infrastructure and paradigm, multi-level modeling addresses many of the conceptual weaknesses found in the four level modeling infrastructure that underpins traditional modeling approaches like UML and EMF. It does this by explicitly distinguishing between linguistic and ontological forms of classification and by allowing the influence of classifiers to extend over more than one level of instantiation. Multi-level modeling is consequently starting to receive attention from a growing number of research groups. However, there has never been a concrete definition of a language designed from the ground-up for the specific purpose of representing multi-level models. Some authors have informally defined the "look and feel" of such a language, but to date there has been no systematic or fully elaborated definition of its concrete syntax. In this paper we address this problem by introducing the key elements of a language, known as the Level-Agnostic Modeling Language (LML) designed to support multi-level modeling.

Keywords: Multi-Level Modeling, Modeling Language.

1 Introduction

Since it kick started the modern era of model driven development in the early 90s, the concrete syntax of the structural modeling features of the UML has changed very little. Almost all the major enhancements in the UML have focused on its abstract syntax and infrastructure [1]. Arguably, however, the original success of the UML was due to its concrete syntax rather than its abstract syntax as the former has been adopted in many other information modeling approaches, such as ontology and data modeling [9]. Today, discussions about the concrete syntax of languages usually take place in the context of domain specific modeling languages [3]. Indeed, domain specific languages are increasingly seen as one of the key enabling technologies in software engineering. While the importance of domain specific languages is beyond doubt, universal languages that capture information in a domain independent way still have an important role to play. They not only provide domain-spanning representations of information, they can tie different domain specific languages together within a single framework.

Existing universal languages such as the UML are not set up to fulfill this role. The UML's infrastructure not only has numerous conceptual weaknesses [5], it does not satisfactorily accommodate the newer modeling paradigms that have become popular in recent years, such as ontology engineering [2]. In this paper we present a proposal for a new universal modeling language that addresses these weaknesses. Known as the Level-agnostic Modeling language, or LML for short, the language is intended to provide support for multi-level modeling, seen by a growing number of researchers as the best conceptual foundation for class/object oriented modeling [6], [7], [8]. It is also intended to accommodate other major knowledge/information modeling approaches and to dovetail seamlessly with domain specific languages. Since the core ideas behind multi-level modeling have been described in a number of other publications, in this paper we focus on the concrete syntax of the LML and only provide enough information about the semantics and abstract syntax to make it understandable. The language is essentially a consolidation and extension of the notation informally employed by Atkinson and Kühne in their various papers on multi-level modeling [6], [11], [12]. Some of the concrete goals the LML was designed to fulfill are as follows:

To be UML-like. Despite its weaknesses, the UML's core structural modeling features, and the concrete syntax used to represent them, have been a phenomenal success and have become the de facto standard for the graphical representation of models in software engineering. To the greatest extent possible the LML was designed to adhere to the concrete syntax and modeling conventions of the UML.

To be level agnostic. The language was designed to support multi-level modeling based on the orthogonal classification architecture described by Atkinson, Kühne and others [6], [7], [12]. The core requirement arising from this goal is the uniform (i.e. level agnostic) representation of model elements across all ontological modeling levels (models) in a multi-level model (ontology).

To accommodate all mainstream modeling paradigms. Although UML is the most widely used modeling language in software engineering, in other communities other languages are more prominent. For example, in the semantic web and artificial intelligence communities ontology languages like OWL [9] are widely used, while in the database design community Entity Relationship modeling languages such as those proposed by Chen [13] are still important. A key goal of the LML therefore is to support as many of these modeling paradigms as possible, consistent with the overall goal of being UML-oriented.

To provide simple support for reasoning services. The lack of reasoning services of the kind offered by languages such as OWL is one of the main perceived weaknesses of UML. However, there are many forms and varieties of reasoning and checking services provided by different tools, often with non-obvious semantics. One goal of the LML is therefore to provide a foundation for a unified and simplified set of reasoning and model checking services.

Unfortunately, due to space restrictions it is only possible to provide a general overview of LML's features, and it is not possible to discuss LML's support for reasoning services in any detail. In the next section we provide a brief overview of multi-level modeling and the infrastructure that underpins the LML. In section 3 we then present the main features of the language in the context of a small case study motivated by the movie Antz. In section 4 we discuss logical relationships, which are one of the foundations for the reasoning services built on the LML, before concluding with some final remarks in section 5.

2 Orthogonal Classification Architecture

As illustrated in Fig. 1, the essential difference between the orthogonal classification architecture and the traditional four level modeling architecture of the OMG and EMF is that there are two classification dimensions rather than one. In the linguistic dimension, each element in the domain modeling level (L1) is classified according to its linguistic type in the L0 level above. Thus, elements such as *AntType*, *Ant*, *Queen* and the relationships between them are defined as instances of model elements in the L0 level according to their linguistic role or form. In the ontological dimension, each element in an ontological level (e.g. O1, O2, ..) is classified according to its ontological type in the ontological level above (i.e. to the left in Fig. 1). Thus *Bala* is an ontological instance of *Queen,* which in turn is an ontological instance of *AntType*. The elements in the top level in each dimension (with the label 0) do not have a classifier. The bottom level in Fig. 1 is not part of the modeling infrastructure, per se, but contains the entities that are represented by the model element in L1. Like the bottom level of the OMG and EMF modeling infrastructures, therefore, it can be regarded as representing the “real world”.

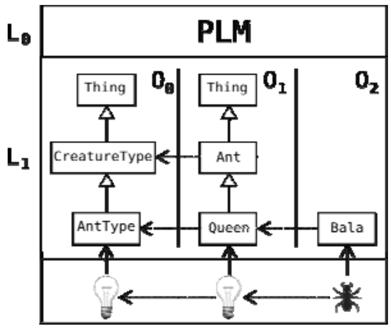


Fig. 1. Multi-Level Modeling Example

In principle, it is possible to have an arbitrary number of linguistic levels, but in practice the arrangement shown in Fig. 1 is sufficient. In our orthogonal classification architecture there are only two linguistic levels and only one of these, L1, is divided into ontological levels. The top level linguistic model therefore defines the underlying representation format for all domain-oriented and user-defined model content in the L1 level, and spans all the ontological levels in L1. For this reason we refer to it as the Pan-Level Model or PLM for short. In effect, it plays the same role as the MOF and Ecore in the OMG and EMF modeling infrastructures, but its relationship to user models respects the tenet of strict meta modeling [5].

The Pan Level Model (PLM) defines the abstract syntax of the language that we introduce in the ensuing sections [10]. The LML therefore essentially defines a concrete syntax for the PLM. To simplify discussions about content organized in the way shown in Fig. 1 we use the terms “ontology” and “model” in a specific way. We call all the modeling content in the L1 level, across all ontological levels, an “ontology”, while we call the modeling content within a single ontological level a “model”. This is consistent with common use of the terms since models in the UML are usually type models [14] and ontologies often define the relationships between instances and types (i.e. the relationship between two ontological levels or models).

There are two basic kinds of elements in an LML ontology: *InstantiatableElements* and *LogicalElements,* each of which has a *name* attribute and an *owner* attribute. *InstantiatableElements* are the core modeling elements that fulfill the role of classes, objects, associations, links and features in traditional modeling languages such as the UML, OWL or ER languages. The main difference is that they fulfill these roles in a level-agnostic way. *LogicalElements* on the other hand represent classification, generalization and set theoretic relationships between *InstanstiatableElements*. *InstantiatableElements* come in two basic forms, *Clabjects*, which represent classes, objects, associations and links, and *Features* that play a similar role to features in the UML – that is, attributes, slots and methods. *Field* is the subclass of *Feature* that plays the role of attributes and/or slots, while *Method* is the subclass of *Feature* that plays the role of methods. *Clabjects*, in general, can simultaneously be classes and objects (hence the name – a derivative of “class” and “object”). In other words they represent model elements that have an instance facet and a type facet at the same time. There are two kinds of *Clabjects*, *DomainEntities* and *DomainConnections*. *DomainEntities* are *Clabjects* that represent core domain concepts, while *DomainConnections*, as there name implies, represent relationship between *Clabjects*. They play the role of associations and links, but in a way that resembles association classes with their own attributes and slots.

3 Clabjects

The concrete representation of clabjects in LML is based on the notational conventions developed by Atkinson and Kühne [11] which in turn are based on the UML. Fig. 2 shows an example multi-level model (i.e. ontology) rendered using the LML concrete syntax. Inspired by the film Antz, this contains three ontological levels (i.e. models) organized vertically rather than horizontally as in Fig. 1.

In its full form a clabject has three compartments: a header compartment, a field compartments and a method compartment. The header is the only mandatory compartment and must contain an identifier for the clabject, shown in bold font. This can be followed, optionally, by a superscript indicating the clabject's potency and a subscript indicating the ontological level (or model) that it occupies. Like a multiplicity constraint, a potency is a non-negative integer value or “*” standing for “unlimited” (i.e. no constraint). Thus the clabject *FlyingAnt*, in Fig. 2 has potency 1 and level 1, while clabject *Z* has potency 0 and level 2. The field and method compartments are both optional. They intentionally resemble the attribute and slot compartments in UML classes and objects. However, the key difference once again is that the notation for fields is level agnostic. This allows them to represent UML-like attributes, slots and a mixture of both, depending on their potency.

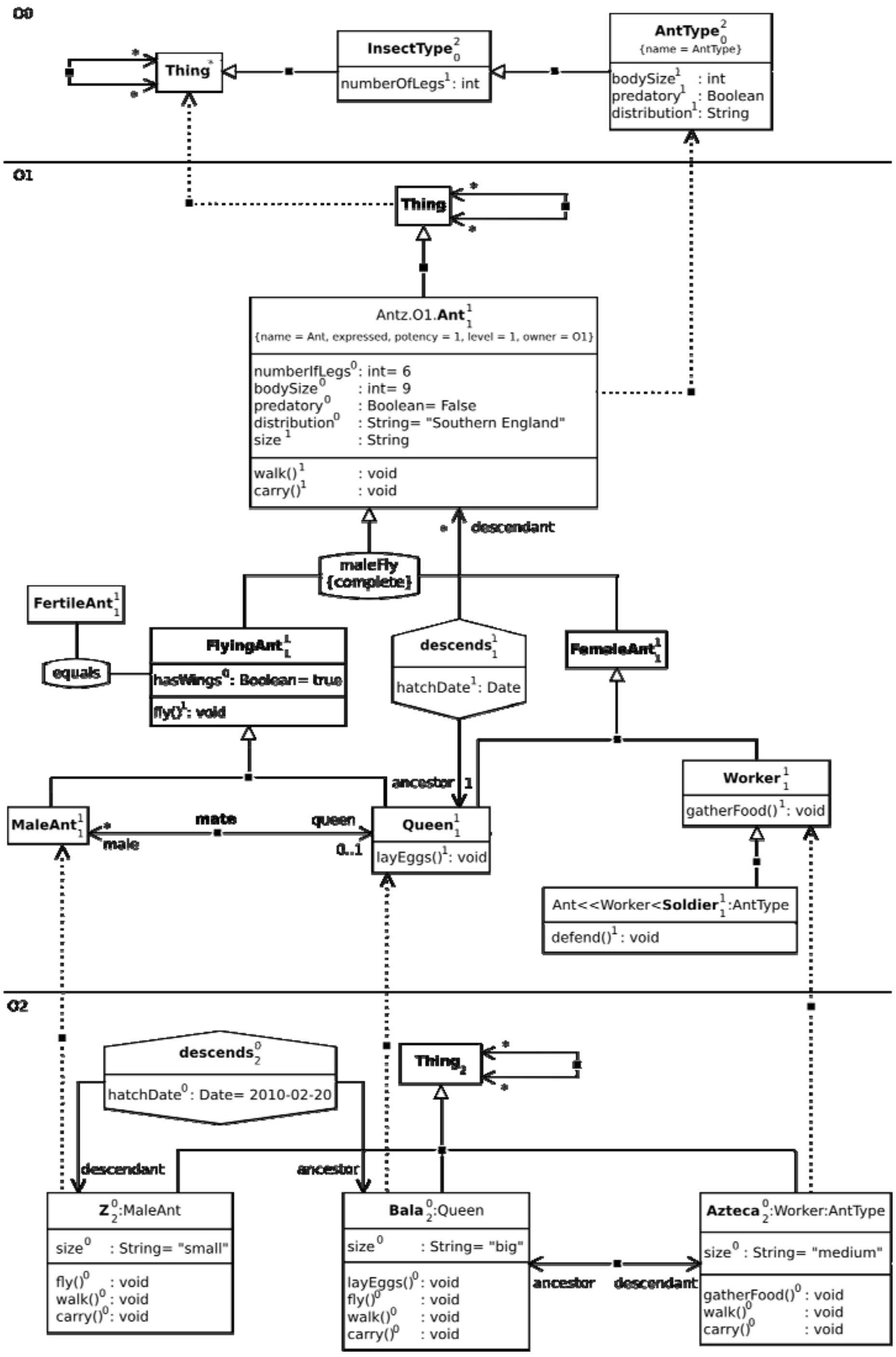


Fig. 2. Antz domain example in LML concrete syntax

Potencies on fields follow the same rules as potencies on domain entities. The only constraint is that a field cannot have a higher potency than the clabject that owns it. In general, the representation of a field has three parts: a name, which is mandatory, a type and a value. A traditional UML slot corresponds to a potency 0 field with a value, whereas a traditional UML attribute corresponds to a potency 1 field without a value.

When a value is assigned to a field with a potency greater than 0, the value represents the default value for corresponding fields of the clabject's instances. In Fig. 2, *FlyingAnt* has a potency 0 field, *hasWings,* of type *Boolean* with the value T*rue*.

Methods are represented in the same way as operations in the UML using signature expressions that include the name, the input parameters and the return value if there is one. The only difference is that methods in LML have a potency associated with them. As with fields, the potency of a method cannot be higher than the potency of its owning clabject.

3.1 Proximity Indication

The UML provides various notational enhancements in the header compartment of classes and objects to show their location within the instantiation, inheritance and containment hierarchies. More specifically, the ":" symbol can be used to to express the fact that an object is an instance of a class and the "::" notation can be used to show that a model element belongs to, or is contained in, another model element. However, these notations are rather ad hoc and cannot be used in a level agnostic way. For example, to show that a class is an instance of another class at a higher meta-level, a stereotype has to be "attached" to the class using the guillemot notation. Alternatively, the powertype concept can also be used to indicate that a class X is an instance of another class Y, but only if it is one of numerous subclasses of another class, Z. In contrast to the UML, the LML provides a fully level-agnostic way of representing a clabject's location in the instantiation, inheritance and containment hierarchies.

Classification (Instantiation) Hierarchy. The basic notation for representing a clabject's location in the instantiation hierarchy in the LML is the same as that used in the UML for objects. In Fig. 2, the header of the *Worker* clabject, "Worker:AntType", indicates that *Worker* is an instance of *AntType*. Of course this notation can also be applied over multiple ontological levels as shown in the header of *Azteca* ("Azteca:Worker:AntType"). It is also possible to use the "::" notation to omit one one or more clabjects in the instantiation hierarchy. For example, if it is only of relevance that *Azteca* is an instance of *AntType*, *Azteca*'s header can be reduced to "Azteca::AntType".

Generalization (Inheritance) Hierarchy. The UML provides no way of showing the ancestry of a class within its header compartment. In LML this can be done using the "<" symbol on the left hand side of a clabject's name to represent subtypeOf relationships. This is intended to resemble the white triangle at the supertype end of a generalization in the UML. As with the "::" notation, two consecutive "<" characters can be used to indicate that one or more clabjects of the inheritance hierarchy are not identified. Thus, the header of *Soldier* in Fig. 2 shows that it is a subclass of *Worker* and *Ant*, but omits the fact that it is also a subclass of *FemaleAnt*. Of course, in Fig. 2 this information is redundant in the header of *Soldier* because the inheritance hierarchy is shown explicitly using generalization relationships.

Ownership (Containment) Hierarchy. Ownership is the basis for defining namespaces. Basically, an element is a namespace for everything that it owns - that is, anything that identifies it as its owner through its owner attribute. The relationship between a clabject and its *Fields* is also ownership. *Fields* have the clabject they belong to as their owner. Ownership information can be included in the identifier of a clabject, like the *Ant* clabject in Fig. 2. In LML this is shown using the "." symbol rather than the "::" symbol as in the UML. This is consistent with the notation used in programming languages such as Java. Thus, the header of *Ant* in Fig. 2, ""Antz.O1.Ant", shows that *Ant* is contained in the model *O1* which in turn is contained in the ontology *Antz*.

3.2 Attribute Value Specifications

In traditional modeling environments, model elements only derive their attributes from one type, and in UML like languages these are shown in the compartment below the header. In a multi-level modeling environment, all model elements except those at the top ontological level (i.e. model) have two types, a linguistic one and an ontological one. The attributes of the ontological types are shown in the field compartment in the traditional way. The linguistic attributes, in contrast, are shown elsewhere in one of two different ways. The first way uses special notations or conventions for different attributes, such as the subscript and superscript notations for level and potency. The second way is to add explicit "attribute specifications" under the name of a clabject, similar to tagged value specifications in the UML. The general form of an attribute value specification, is the following

```
        b  e  me        e        b  e  me        e
```

For boolean attributes the UML convention applies. If a boolean attribute is true it is only necessary to include its name in the list. However, if this convention is used then all true boolean attributes of the clabject must be shown as well. This is because all boolean attributes that are not shown are assumed to be false. In Fig. 2 the *Ant* clabject has an attribute value specification indicating the value of it attributes.

3.3 Domain Connections

As mentioned in section 2, there are two distinct kinds of clabjects – domain entities and domain connections. The former play a role similar to classes/objects in the UML and the latter play a role similar to associations/links. As shown in Fig. 2 they are distinguished visually by different symbols - rectangles in the case of domain entities and flattened hexagons in the case of domain connections. This is intended to resemble the diamond symbol that is used to represent connections in Entity Relationship diagrams [13]. ERA diagrams can therefore easily be represented in LML. We use flattened hexagons rather than diamonds because they are ergonomically more compact.

The other major difference is that domain connections are responsible for "carrying" the lines that are used in the representation of connections. The problem with using a symbol like a hexagon to represent a domain connection is that this breaks the fundamental visualization metaphor of the UML in which relationships are mapped to edges rather than nodes. To address this problem and accommodate both the UML and ERA visualization metaphors at the same time, LML allows a domain connection to be represented in an imploded form by a "visually insignificant" dot as well as in the full exploded form as a flattened hexagon [15]. In the former case, the domain connection is still conceptually represented as a node, but the overall visual effect is that of an edge as in the UML. In Fig. 2, the domain connection *mate* is shown in this dotted form. This shows that the multiplicity constraints, role names and navigability values owned by the domain connection can be shown at the end of the appropriate lines using the standard UML conventions. In general, it is possible to present all the information that can appear in the header of the expanded form of a domain connection next to the dot. This includes the name and an attribute value specification. However, the fields and methods of a domain connection can only be shown in the expanded form, as indicated by the *descends* connections in Fig. 2.

4 Logical Elements

Logical relationships capture set theoretic relationships between clabjects and come in three basic forms – *Generalization*, *Instantiation*, and *SetRelationship*. The first kind captures subtyping/supertyping relationships between clabjects, the second kind captures classification relationships between clabjects and the third kind captures other general set theoretic relationships between clabjects.
Since they are relationships, the same basic representation options used for *DomainConnections* is also supported for *LogicalElement*s – namely, they can be rendered in an exploded form and in an imploded form as a "dot". The expanded form uses a slightly different symbol to *DomainConnections* – namely, a "rectangle" with rounded sides at the top and bottom. Another minor difference is that for logical relationships the name is optional in the expanded form.

Generalization. Generalization relationships show the basic subtype/supertype relationships in set-based models. In Fig. 2, the connection-line between *Ant* and *Thing* using the white triangular arrow (which is similar to the UML representation of generalizations) indicates that *Ant* is a subtype of *Thing* in the imploded representation. The generalization, *maleFly,* on the other hand shows generalization in an exploded form. It indicates that *FlyingAnt* and *FemaleAnt* are both subtypes of *Ant*. Besides the generalization's name the generalization symbol contains an attribute value specification indicating that *maleFly* is complete.

Instantiation. Instantiation relationships show that one clabject is an instance of another clabject. As such they are the only kind of relationships that are allowed to cross an ontological level boundary (in fact they have to) to connect two clabjects at different levels. As the instantiation between *Bala* and *Queen* in Fig. 2 shows, *Instantiation* relationships are represented in a similar way to the UML using an open-headed, dashed arrow going from the instance to the type. They can also be represented in an imploded or exploded form.

SetRelationship. Set relationships specificity some other kind of set theoretic relationship between clabject. Typical examples are *complements, inverseOf* and *equals.* The equals relationship between *FlyingAnt* and *FertileAnt* in Fig. 2 indicates that they essentially represent the same set, since every fertile ant can fly and only fertile ants can fly. As usual the name of the relationship is shown inside the shape. Set relationships do not normally have attribute value specifications.

5 Conclusion

In this paper we have presented a language, LML, designed to support multi-level modeling using the orthogonal classification architecture, with a focus on its concrete syntax. As mentioned in the introduction this work builds on the informal notation developed by Atkinson and Kühne in a series of papers. As well as supporting multi-level modeling the language has also been designed to exhibit several other important characteristics. However, for space reasons it is not possible to fully explain them all in a paper of this size.

First, to the greatest extent possible, LML was designed to support and be consistent with the notations and conventions popularized by the UML. As is hopefully evident from the discussion, LML can easily be made to have the look and feel of traditional UML. Like the UML, the LML concrete syntax is intended to be renderable in black and white and to be readily drawable by hand where necessary. There are lots of ways in which colour could enhance the information shown in LML diagrams, but this is left to individual tools and modelers.

Second, LML is also designed to support the look and feel of other important modeling paradigms as well. In particular it can support the entity relationship modeling approach pioneered by Chen for data modeling and contains all the features needed to represent OWL ontologies. Because it supports all ontological levels in a uniform and relatively rich way, the LML provides a natural representation of both instance and type knowledge. OWL, in contrast, only directly supports the latter.

Third, the language has built-in support for depicting which information in a model has been explicitly expressed by a human modeler and which has been computed by a reasoning or transformation engine. This allows the LML to support simpler, more understandable reasoning services. For example the traditional "subsumption" service offered by ontology engineering tools such as Protegé [16] essentially computes new generalization and classification relationships based on the properties of the classes and objects in the ontology. With LML, these could be offered in a more understandable way through services such as "add all generalizations" or "add all instantiation relationships". This not only breaks down the service into its constituent parts which can be invoked independently, it also allows them to be fine tuned into more useful services.

Finally, the language includes various kinds of elision symbols which can show when expressed information is not shown in a particular view of a model. In other words, it can be used to show that the underlying model or knowledge base has further expressed information that is not shown in the current diagram. This is important in clarifying whether a diagram, and queries direct to it, should be interpreted in an "open world" or a "close world" way.

Although the focus of modeling language research has turned from universal modeling languages like the UML to DSLs we believe the evolution of universal modeling languages is far from over. We hope the ideas presented in this paper will contribute towards this evolution and will help lead towards a better synergy of DSL and universal modeling languages. We also hope it will pave the way for a unification of software modeling (i.e. UML-oriented), knowledge representation (i.e. OWL-oriented) and information modeling (i.e. ER-oriented) technologies.

References

1. OMG UML 2.1.2 Infrastructure Specification, Object Management Group (OMG), Tech. Rep. (2007)
2. Gasevic, D., Djuric, D., Devedzic, V., Damjanovi, V.: Converting UML to OWL ontologies. In: Proceedings of the 13th International World Wide Web, New York, NY (2004)
3. Tolvanen, J.: MetaEdit+: Domain-Specific Modeling for Full Code Generation Demonstrated. In: Proc. 19th Ann. ACM SIGPLAN Conf. Object-Oriented Programming Systems, Languages, and Applications (2004)
4. GreenField, J., Short, K., Cook, S., Kent, S., Crupi, J.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. John Wiley and Sons, Chichester (2004)
5. Atkinson, C., Kühne, T.: Rearchitecting the UML Infrastructure. ACM Journal Transactions on Modeling and Computer Simulation 12(4) (2002)
6. Atkinson, C., Kühne, T.: Model-Driven Development: A Metamodeling Foundation. IEEE Software (2003)
7. Asikainen, T., Männistö, T.: Nivel:a metamodelling language with a formal semantics. Software and Systems Modeling 8(4), 521–549 (2009)
8. Aschauer, T., Dauenhauer, G., Pree, W.: Multi-level Modeling for Industrial Automation Systems. In: Software Engineering and Advanced Applications, Euromicro Conference, pp. 490–496 (2009)
9. OWL (2004), `http://www.w3.org/2004/OWL`
10. Atkinson, C., Gutheil, M., Kennel, B.: A Flexible Infrastructure for Multi-Level Language Engineering. IEEE Transactions on Software Engineering 35(6) (2009)
11. Atkinson, C., Kühne, T.: The essence of multilevel metamodeling. In: Gogolla, M., Kobryn, C. (eds.) UML 2001. LNCS, vol. 2185, pp. 19–33. Springer, Heidelberg (2001)
12. Atkinson, C., Kühne, T.: Reducing accidental complexity in domain models. In: Software and Systems Modeling (2007)
13. Chen, P.: The Entity-Relationship Model: Toward a Unified View of Data. ACM Transactions on Database Systems 1, 9–36 (1976)
14. Kühne, T.: Matters of (Meta-)Modeling. Journal on Software and Systems Modeling 5(4), 369–385 (2006)
15. Gutheil, M., Kennel, B., Atkinson, C.: A Systematic Approach to Connectors in a Multi-Level Modeling Environment. In: Proc. 11th Int'l Conf. Model Driven Eng. Languages and Systems (2008)
16. Protege Tool (2008), `http://protege.stanford.edu`

Debugging in Domain-Specific Modelling

Raphael Mannadiar[1] and Hans Vangheluwe[1,2]

Modelling, Simulation and Design Lab
[1] McGill University
3480 University Street
Montréal, Québec, Canada
[2] University of Antwerp
Middelheimlaan 1
2020 Antwerpen, Belgium

Abstract. An important obstacle to the wide-spread adoption of model-driven development approaches in industry is the lack of proper debugging facilities. Software debugging support is provided by a combination of language and Integrated Development Environment (IDE) features which enable the monitoring and altering of a running program's state. In Domain-Specific Modelling (DSM), debugging activities have a wider scope: *designers* debug model transformations (MTs) and synthesized artifacts, while domain-specific *modellers* debug their models, unaware of generated artifacts. This work surveys the state-of-the-art of debugging in the context of DSM and proposes a mapping between debugging concepts (e.g., *breakpoints*, *assertions*) in the software and DSM realms.

1 Introduction

DSM allows domain experts to play active roles in (software) development efforts. It provides them with means to manipulate constructs they are familiar with and to automate the error-prone and time-consuming conceptual mapping between the (often very distant) problem and solution domains. Empirical evidence suggests increases in productivity of up to one order of magnitude when using DSM and automatic artifact synthesis as opposed to traditional code-driven development approaches [9,5].

MTs[1] are used to specify the semantics of Domain-Specific Languages (DSLs) by defining interpreters or by mapping onto formalisms whose semantics is well understood such as Petri Nets, Statecharts or code. In this work, we focus on rule-based approaches, where MTs are composed of rules, each parameterized by a left-hand side (LHS) and right-hand side (RHS) pattern, an optional negative application condition (NAC) pattern, condition code, and action code. The LHS and NAC patterns respectively describe what sub-graphs should and should not be present in the source model for the rule to be applicable while the RHS pattern describes how the LHS pattern should be transformed by its application.

[1] See [2] for a detailed feature-based classification of existing MT techniques.

Further applicability conditions may be specified within the condition code while post-application actions may be specified within the action code.

The typical work flow of a DSM project consists of the specification of one or more DSLs and of the MTs that define their semantics. Subsequently, Domain-Specific models (DSms[2]) are created. In practice, DSms, MTs, and synthesized artifacts may all require debugging. In Section 2, we survey the state-of-the-art of debugging in the context of DSM. In Section 3, we review common debugging concepts such as *breakpoints* and *assertions* from the programming world, which we map onto the DSM world in Section 4. The contributions of this work are our mapping of debugging concepts from the programming to the DSM world, and the demystification of the amount of effort required to produce DSM debuggers.

2 Related Work

Little research has focused on debugging in DSM. In the numerous industrial applications of DSM presented in [9,5], the debugging of DSms, of their associated ad hoc generators, and of the synthesized artifacts is always accomplished without tool support and is performed at the code rather than the DSm level, dealing only with artifacts and modelling tool APIs. Conceptually, this equates to debugging compilers and bytecode to find and resolve issues in a coded program.

Wu et al. presented the most advanced DSm debugger to-date in [13]. By transparently building a detailed mapping between model entities and synthesized code, they allow the creator of textual DSms to re-use Eclipse's debugging facilities (e.g., setting breakpoints and stepping in DSms) without being exposed to the synthesized code or its generator. The main advantage of this technique is perhaps that it avoids the implementation of a brand new debugger. However, it is limited to textual DSLs, restricts the modeller to the Eclipse tool, assumes that generated artifacts are code, and does not consider MT debugging.

In [7], we laid the basis for extending Wu et al.'s work to visual DSLs. MT rules are instrumented such that *backward links* (or "traceability links") are maintained between constructs at different levels of abstraction during artifact synthesis. These links enable DSms to be animated and updated in real-time as their corresponding synthesized artifacts are executed. Means to further exploit these links to ease DSm and artifact debugging will be explored in Section 4.3.

Certain tools (e.g., AToM^3 [3]) enable basic MT debugging by allowing rule-by-rule execution, manual intervention when multiple rules are simultaneously applicable (across multiple matches), and model modification between rule applications. Thus, the rule designer can observe the effects of each rule in isolation and stear the MT. However, advanced functionality such as pausing the execution when arbitrary rules or patterns are encountered are not natively supported.

The inclusion of exceptions within Model Transformation Languages (MTLs) is proposed in [11]. A control flow environment for rule execution is extended with provisions for specifying exceptions (and their handlers). The technique for

[2] We refer to domain-specific modelling and model as DS*M* and DS*m*, respectively.

the modelling of interruptions by exceptions can readily be extended to support interruptions by debugging commands, as we will discuss in Section 4.1.

3 Debugging Code

Several authors have looked into common sources of bugs, what makes some of them more insidious than others, and popular debugging activities [4,14]. It seems observing system state and hand-simulation are often used for locating and resolving bugs. Means to carry out these activities are thus commonly found in modern programming languages and IDEs. Below, an overview is given of common and useful debugging facilities featured in languages and IDEs.

Print Statements. Print statements are commonly used to output variable contents and verify that arbitrary code segments are executed.

Assertions. Assertions enable the verification of arbitrary conditions during program execution. They cause the execution to be aborted when their condition fails, and compilers provide means to enable and disable them – as opposed to manual removal – to avoid undesired output or computation.

Exceptions. Exceptions are *thrown* at runtime to indicate and report on problematic system state. They may halt the execution of a program or be *caught* by *handlers* which take appropriate action. Provisions for defining new types of exceptions enable support for application-specific exceptional situations.

The language primitives above are traditionally used to create a "poor man's debugger". The facilities below are commonly provided by modern IDEs.

Execution Control. Modern IDE debuggers support continuous and line-by-line execution, as well as terminating and non-terminating interruption via "play", "step", "stop" and "pause" commands respectively. There are usually three step commands: *step over*, *step into* and *step out* (or *step return*). The first atomically executes the current statement. The second executes one sub-statement contained within the current statement, if any, thus effecting a change in scope. Advanced debuggers support stepping into seemingly atomic constructs into their corresponding lower-level representations, if any (e.g., Eclipse supports stepping through Java bytecode). Stepping out causes continuous execution until the current scope is exited. Finally, most IDEs allow running code in release (as opposed to debug) mode leaving only the play and stop commands enabled.

Runtime Variable I/O. IDE debuggers usually provide means to read (and change) global and local variables when the program's execution is paused.

Breakpoints. Breakpoints are commonly set on statements indicating that program execution should be interrupted before running the said statements.

Stack Traces. Visible when the execution is paused, stack traces display which function calls led the program into its current state and enable navigation between the scopes of any level in the call stack for debugging purposes.

4 Debugging in DSM

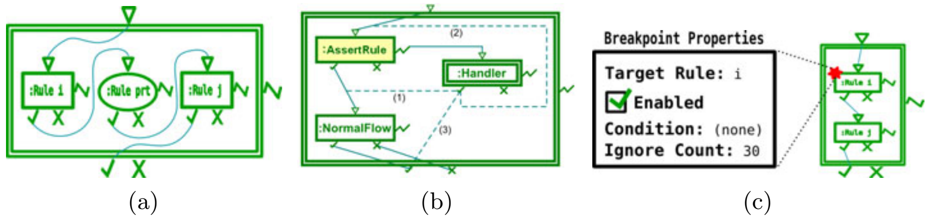
The development process in DSM has two important facets: developing models and developing MTs[3]. These facets introduce two very important differences between the programming and DSM worlds. First, in the latter realm, artifacts to debug are no longer restricted to code and include model transformations, synthesized and hand-crafted models, and other arbitrary non-code artifacts. Second, the counterparts of the common DSM activities of designing and debugging MTs (i.e., designing and debugging code compilers/interpreters) are both specialized activities in the coding realm. This section explores how the previous concepts translate into the debugging stages of both facets of DSM development.

4.1 Debugging Transformations

To carry meaning and be more than metamodels for blueprints, DSLs need associated semantics which the Model-Driven paradigm dictates should be specified as MTs [1]. Operational semantics can be specified as a collection of rules each describing the transformation between valid system states. Executing these rules effectively executes the model. Denotational semantics define the meaning of a DSL by mapping its concepts onto other formalisms for which semantics are well defined. This mapping is often encoded within code generators that transform DSms to code[4] [9,5]. We demonstrated in [7] that modelled mapping onto intermediate modelling formalisms (as opposed to directly onto code) is modular, adheres closely to the Multi-Paradigm Modelling philosophy [3], and considerably facilitates debugging by easing the specification, display and maintenance of backward links between models and synthesized artifacts. For both operational and denotational semantics, the resulting transformation model describes a flow of rule applications which may require debugging. Below, we re-visit the debugging concepts described earlier and translate them to MT debugging.

Print Statements. Print statements for MTs could be emulated by creating rules with output function calls in their action code. This entails accidental complexity: LHS and RHS patterns need to be identical to avoid modifying source models which in turn implies the necessity of loop prevention action code and/or NAC patterns. A more domain-specific solution is to enhance MTLs with printing functionality by introducing *print rules* that could be sequenced with other rules, as shown in Fig. 1a. Print rules would be parameterized by a pattern, condition code and *printing code*. Their natural semantics would be to execute the printing code if the condition code is satisfied when the pattern is found in the host model. Automatic synthesis of the contrived traditional rules described above could trivially be achieved from print rules using higher-order transformation rules[5] thus leaving the transformation execution engine

[3] Describing a formalism's operational or denotational semantics.
[4] Non-code artifacts such as XML, documentation and models may also be synthesized.
[5] Rules that take other rules as input and/or output.



Breakpoint Properties
Target Rule: i
✓ Enabled
Condition: (none)
Ignore Count: 30

Fig. 1. (a)Print rule `prt` inserted between traditional rules i and j; (b)Resuming, restarting or terminating after handling an assertion; (c)A breakpoint on a rule

unchanged. Although it may seem odd to support a language construct whose usefulness is mostly restricted to debugging purposes, we should remember that print statements (whose usefulness is mostly restricted to debugging purposes) are supported in every modern GPL.

Assertions. Assertions in transformation models can be emulated similarly to print statements: their conditions encoded in rule condition code or patterns, and exception throwing calls in rule action code[6]. This entails the same accidental complexities as those mentioned for print statements. A similar solution applies: enhancing the MTL with *assertion rules* – parameterized by a pattern, condition code and *assert code* – that could also be sequenced arbitrarily with other rules. Analogous reasoning applies regarding their semantics, their automatic translation to traditional rules and the validity of their inclusion in MTLs.

Exceptions. Exceptions and their handlers in the context of model transformation debugging were extensively studied in [11]. Syriani et al. provide a classification of several common exceptions and support user-defined exception types. They propose enhancing MTLs with *exception handler rules* to which traditional rules can be sequenced to in case of exceptions. Figure 1b shows an assertion rule sequenced to a normal rule and an exception handler.

Execution Control. Certain MT engines already support continuous and step-by-step execution. In the former, the MT is executed until none of its rules apply with user input optionally solicited when more than one rule applies simultaneously. In the latter, the user is prompted after every rule application. Stepping over in the context of rule-based MT corresponds to the execution of one (possibly composite) rule. Stepping into a composite rule should allow the modeller to execute its sub-rules one at a time. In T-Core-based languages [12], where rules are no longer atomic blocks but rather sequences of primitive operations, stepping into non-composite rules may also be sensible. Conversely, stepping out should cause the continuous execution of any remaining rules or primitives in the current scope. As for pausing an ongoing MT[7], the naive approach is immediate interruption although transactional systems might choose to commit or roll-back

[6] This exact approach is presented in [11].
[7] To our knowledge, this is only supported by VMTS [6].

the current rule before pausing while T-Core-based systems might offer pausing between primitives. Either choice has its merits and, like the step commands, is heavily dependent on MTL and engine features: it seems reasonable that pausing only occur when the system state is consistent and observable. Finally, release (as opposed to debug) mode in this context should also disable pausing and stepping functionality.

Runtime Variable I/O. MT debuggers should allow modellers to observe and change a rule's (or T-Core primitive's) inputs and outputs. Ideally, these should be presented using domain-specific constructs (and editors) rather than in their internal format. If future MT engines support on-the-fly changes (i.e., do not require the transformation model to be recompiled and/or relaunched for changes to take effect)[8], rule parameters and sequencing should themselves become viewable and editable at runtime.

Breakpoints. MT breakpoints could be set on composite and non-composite rules, and T-Core primitives, as depicted in Figure 1c.

Stack Traces. Stack traces could enable the navigation between the contexts of sub-rules and their parent rules, and between the contexts of T-Core primitives and their enclosing rule. Like stepping and pausing, they are thus clearly and closely related to the supported level of granularity of MTLs.

4.2 Creating Debuggable Artifacts

Before proceeding, we briefly review our technique for artifact synthesis from DSms. In [7], we proposed a means of generating backward links between DSms of mobile device applications, generated Google Android code, and intermediate representations. Our technique is based on triple graph rules (TGRs)[10], which provide generic means of relating constructs in LHS and RHS rule patterns. Figure 2 shows four perspectives of the same system connected via a complex "web" of links that reflects the application of the numerous TGRs that describe the DSL's denotational semantics. Though this web is not intended for direct human consumption, it can considerably facilitate the implementation of numerous components of a DSM model and artifact debugger by, amongst other things, enabling elegant two-way communication between DSms and generated artifacts, and providing implicit and navigable relationships between related concepts.

4.3 Debugging Models and Artifacts

MT debugging facilities may not be sufficient for debugging models whose semantics are specified denotationally. Running such models implies executing synthesized artifacts[9], not MTs. It is sensible to assume that in industry, DSLs (and their semantics) will often be defined by different actors than the end-users of the DSLs. Thus, we distinguish between two types of users. *Designers* are

[8] To our knowledge, no such MT tools exist yet.

[9] We restrict our attention here to programs and models.

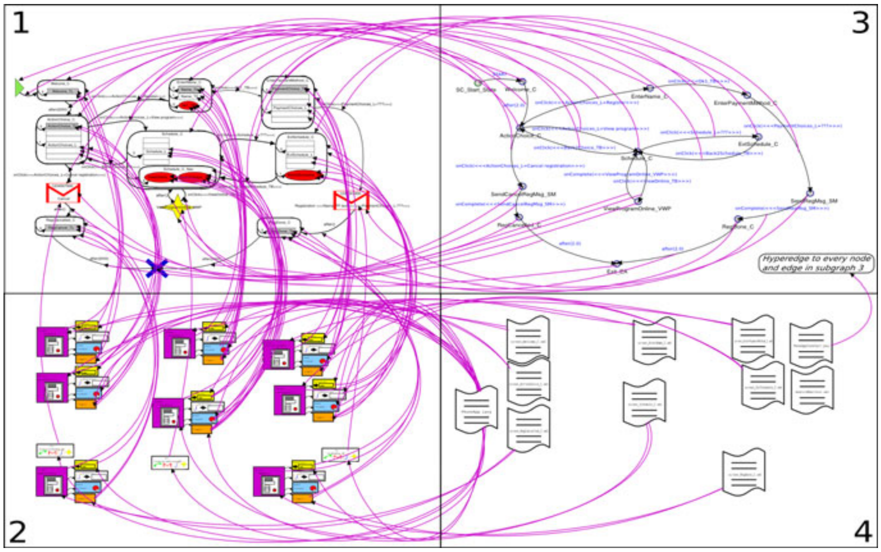
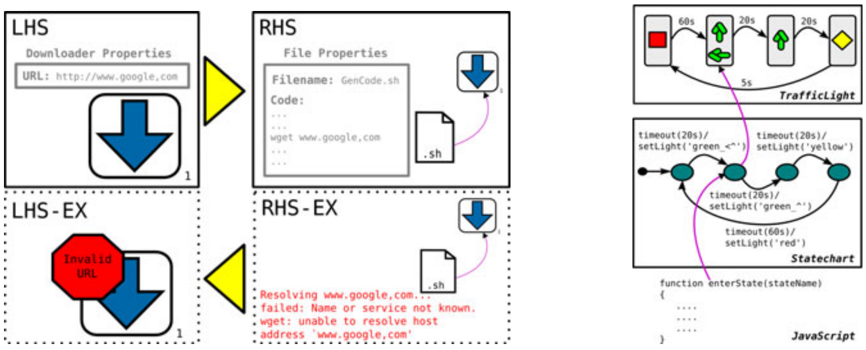


Fig. 2. (1)A DSm of a mobile conference registration application, generated models which isolate its (2)layout and (3)behavioural aspects, and (4) a trivial generated model of the final files on disk, all connected via traceability links

fully aware of the MTs that describe their models' semantics and generate artifacts. *Modellers* have an implicit understanding of their models' semantics but have little or no knowledge about how they are specified. As an example, consider Figure 2: a designer would be aware of each perspective, the links between them, and of the synthesized code and executable; a modeller would only be concerned with the DSm and synthesized executable. Consequently, the debugging scenarios for designers and modellers differ. Designer debugging focuses on ensuring the correctness of the MTs whereas modellers may assume that the MTs applied to their models are flawless and must instead establish the correctness of their models. In both scenarios, the basic work flow entails observing DSm versus artifact evolution[10] (with the designer possibly studying intermediate forms, if any). A formal discussion on implementing this exceeds the scope of this paper but the approach reviewed in Section 4.2 enables the sort of communication between DSm and artifact that would be required. Below, we re-visit the debugging concepts described earlier and translate them to DSm and artifact debugging.

Print Statements. Modern DSm editors offer means to display pertinent information as concrete syntax (e.g., depictions of Petri Net *place*s often include their number of *token*s) and provide easy access to construct parameters. Still, explicit output may be required. An elegant solution is the (semi-)automatic integration of output constructs in DSLs at design time. This is closely related to what we propose for MTLs and what is done in GPLs.

[10] DSm evolution here equates to program variables taking on values at runtime.



```
Resolving www.google.com...
failed: Name or service not known.
wget: unable to resolve host
address `www.google.com'
```

```
function enterState(stateName)
{
    ....
    ....
    ....
}
```

(a) (b)

Fig. 3. (a)A high-level construct, its mapping to code, and a possible rendering of exception translation specification facilities; (b)Stepping into in "designer debug mode" navigates across levels of abstraction

Assertions. Assertions can also be (semi-)automatically specified DSL constructs. Provisions to enable and disable them, and to halt model animation and artifact execution when they fail are needed. These might be rules or function calls and should be generated (and weaved into artifacts) automatically.

Exceptions. Although DSms may be animated, what is truly being executed are synthesized artifacts. Consequently, exceptions originate from artifacts and are described in terms of their metamodel. Some exceptions may describe irrelevant transient issues, others may describe issues relevant only to designers. Their handling is thus a design choice of the DSL architect. "Silent" exception handlers for irrelevant exceptions should be generated along with synthesized artifacts while relevant exceptions should be translated into domain-specific terms and propagated[11] to the modeller or designer (as depicted in Fig. 3a where a low-level error message results in updated concrete syntax at the DSm level).

Execution Control. The play, pause, stop and step commands also require two-way communication between DSm and artifact. Playing and stopping simply require means to remotely run or kill the generated model or program. The meaning of a step in an arbitrary DSm, however, is not obvious. A general definition is that *a step constitutes any modification to any parameter of any entity in a DSm*. Still, stepping can be considered from two orthogonal perspectives: the modeller's and the designer's. On one hand, the three step commands, in conjunction with our general definition, intuitively translate to DSLs which support hierarchy and composition. On the other, generating artifacts from DSms creates an implicit hierarchy between them. Thus, a designer, may prefer for the step into operation to take a step at the level of corresponding lower level entities. This scenario is depicted in Figure 3b where two successive step into

[11] The backward links described in Section 4.2 may be instrumental in this propagation.

operations lead a designer from a domain-specific traffic light model entity to a corresponding Statechart *state* and finally to a function in the generated code. Conversely, stepping out would bring the designer back to higher level entities and stepping over would perform a step given the current formalism (e.g., one code statement, one operational semantics rule). As for the pause command, a sensible approach is to pause the execution before running what would have been the next step at the DSm level for the modeller, and at the artifact level for the designer. These distinct stepping and pausing modes further motivate our previous proposal of separate debugging modes for designers and modellers. Finally, all of the above implies that (ideally automatic) instrumentation of artifacts to enable running only parts of them at a time are required[12].

Runtime Variable I/O. Model editing tools can be used to view and modify DSm and generated model variable values whereas code IDE debugger facilities can be used for synthesized coded artifacts. The challenge is to propagate changes such that consistency is preserved across all levels of abstraction[12].

Breakpoints. Breakpoints in state-based languages (e.g., Statecharts) could be set on states. For languages where state is implicit (e.g., Petri Nets whose state is their *marking*), however, they could be specified as patterns (pausing execution upon detection). Finally, designers should be able to specify breakpoints at any level of abstraction while modellers should be restricted to the DSm level.

Stack Traces. Stack traces remain tightly bound to the step into and out commands. Thus, they might display related actions at different levels of abstraction for designers while reflecting construct composition, if any, for modellers.

5 Conclusion and Future Work

We proposed a mapping between concepts in the software and DSM debugging realms. We distinguished between MT and DSm debugging, and between debugging scenarios for *designers*, who are fully aware of the MTs that describe their models' semantics and generate artifacts, and *modellers*, who have an implicit understanding of their models' semantics but little or no knowledge about how they are specified. Our work is meant as a guide for developing DSM debuggers: numerous MT debugger features can be built-in to MTLs and engines, while modular TGR-based artifact synthesis can considerably facilitate the implementation of most DSm and artifact debugger facilities. We plan to fully implement the concepts proposed in this work in AToM^3's successor, AToMPM.

References

1. Brown, A.W.: Model driven architecture: Principles and practice. Software and Systems Modeling (SoSym) 3, 314–327 (2004)
2. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal (IBMS) 45, 621–645 (2006)

[12] Once again, traceability links between DSms and artifacts may be key.

3. de Lara, J., Vangheluwe, H., Alfonseca, M.: Meta-modelling and graph grammars for multi-paradigm modelling in AToM^3. Software and Systems Modeling (SoSym) 3, 194–209 (2004)
4. Eisenstadt, M.: “My Hairiest Bug” war stories. Communications of the ACM (CACM) 40, 30–37 (1997)
5. Kelly, S., Tolvanen, J.-P.: Domain-Specific Modeling: Enabling Full Code Generation. Wiley-Interscience, Hoboken (2008)
6. Lengyel, L., Levendovszky, T., Mezei, G., Charaf, H.: Model transformation with a visual control flow language. International Journal of Computer Science (IJCS) 1, 45–53 (2006)
7. Mannadiar, R., Vangheluwe, H.: Modular synthesis of mobile device applications from domain-specific models. In: The 7th International Workshop on Model-Based Methodologies for Pervasive and Embedded Software, MOMPES (2010)
8. Mannadiar, R., Vangheluwe, H.: Modular synthesis of mobile device applications from domain-specific models. Technical Report SOCS-TR-2010.5, McGill University (2010)
9. Safa, L.: The making of user-interface designer a proprietary DSM tool. In: 7th OOPSLA Workshop on Domain-Specific Modeling (DSM), p. 14 (2007), `http://www.dsmforum.org/events/DSM07/papers.html`
10. Schürr, A.: Specification of graph translators with triple graph grammars. In: Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science (1995)
11. Syriani, E., Kienzle, J., Vangheluwe, H.: Exceptional transformations. In: Tratt, L., Gogolla, M. (eds.) ICMT 2010. LNCS, vol. 6142, pp. 199–214. Springer, Heidelberg (2010)
12. Syriani, E., Vangheluwe, H.: De-/re-constructing model transformation languages. In: 9th International Workshop on Graph Transformation and Visual Modeling Techniques, GT-VMT (2010)
13. Wu, H., Gray, J., Mernik, M.: Grammar-driven generation of domain-specific language debuggers. Software: Practice and Experience 38, 1073–1103 (2008)
14. Zeller, A.: Why Programs Fail: A Guide to Systematic Debugging, 2nd edn. Morgan Kaufmann, San Francisco (2009)

COPE – A Workbench for the Coupled Evolution of Metamodels and Models

Markus Herrmannsdoerfer

Institut für Informatik, Technische Universität München
Boltzmannstr. 3, 85748 Garching b. München, Germany
herrmama@in.tum.de

Abstract. Model-driven software development promises to increase productivity by offering modeling languages tailored to a problem domain. Consequently, an increasing number of modeling languages are built using metamodel-based language workbenches. In response to changing requirements and technologies, the modeling languages and thus their metamodels need to be adapted. Manual migration of existing models in response to metamodel adaptation is tedious and error-prone. In this paper, we present our tool COPE to automate the coupled evolution of metamodels and models. To not lose the intention behind the adaptation, COPE records the coupled evolution in an explicit history model. Based on this history model, COPE provides advanced tool support to inspect, refactor and recover the coupled evolution.

1 Introduction

Model-driven development promises to raise the abstraction level of today's software development with the help of the pervasive use of models. Models are built by using modeling languages that allow the users to directly express the abstractions from their problem domain. To reduce the effort for software development, implementation code can be automatically generated from these models. Recent approaches such as Model-Driven Architecture [11], Software Factories [5] and Domain-Specific Modeling [10] advocate to also develop modeling languages in a model-driven way—using metamodels to define the modeling language's syntax. Language workbenches—such as the Eclipse Modeling Project (EMP), Microsoft DSL Tools and MetaCase MetaEdit+—significantly reduce the effort to build tool support for modeling languages around the metamodels. In EMP, for instance, metamodels are built using the Ecore formalism provided by the Eclipse Modeling Framework (EMF). As a consequence, an increasing number of modeling languages is built both in industry and research. These languages range from general-purpose modeling languages like UML, over industry-wide languages like AUTOSAR, to languages used only inside an organization.

Problem. Like software, modeling languages are subject to evolution due to changing requirements and technological progress [3]. A modeling language is evolved by first adapting its metamodel to the new requirements. *Metamodel* *adaptation* requires the migration of tools for the modeling language such as editors and code generators. Most importantly, existing models may need to be migrated to conform to the adapted metamodel again. Avoiding *model migration* by downwards-compatible metamodel changes is often a poor solution, since it reduces the quality of the metamodel and thus the modeling language. By contrast, manual migration of models is tedious and error-prone, and therefore model migration needs to be automated. Building an automated model migration—even if highly desired in practice—is a non-trivial task, as it has to ensure the preservation of the meaning of a possibly unknown set of models. This task is further complicated by the issue that, in current practice, the intention behind the metamodel changes is lost in the evolution process.

Contribution. To not lose the intention behind the adaptation, we advocate to record the metamodel changes throughout the evolution process. In this paper, we present our tool COPE to record the metamodel adaptation together with the model migration—we call this the *coupled evolution* of metamodels and models. COPE records the evolution as a sequence of coupled operations in an explicit history model. Each coupled operation encapsulates both metamodel adaptation as well as reconciling model migration. Recurring coupled operations can be reused to further reduce the effort for building a model migration. Using the history model, existing models can be automatically migrated to the adapted version of the metamodel. Sometimes the recorded history does not perfectly specify the intended model migration—which is found out after testing the model migration—and thus needs to be modified. Sometimes the history cannot be recorded and hence needs to be recovered from the metamodel versions before and after the adaptation. To address these issues, COPE provides additional tool support to inspect, refactor and recover the coupled evolution.

2 Related Work

An approach to build a model migration for a metamodel adaptation needs to face two main challenges. First, it should lead to a correct migration which preserves the meaning of all models. Second, it should automate the building process to reduce the effort for model migration.

Manual specification approaches provide transformation languages to manually specify the model migration. Sprinkle presents a visual model transformation language to specify the transformation only for the difference between two metamodel versions [14]. Balasubramanian extends this language with usage patterns for typical migration scenarios [12]. The resulting Model Change Language (MCL) comes with the Generic Modeling Environment (GME). Flock is an EMF-based textual model transformation language that also automatically unsets model elements that are no longer conforming to the adapted metamodel [13]. While manual specification fosters correctness of the model migration, it also requires the most effort.

Matching approaches try to automatically derive a model migration from the matching between two metamodel versions. Cicchetti presents an EMF-based tool prototype that can not only derive a model migration for primitive changes, but also for compound changes [2]. Garcés presents an EMF-based matching language that allows the user to customize the matching process and to add new matching patterns [4]. While difference-based approaches completely automate the building process, they may not lead to a correct model migration. However, matching approaches can be complemented by manual specification to complete and correct the generated model migration.

Operation-based approaches record the coupled operations which are used to adapt the metamodel and which also encapsulate a model migration. Operation-based approaches provide a compromise for both challenges: They are likely to lead to a correct migration by means of recording, and at the same time reduce the effort by means of reusable coupled operations. However, they also require integration into the editor for the metamodel. Wachsmuth presents his EMF-based tool prototype ECORAL that provides a library of reusable coupled operations [15]. To increase expressiveness, we extended COPE with a means to specify a custom model migration for a recorded metamodel adaptation [8]. Compared to our previous work, in this paper, we introduce additional functions of COPE to inspect, refactor and recover the coupled evolution.

3 COPE – Coupled Evolution of Metamodels and Models

COPE is implemented based on the widely used Eclipse Modeling Framework (EMF)[1]. Besides recording the coupled evolution in an explicit history model, COPE provides support to inspect, refactor and recover the coupled evolution.

3.1 Recording the Coupled Evolution

As is depicted in Figure 1, COPE records the metamodel adaptation as a sequence of primitive changes in an explicit history model [6]. As is depicted in Figure 2, COPE's user interface—which is directly integrated into the existing EMF *metamodel editor*—provides access to the recorded *history model.* COPE supports two methods to form coupled operations [8], i.e. to attach a model migration to a sequence of primitive changes.

Enabling Reuse. Reuse of recurring migration specifications allows to significantly reduce the effort associated with building a model migration [7]. COPE thus provides *reusable coupled operations* which encapsulate metamodel adaptation and model migration in a metamodel-independent way. Reusable coupled operations are organized in a *library* which can be extended by declarations of new operations. The declaration is made independent of a specific metamodel through parameters, and may provide constraints to restrict the applicability of

[1] See EMF web site: `http://www.eclipse.org/modeling/emf`

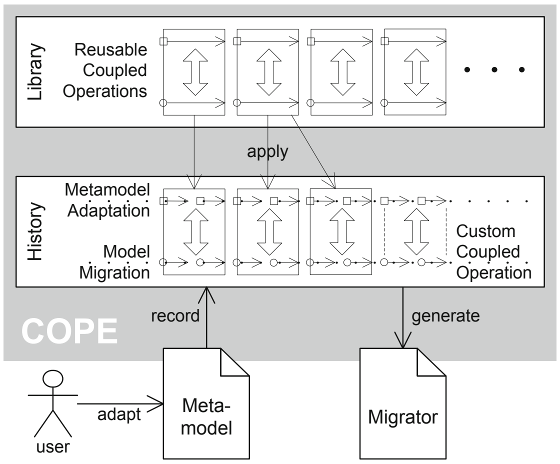
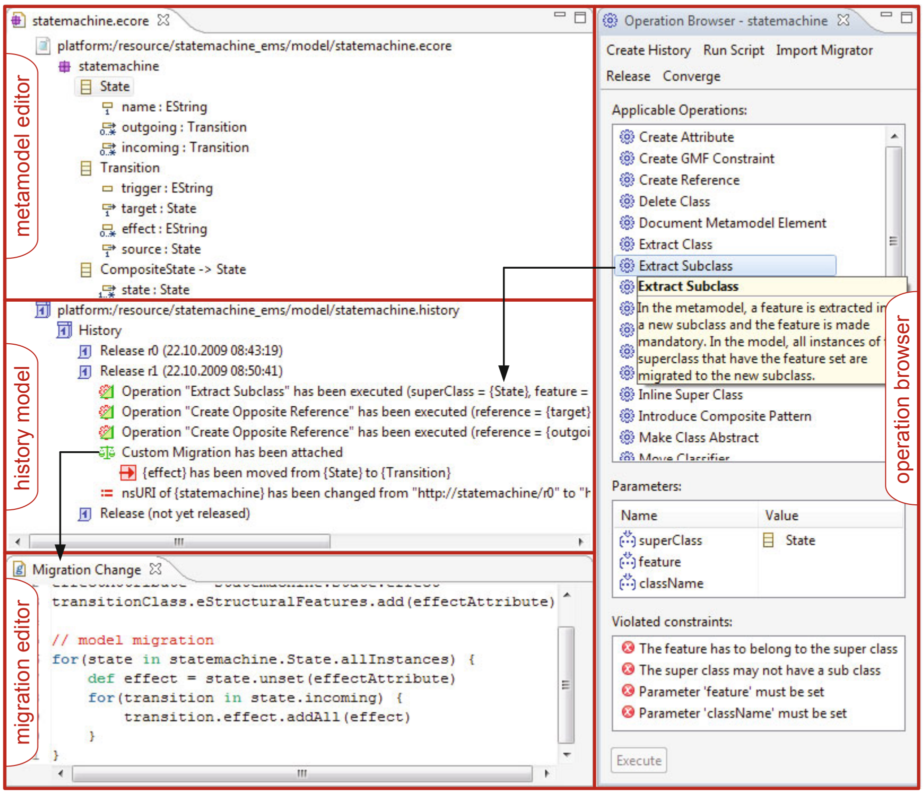


Fig. 1. Overview of COPE

the operation. Currently, COPE comes with about 60 reusable coupled operations. The user can adapt the metamodel by applying reusable coupled operations through the *operation browser*. The operation browser allows the user to set the parameters of a reusable coupled operation, and gives feedback on its applicability based on the constraints. When a reusable coupled operation is executed, its application is automatically recorded in the history model. Figure 2 shows the operation *Extract Subclass* being selected in the operation browser and recorded to the history model.

Supporting Expressiveness. Specifications of model migration can become so specific to a certain metamodel that reuse makes no sense [7]. To be able to cover these specifications, COPE allows the user to manually define a *custom coupled operation*. In order to do so, the user has to manually encode a model migration for a recorded metamodel adaptation. To encode a model migration, COPE provides a language expressive enough to cater for complex model migrations [8]. The user needs to perform a custom coupled operation only, in case no reusable coupled operation is available for the change at hand. First, the metamodel is directly adapted in the metamodel editor, in response to which the changes are automatically recorded in the history. A migration can later be attached to the sequence of metamodel changes. Figure 2 shows the *migration editor* to encode the custom migration for a metamodel adaptation.

Migrator Generation. A migrator can be generated from the history model that allows for the batch migration of existing models. The migrator packages the sequence of coupled operations which can be executed to automatically migrate existing models.



statemachine.ecore

platform:/resource/statemachine_ems/model/statemachine.ecore
- statemachine
  - State
    - name : EString
    - outgoing : Transition
    - incoming : Transition
  - Transition
    - trigger : EString
    - target : State
    - effect : EString
    - source : State
  - CompositeState -> State
    - state : State

platform:/resource/statemachine_ems/model/statemachine.history
- History
  - Release r0 (22.10.2009 08:43:19)
  - Release r1 (22.10.2009 08:50:41)
    - Operation "Extract Subclass" has been executed (superClass = {State}, feature =
    - Operation "Create Opposite Reference" has been executed (reference = {target}
    - Operation "Create Opposite Reference" has been executed (reference = {outgoi
    - Custom Migration has been attached
      - {effect} has been moved from {State} to {Transition}
    - nsURI of {statemachine} has been changed from "http://statemachine/r0" to "h
  - Release (not yet released)

Migration Change

```
transitionClass.eStructuralFeatures.add(effectAttribute)

// model migration
for(state in statemachine.State.allInstances) {
    def effect = state.unset(effectAttribute)
    for(transition in state.incoming) {
        transition.effect.addAll(effect)
    }
}
```

Operation Browser - statemachine

Create History Run Script Import Migrator
Release Converge

Applicable Operations:
- Create Attribute
- Create GMF Constraint
- Create Reference
- Delete Class
- Document Metamodel Element
- Extract Class
- Extract Subclass
- Inline Super Class
- Introduce Composite Pattern
- Make Class Abstract
- Move Classifier

Extract Subclass
In the metamodel, a feature is extracted in a new subclass and the feature is made mandatory. In the model, all instances of superclass that have the feature set are migrated to the new subclass.

Parameters:

Name	Value
superClass	State
feature	
className	

Violated constraints:
- The feature has to belong to the super class
- The super class may not have a sub class
- Parameter 'feature' must be set
- Parameter 'className' must be set

Execute

Fig. 2. User Interface of COPE

3.2 Inspecting the Coupled Evolution

The recorded history model allows the user to understand the intention behind the metamodel adaptation. Based on the history model, COPE provides the following functions to ease understanding the coupled evolution.

Identifying Breaking Changes. Breaking changes are changes which can possibly invalidate existing models [1]. To prevent errors during model migration, these changes need to have a model migration attached. COPE provides an analysis to identify breaking changes which do not yet have a model migration attached. Breaking changes are identified on the metamodel-level, i.e. independently of the existing models.

Metamodel Reconstruction. To understand the evolution, COPE allows the user to reconstruct metamodel versions from the history model. Earlier metamodel versions can be simply reconstructed by interpreting the primitive changes recorded in the history model. Thus it is not necessary to store all the intermediate metamodel versions which would require a large memory footprint. This reconstruction is interactive, allowing the user to browse through the history model.

When the user selects a change in the user interface, COPE reconstructs the snapshot of the metamodel right after the change. The metamodel snapshot is shown in a separate view through which it can be inspected.

History Differencing. Comparing two metamodel snapshots in the Eclipse Modeling Framework using EMF Compare[2] does not always yield an accurate difference model. This is due to the fact that—in the absence of universally unique identifiers—the matching between the metamodel elements from the two snapshots has to be inferred. To produce a more accurate difference model, the matching can be generated from the history model. In the user interface, COPE allows the user to select the source and target version for the comparison directly in the history model. COPE produces a view showing the difference model between the two metamodel versions.

3.3 Refactoring the Coupled Evolution

As the recorded history does not always perfectly specify the intended model migration, COPE provides functions to refactor the history model. The refactorings have to ensure the overall consistency of the history model: the history model has to reconstruct the current metamodel version.

Undoing Changes. When performing further changes, earlier changes might prove to be wrong. Manually performing the reverse changes is a possible solution, but might lead to a different intention when regarding the model migration. For example, deleting an attribute and creating it again would lead to the loss of the attribute's values in the model. Therefore, COPE provides support to undo changes after they have already been recorded. To facilitate undoing changes, the changes are stored in the history model both in forward and reverse direction. Then the changes to be undone need to be simply applied in the reverse direction. To preserve the consistency of the history model, changes can only be undone if no later changes depend on them.

Replacing Changes. For certain breaking changes, we might later identify a reusable coupled operation that provides the intended model migration. Rather than manually reimplementing the model migration, it is better to apply the reusable coupled operation instead. The primitive changes, however, have already been recorded to the history model. COPE thus provides support to replace a sequence of primitive changes with the application of a reusable coupled operation. COPE reconstructs the metamodel version before the changes and presents it to the user in a dialog where she can select and apply the appropriate reusable coupled operation. To keep the history model consistent, the primitive changes can only be replaced, if the operation application yields the same result.

Reordering Changes. Only consecutive changes can be replaced by a reusable coupled operation or can be enriched by a custom model migration. Sometimes the changes which we want to replace or enrich are not consecutive. Certain

[2] See EMF Compare web site: `http://wiki.eclipse.org/index.php/EMF_Compare`

changes, however, are independent of each other and thus can be reordered to make them consecutive. COPE therefore provides support to move changes to another position in the history model. To ensure the overall consistency of the history model, the following constraints need to be fulfilled: The changes can only be moved to an earlier position, if they do not depend on the changes that are jumped over, and the changes can only be moved to a later position, if the changes that are jumped over do not depend on them.

3.4 Recovering the Coupled Evolution

There are certain cases where COPE cannot be used during the adaptation of the metamodel. For example, the metamodel might be generated from another artifact, and therefore a different tool is used to edit the artifact. In these cases, the history model needs to be recovered from the metamodel versions before and after the adaptation.

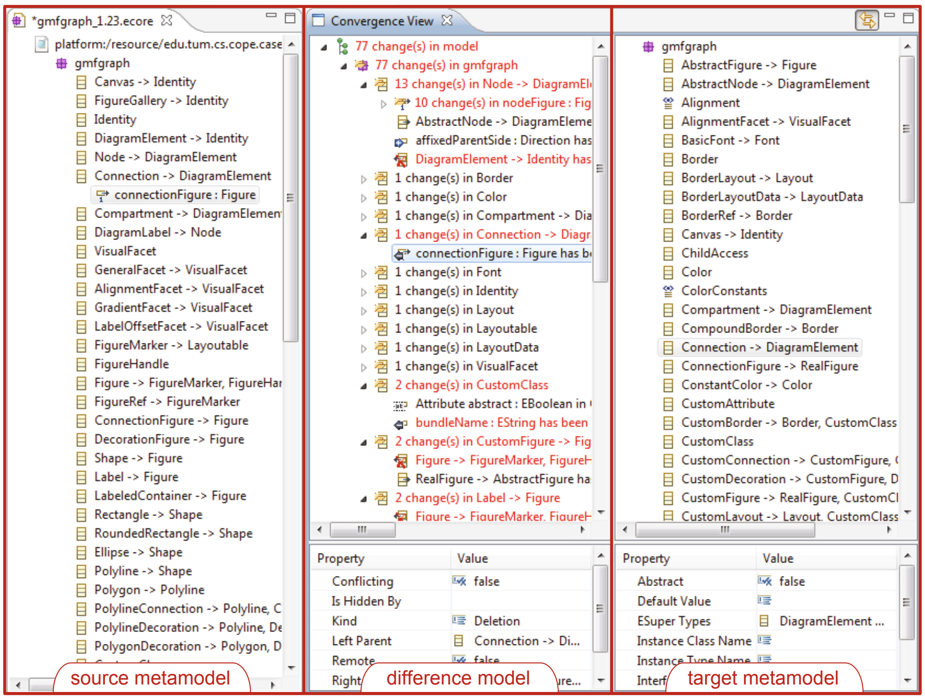
Metamodel Convergence. COPE provides advanced tool support to reverse engineer the history model. Figure 3 depicts the user interface to let a source metamodel version converge to a target metamodel version. The *source metamodel* version is loaded directly in the metamodel editor, whereas the *target metamodel* is displayed in a separate view. This view also displays the current *difference model* which results from the comparison between the source and target metamodel using EMF Compare. The differences are linked to the metamodel elements from both source and target version to which they apply. Breaking changes in the difference model—which necessitate a model migration—are highlighted in red. By means of the operation browser, the user can apply reusable coupled operations to bring the source metamodel nearer to the target metamodel. After an operation is executed on the source metamodel, the difference is automatically updated to reflect the changes. Non-breaking changes in the difference model can be easily applied to the source metamodel by double-clicking on them.

4 Evaluation

The presented functions are motivated from a number of case studies in which COPE has been applied. In these case studies, COPE has been applied to either reverse or forward engineer the coupled evolution.

Graphical Modeling Framework (GMF) is a widely used open source framework for the model-driven development of diagram editors[3]. To demonstrate the applicability of COPE, we have reverse engineered the coupled evolution of the four metamodels defined by GMF [9]. More specifically, we have applied the function for metamodel convergence to facilitate the reverse engineering of the coupled evolution. The resulting coupled evolution covers all the intermediate versions of the metamodels, meaning that COPE could have been directly applied for the maintenance of all four metamodels. The GMF case study indicates that,

[3] See GMF web site: `http://www.eclipse.org/modeling/gmf`



platform:/resource/edu.tum.cs.cope.case
gmfgraph
Canvas -> Identity
FigureGallery -> Identity
Identity
DiagramElement -> Identity
Node -> DiagramElement
Connection -> DiagramElement
connectionFigure : Figure
Compartment -> DiagramElemen
DiagramLabel -> Node
VisualFacet
GeneralFacet -> VisualFacet
AlignmentFacet -> VisualFacet
GradientFacet -> VisualFacet
LabelOffsetFacet -> VisualFacet
FigureMarker -> Layoutable
FigureHandle
Figure -> FigureMarker, FigureHar
FigureRef -> FigureMarker
ConnectionFigure -> Figure
DecorationFigure -> Figure
Shape -> Figure
Label -> Figure
LabeledContainer -> Figure
Rectangle -> Shape
RoundedRectangle -> Shape
Ellipse -> Shape
Polyline -> Shape
Polygon -> Polyline
PolylineConnection -> Polyline, C
PolylineDecoration -> Polyline, De
PolygonDecoration -> Polygon, D

77 change(s) in model
77 change(s) in gmfgraph
13 change(s) in Node -> DiagramEl
10 change(s) in nodeFigure : Fig
AbstractNode -> DiagramEleme
affixedParentSide : Direction has
DiagramElement -> Identity has
1 change(s) in Border
1 change(s) in Color
1 change(s) in Compartment -> Dia
1 change(s) in Connection -> Diagr
connectionFigure : Figure has b
1 change(s) in Font
1 change(s) in Identity
1 change(s) in Layout
1 change(s) in Layoutable
1 change(s) in LayoutData
1 change(s) in VisualFacet
2 change(s) in CustomClass
Attribute abstract : EBoolean in
bundleName : EString has been
2 change(s) in CustomFigure -> Fig
Figure -> FigureMarker, FigureH
RealFigure -> AbstractFigure ha
2 change(s) in Label -> Figure
Figure -> FigureMarker, FigureH

Property	Value
Conflicting	false
Is Hidden By	
Kind	Deletion
Left Parent	Connection -> Di...
Remote	false
Right	...re...

gmfgraph
AbstractFigure -> Figure
AbstractNode -> DiagramElement
Alignment
AlignmentFacet -> VisualFacet
BasicFont -> Font
Border
BorderLayout -> Layout
BorderLayoutData -> LayoutData
BorderRef -> Border
Canvas -> Identity
ChildAccess
Color
ColorConstants
Compartment -> DiagramElement
CompoundBorder -> Border
Connection -> DiagramElement
ConnectionFigure -> RealFigure
ConstantColor -> Color
CustomAttribute
CustomBorder -> Border, CustomClass
CustomClass
CustomConnection -> CustomFigure, (
CustomDecoration -> CustomFigure, D
CustomFigure -> RealFigure, CustomCl
CustomLayout -> Layout, CustomClass

Property	Value
Abstract	false
Default Value	
ESuper Types	DiagramElement ...
Instance Class Name	
Instance Type Name	
Interf	

Fig. 3. Metamodel Convergence

in practice, most of the coupled evolution can be captured by reusable coupled operations. Moreover, this case study helped to build an extensive library of reusable coupled operations.

Palladio Component Model (PCM) is a modeling language for the specification and analysis of component-based software architectures[4]. As a second application of the function for metamodel convergence, we have reverse engineered the coupled evolution of the metamodel defined by the Palladio Component Model (PCM) [8]. Similar to GMF, most of the coupled evolution could be covered by reusable coupled operations. In contrast to GMF, the resulting coupled evolution does not capture all the intermediate versions of the metamodel, as there were a lot of destructive changes that were reversed at a later instant. To be able to deal with these destructive changes in a clean manner, we have implemented the function for undoing changes.

Quamoco is a research project whose goal is to develop a language for modeling the product quality of software[5]. Currently, we are applying COPE for the evolutionary development of the metamodel on which the modeling language is based. The coupled evolution forward engineered by this case study is not

[4] See PCM web site: `http://www.palladio-approach.net`
[5] See QUAMOCO web site: `http://www.quamoco.de`

much different from the reverse engineered metamodel histories: most of the coupled evolution can still be covered by reusable coupled operations. Many of the reusable coupled operations identified in the previous case studies proved to be useful for the adaptation of the Quamoco metamodel. Furthermore, the functions to refactor the coupled evolution showed to be useful to introduce reusable coupled operations that have been newly identified.

UNICASE is a CASE tool that integrates models from the different development activities—amongst others, requirements engineering, design and project planning—into a unified model[6]. COPE is applied to forward engineer the coupled evolution of the metamodel to which these models conform. To support model evolution, UNICASE records the changes performed on the models and stores them in a repository. When the metamodel is adapted, we thus do not only have to migrate the models, but also the recorded changes. While COPE's migration language showed to be expressive enough to specify the change migration, we have extended COPE to be able to refine existing reusable coupled operations with the appropriate change migration.

5 Conclusion

Just as other software artifacts, modeling languages and thus their metamodels have to be adapted. To facilitate language evolution, adequate tool support is required to automate the migration of models. To obtain a correct model migration, COPE records the coupled evolution of metamodels and models in an explicit history model. As a result, the history model stores the sequence of coupled operations that have been performed during evolution. A coupled operation encapsulates the metamodel adaptation and reconciling model migration. Recurring coupled operations can be reused to further reduce the effort for building a model migration. To make the operation-based approach usable in practice, more advanced tool support is necessary to maintain the history model. COPE provides functions to inspect, refactor and recover the history model to better understand, correct and reverse engineer the coupled evolution.

Even if the presented functions have shown to be useful in a number of case studies, there is still room for further improvement. To facilitate reverse engineering the coupled evolution, we envision a function that automatically proposes reusable coupled operations based on the difference between two metamodel versions. To validate the resulting model migration, we plan to develop a framework for testing the coupled evolution.

COPE is open source and can be obtained from its website[7]. Moreover, COPE is about to be made available via the newly created Eclipse Project Edapt.

Acknowledgments. We are grateful to Daniel Ratiu for providing feedback on earlier versions of this paper. The presented work was partially funded by the German Federal Ministry of Education and Research (BMBF), grants "SPES2020, 01IS08045A" and "Quamoco, 01IS08023B".

[6] See UNICASE web site: `http://www.unicase.org`
[7] See COPE web site: `http://cope.in.tum.de`

References

1. Becker, S., Goldschmidt, T., Gruschko, B., Koziolek, H.: A process model and classification scheme for semi-automatic meta-model evolution. In: Proc. 1st Workshop MDD, SOA und IT-Management (MSI 2007), pp. 35–46. GiTO-Verlag (2007)
2. Cicchetti, A., Ruscio, D.D., Eramo, R., Pierantonio, A.: Automating co-evolution in model-driven engineering. In: 12th Intl. Enterprise Distributed Object Computing Conference (EDOC 2008). IEEE, Los Alamitos (2008)
3. Favre, J.M.: Languages evolve too! changing the software time scale. In: 8th Intl. Workshop on Principles of Software Evolution, pp. 33–42 (2005)
4. Garcés, K., Jouault, F., Cointe, P., Bézivin, J.: Managing model adaptation by precise detection of metamodel changes. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 34–49. Springer, Heidelberg (2009)
5. Greenfield, J., Short, K., Cook, S., Kent, S.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley, Chichester (2004)
6. Herrmannsdoerfer, M.: Operation-based versioning of metamodels with COPE. In: 2nd Intl. Workshop on Comparison and Versioning of Software Models (CVSM 2009), pp. 49–54. IEEE, Los Alamitos (2009)
7. Herrmannsdoerfer, M., Benz, S., Juergens, E.: Automatability of coupled evolution of metamodels and models in practice. In: Busch, C., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 645–659. Springer, Heidelberg (2008)
8. Herrmannsdoerfer, M., Benz, S., Juergens, E.: COPE - automating coupled evolution of metamodels and models. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 52–76. Springer, Heidelberg (2009)
9. Herrmannsdoerfer, M., Ratiu, D., Wachsmuth, G.: Language evolution in practice: The history of GMF. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 3–22. Springer, Heidelberg (2010)
10. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling. Wiley, Chichester (2007)
11. Kleppe, A.G., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley, Reading (2003)
12. Narayanan, A., Levendovszky, T., Balasubramanian, D., Karsai, G.: Automatic domain model migration to manage metamodel evolution. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 706–711. Springer, Heidelberg (2009)
13. Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Model migration with epsilon flock. In: Tratt, L., Gogolla, M. (eds.) ICMT 2010. LNCS, vol. 6142, pp. 184–198. Springer, Heidelberg (2010)
14. Sprinkle, J., Karsai, G.: A domain-specific visual language for domain model evolution. Journal of Visual Languages and Computing 15(3-4), 291–307 (2004)
15. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In: Bateni, M. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 600–624. Springer, Heidelberg (2007)

DSLTrans: A Turing Incomplete Transformation Language

Bruno Barroca[1], Levi Lúcio[2], Vasco Amaral[1], Roberto Félix[1], and Vasco Sousa[1]

[1] CITI, Departamento de Informática, Faculdade de Ciencias e Tecnologia Universidade Nova de Lisboa, Portugal
[2] LASSY, University of Luxembourg, Luxembourg
`Levi.Lucio@uni.lu`,
`{Bruno.Barroca,Vasco.Amaral,Roberto.Felix,Vasco.Sousa}@di.fct.unl.pt`

Abstract. In this paper we present DSLTrans: a visual language and a tool for model transformations[1]. We aim at tackling a couple of important challenges in model transformation languages — transformation termination and confluence. The contribution of this paper is the proposition of a transformation language where all possible transformations are guaranteed to be terminating and confluent by construction. The resulting transformation language is simple, turing incomplete and includes transformation abstractions to support transformations in a software language engineering context. Our explanation of DSLTrans includes a complete formal description of our visual language and its properties.

Keywords: Model Transformations, Turing Incompleteness, Termination, Confluence.

1 Introduction

A problem in modern model transformation languages that has recently received some attention is to how to guarantee that a transformation terminates. Because the semantics of transformation languages are usually based on graph grammars, the termination problem is in general undecidable [7]. Termination has been described in [6] as one of the '*quality requirements for a transformation language or tool*'. The problem has been approached by several authors [2,3,4] who have proposed criteria that can be applied to decide about the termination of transformations under particular conditions. The EMFTrans tool [1] presents a complete formalization of all concepts involved in a transformation and it is possible under certain conditions to decide if a given transformation is locally confluent and terminates.

[1] This work has been developed in the context of project BATIC3S partially funded by the Portuguese FCT/MCTES ref. PTDC/EIA/65798/2006, the doctoral grant ref. SFRH/BD/38123/2007, the post doctoral grant ref. SFRH/BPD/65394/2009 and the Luxembourguese FNR/CORE project MOVERE ref. C09/IS/02.

In this paper we propose an 'egg of Columbus' approach to the termination problem by building a visual transformation language called DSLTrans which guarantees that the number of steps in a model transformation is always finite. As a consequence, any transformation expressed in our language will always end. We also guarantee by construction the confluence of any model transformation written in DSLTrans, which is an important correctness property of model transformations as mentioned in [3]. DSLTrans is, by construction, a turing incomplete language. This is due to the fact that our language is free of loop or recursion constructs. The work presented in this paper provides the basis for the work we present in [5], where a technique for proving properties of the type '*if a structural relation between some elements of the source model holds, then another structural relation between some elements of the target model should also hold*' is presented. By proposing such a technique we are able to provide additional '*success criteria for a transformation language or tool*' [6], which is the ability to verify our model transformations.

The rest of this paper is organized as follows. In section 2, we informally describe the syntax and semantics of DSLTrans; In section 3, we describe the mathematical underpinnings of our transformation language; Finally, in section 4 we summarize and discuss future work.

2 Language Overview

Let us present the DSLTrans language with a simple example.

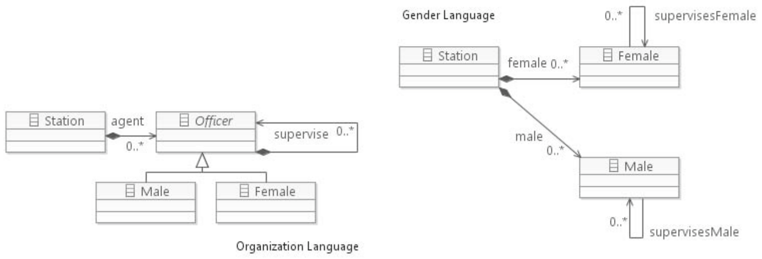


Fig. 1. Metamodels of a squad of agents(left) and a squad organized by gender(right)

Figure 1 presents two metamodels of languages for describing views over the organization of a police station. The metamodel annotated with 'Organization Language' represents a language for describing the chain of command in a police station, which includes male ($Male$ class) and female officers ($Female$ class). The metamodel annotated with 'Gender Language' represents a language for describing a different view over the chain of command, where the officers working at the police station are classified by gender. In figure 2 we present a transformation written in DSLTrans between models of both languages. The purpose of this

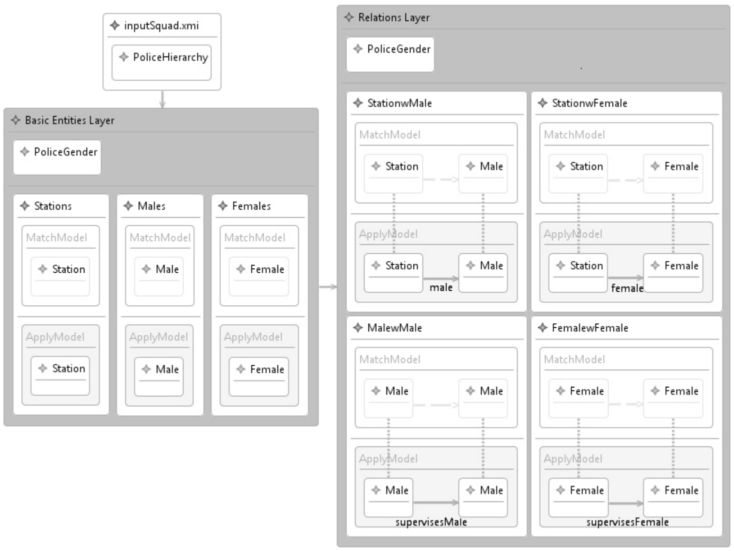


Fig. 2. A model transformation expressed in DSLTrans using a concrete visual syntax of an Eclipse diagrammatic editor [8]

transformation is to flatten a chain of command given in language 'Organization Language' into two independent sets of male and female officers. Within each of those sets the command relations are kept, i.e. a female officer will be directly related to all her female subordinates and likewise for male officers.

An example of an instance of this transformation can be observed in figure 3, where the original model is on the left and the transformed one on the right. Notice that the elements s, m_k and f_k in the figure on the left are instances of the source metamodel elements $Station$, $Male$ and $Female$ respectively (in figure 1). The primed elements in the figure on the right are their instance counterparts in the target metamodel.

A transformation in DSLTrans is formed by a set of input model sources called *file-ports* ('inputSquad.xmi' in figure 2) and a list of *layers* ('Basic entities' and 'Relations' layers in figure 2). Both layers and file-ports are typed according to metamodels. DSLTrans executes sequentially the list of layers of a transformation specification. A layer is a set of transformation rules, which executes in a non-deterministic fashion. Each transformation rule is a pair $(match, apply)$ where *match* is a pattern holding elements from the source metamodel, and *apply* is a pattern holding elements of the target metamodel. For example, in the transformation rule 'Stations' in the 'Basic entities' layer (in figure 2) the *match* pattern holds one 'Station' class from the 'Squad Organization Language' metamodel — the source metamodel; the *apply* pattern holds one 'Station' class from the 'Squad Gender Language' metamodel — the target metamodel. This means that all elements in the input source which are of type 'Station' of the

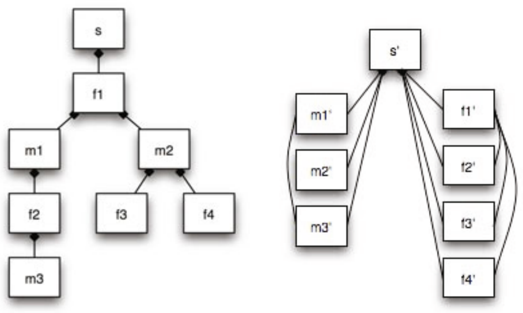


Fig. 3. Original model (left) and transformed model (right)

source metamodel will be transformed into elements of type 'Station' of the target metamodel.

Let us first define the constructs available for building transformation rules' match patterns. We will illustrate the constructs by referring to the transformation in figure 2.

- *Match Elements*: are variables typed by elements of the source metamodel which can assume as values elements of that type (or subtype) in the input model. In our example, a match element is the 'Station' element in the 'Stations' transformation rule of layer 'Basic Entities' layer;
- *Attribute Conditions*: conditions over the attributes of a *match* element;
- *Direct Match Links*: are variables typed by labelled relations of the source metamodel. These variables can assume as values relations having the same label in the input model. A direct match link is always expressed between two match elements;
- *Indirect Match Links*: indirect match links are similar to direct match links, but there may exist a path of containment associations between the matched instances. In our implementation, the notion of indirect links captures only EMF containment associations. In our example, indirect match links are represented in all the transformation rules of layer 'Relations' as dashed arrows between elements of the match models;
- *Backward Links*: backward links connect elements of the match and the apply models. They exist in our example in all transformation rules in the 'Relations' layer, depicted as dashed vertical lines. Backward links are used to refer to elements created in a previous layer in order to use them in the current one. An important characteristic of DSLTrans is that throughout all the layers the source model remains intact as a match source. Therefore, the only possibility to reuse elements created from a previous layer is to reference them using backward links;
- *Negative Conditions*: it is possible to express negative conditions over match elements, backward, direct and indirect match links.

The constructs for building transformation rules' apply patterns are:

- *Apply Elements and Apply Links*: apply elements, as match elements, are variables typed by elements of the source metamodel. Apply elements in a given transformation rule that are not connected to backward links will create elements of the same type in the transformation output. A similar mechanism is used for apply links. These output elements and links will be created as many times as the match model of the transformation rule is instantiated in the input model. In our example, the 'StationwMale' transformation rule of layer 'Relations Layer' takes instances of *Station* and *Male* (of the 'Gender Language' metamodel) which were created in a previous layer from instances of *Station* and *Male* (of the 'Organization Language' metamodel), and connects them using a 'male' relation;
- *Apply Attributes*: DSLTrans includes a small attribute language allowing the composition of attributes of apply model elements from references to one or more match model element attributes.

3 Formal Syntax and Semantics

In this section, we build a formal definition of DSLTrans in order to provide a clear specification of our language and a basis for studying and proving properties about it. In the mathematical theory we disregard the formalization of: class attributes; negative conditions; class inheritance at the metamodel level. We present a light formalization of the relations at the metamodel and model levels which deals only with the difference between reference and containment relations between classes. These non formalized — but implemented in [8] — features of the language do not affect the termination or confluence properties of our language.

3.1 Transformation Language Syntax

Definition 1. *Typed Graph and Indirect Typed Graph*
A typed graph is a triple $\langle V, E, \tau \rangle$ *where* V *is a finite set of vertices,* $E \subseteq V \times V$ *is a finite set of directed edges connecting the vertices and* $\tau : \{V \cup E\} \rightarrow Type \cup \{containment, reference\}$ *is a typing function for the elements of* V *and* E *such that* $\tau(v) \in Type$ *if* $v \in V$ *and* $\tau(e) \in \{containment, reference\}$ *if* $e \in E$. *Edges* $(v, v') \in E$ *are noted* $v \rightarrow v'$. *We furthermore impose that the graph* $\langle V, \{v \rightarrow v' \in E | \tau(v \rightarrow v') = containment\} \rangle$ *is acyclic*[2]. *The set of all typed graphs is called* TG.

An indirect typed graph is a 4-tuple $\langle V, E, T, Il \rangle$, *where* $\langle V, E, T \rangle$ *is a typed graph and* $Il \subseteq E$ *is a set of edges called* indirect links. *The set of all indirect typed graphs is called* ITG.

[2] By using *containment* and *reference* as types for edges we allow modeling the different types of associations between the elements of a metamodel or a model. In our implementation, the EMF containment associations implements the acyclic subgraph of containment relations in a typed graph.

Definition 2. *Typed Graph Union*
Let $\langle V, E, \tau\rangle, \langle V', E'\tau'\rangle \in TG$ be typed graphs. The typed graph union is the function $\sqcup : TG \times TG \to TG$ defined as:

$$\langle V, E, \tau\rangle \sqcup \langle V', E', \tau'\rangle = \langle V \cup V', E \cup E', \tau \cup \tau'\rangle$$

Definition 3. *Typed Subgraph and Indirect Typed Subgraph*
Let $\langle V, E, \tau\rangle = g, \langle V', E', \tau'\rangle = g' \in TG$ be typed graphs. g' is a typed subgraph of g, written $g' \blacktriangleleft g$, iff $V' \subseteq V$, $E' \subseteq E$ and $\tau' = \tau|_{V'}$.
An indirect typed graph $\langle V', E', \tau', Il\rangle \in ITG$ is an indirect typed subgraph of a typed graph $\langle V, E, \tau\rangle \in TG$, written $\langle V', E', \tau', Il\rangle \lhd \langle V, E, \tau\rangle$ iff:

1. *$\langle V', E' \setminus Il, \tau'\rangle \blacktriangleleft \langle V, E, \tau\rangle$*
2. *if $v_i \to v_i' \in Il$ then there exists $v \to v' \in E_c^*$ where $\tau(v_i) = \tau(v)$, $\tau(v_i') = \tau(v')$ and E_c^* is obtained by the transitive closure of $E_c = \{v \to v' \in E | \tau(v \to v') = containment\}$.*

Definition 4. *Typed Graph Equivalence*
Let $\langle V, E, \tau\rangle = g, \langle V', E', \tau'\rangle = g' \in TG$ be typed graphs. g and g' are equivalent, written $g \cong g'$, iff there is a graph isomorphism $f : V \to V'$ of graphs $\langle V, E\rangle$ and $\langle V', E'\rangle$ such that $\forall x \in V \cup E \,.\, \tau(x) = \tau'(f(x))$ and $\forall x' \in V' \cup E' \,.\, \tau'(x') = \tau(f^{-1}(x'))$

More informally, two typed graphs are defined equivalent if they have the same shape and related vertices and edges have the same type.

Definition 5. *Typed Graph Instance*
Let $\langle V, E, \tau\rangle = g, \langle V', E', \tau\rangle = g' \in TG$ be typed graphs. g' is a typed graph instance of g, written $g' \Vdash g$, iff for all $v_1' \to v_2' \in E'$ there is a $v_1 \to v_2 \in E$ such that $\tau(v_1') = \tau(v_1)$, $\tau(v_2') = \tau(v_2)$ and $\tau(v_1' \to v_2') = \tau(v_1 \to v_2)$.

Notice that we only enforce that connections between vertices of g' must exist also in g and have the same type.

Definition 6. *Metamodel and Model*
A metamodel $\langle V, E, \tau\rangle \in TG$ is a typed graph where τ is a bijective typing function. The set of all metamodels is called $META$.

A model is a 4-tuple $\langle V, E, \tau, M\rangle$ where $\langle V, E, \tau\rangle$ is a typed graph. Moreover $M = \langle V', E', \tau'\rangle \in META$ is a Metamodel and the codomain of τ equals the codomain of τ'. Finally $\langle V, E, \tau\rangle \Vdash M$, which means $\langle V, E, \tau\rangle$ is an instance of a metamodel M. The set of all models for a metamodel M is called $MODEL^M$.

Definition 7. *Match-Apply Model*
A Match-Apply Model is a 6-tuple $\langle V, E, \tau, Match, Apply, Bl\rangle$, where $Match = \langle V', E', \tau', s\rangle$ and $Apply = \langle V'', E'', \tau'', t\rangle$ are models and $\langle V, E, \tau\rangle = \langle V', E', \tau'\rangle \sqcup \langle V'', E'', \tau''\rangle$. Edges $Bl \subseteq V' \times V'' \subseteq E$ are called backward links. *s is called*

the source *metamodel and t the target metamodel. The set of all Match-Apply models for a source metamodel s and a target metamodel t is called MAM^s_t. Vertices in the Apply model which are not connected to* backward links *are called* free vertices. *The back* $: MAM^s_t \rightarrow MAM^s_t$ *function connects all vertices in the Match model to all* free *vertices with* backward link *edges.*

The $Match$ part of a match-apply model is used to hold the immutable source model during a transformation. The *Apply* part is used to hold the intermediate results of the transformation.

Definition 8. *Transformation Rule*
A Transformation Rule is a 7-tuple $\langle V, E, \tau, Match, Apply, Bl, Il\rangle$*, where* $\langle V, E, \tau, Match, Apply, Bl\rangle \in MAM^s_t$ *is a match-apply model.* $Match = \langle V, E, \tau, M\rangle$ *and the edges* $Il \subseteq E$ *are called* indirect links *(see definition 3). The set of all transformation rules is called* TR^s_t*. The strip* $: TR^s_t \rightarrow TR^s_t$ *function removes from a transformation rule all free vertices and associated edges.*

We define a *transformation rule* as a kind of match-apply model which allows *indirect links* in the *match* pattern.

Definition 9. *Layer, Transformation*
A layer is a finite set of transformation rules $tr \subseteq TR^s_t$*. The set of all layers for a source metamodel s and a target metamodel t is called* $Layer^s_t$*. A transformation is a finite list of layers denoted* $[l_1 :: l_2 :: \ldots :: l_n]$ *where* $l_k \in Layer^s_t$ *and* $1 \leq k \leq n$*. The set of all transformations for a source metamodel s and a target metamodel t is called* $Transformation^s_t$*.*

We naturally extend the notion of union (definition 2) to models (definition 6), match-apply models (definition 7) and transformation rules (definition 8). We also extend the notion of indirect typed subgraph (definition 3) to transformation rules (definition 8) and match-apply models (definition 7). Finally, we extend the notion of typed graph equivalence (definition 4) to transformation rules (definition 8).

3.2 Transformation Language Semantics

Definition 10. *Match Function*
Let $m \in MAM^s_t$ *be a model and* $tr \in TR^s_t$ *be a transformation rule. The match* $: MAM^s_t \times TR^s_t \rightarrow \mathcal{P}(TR^s_t)$ *is defined as follows:*

$$match_{tr}(m) = remove\big(\{g \mid g \lhd m \wedge g \cong strip(tr)\}\big)$$

Due to the fact that the $\cong$ *relation is based on the notion of graph isomorphism, permutations of the same match result may exist in the* $\{g \mid g \lhd m \wedge g \cong strip(tr)\}$ *set. The — undefined — remove* $: \mathcal{P}(TR^s_t) \rightarrow \mathcal{P}(TR^s_t)$ *function is such that it removes such undesired permutations.*

Definition 11. *Apply Function*
Let $m \in MAM_t^s$ be a match-apply model and $tr \in TR_t^s$ a transformation. The $apply : MAM_t^s \times TR_t^s \to MAM_t^s$ is defined as follows:

et $m \in MAM_t^s$ be a match-apply model and $tr \in TR_t^s$ a transformation. The $apply : MAM_t^s \times TR_t^s \to MAM_t^s$ is defined as follows:

$$apply_{tr}(m) = \bigsqcup_{g \in match_{tr}(m)} back(g \sqcup g_{\triangle})$$

$$\textit{where } g_{\triangle} \textit{ is such that } g \sqcup g_{\triangle} \cong tr$$

The freshly created vertices of $g_{\triangle}$ in the flattened $apply_{tr}(m)$ set are disjoint.

Definitions 10 and 11 are complementary: the former gathers all subgraphs of a match-apply graph which match a transformation rule; the latter builds the new instances which are created by applying that transformation rule as many times as the number of subgraphs found by the *match* function. The *strip* function is used to enable matching over backward links but not elements to be created by the transformation rule. The *back* function connects all newly created vertices to the elements of the source model that originated them.

Definition 12. *Layer Step Semantics*
Let $l \in Layer$ be a Layer. The layer step relation $\overset{layerstep}{\rightarrow} \subseteq MAM_t^s \times TR_t^s \times MAM_t^s$ *is defined as follows:*

$$\frac{}{\langle m, m', \emptyset \rangle \xrightarrow{layerstep} m \sqcup m'} \qquad \frac{tr \in l,\ apply_{tr}(m) = m''',\quad \langle m, m'' \sqcup m''', l \backslash \{tr\} \rangle \xrightarrow{layerstep} m'}{\langle m, m'', l \rangle \xrightarrow{layerstep} m'}$$

where $\{m, m', m''\} \subseteq MAM_t^s$ are match-apply models.

The freshly created vertices in m''' are disjoint from those in m''.

For each layer we go through all the transformation rules and build for each one of them the set of new instances created by their application. These instances are built using the *apply* function in the second rule of definition 12. The new instance results of the *apply* function for each transformation rule are accumulated until all transformation rules are treated. Then, the first rule of definition 12 will merge all the new instances with the starting match-apply model. The merge is performed by uniting (using the non-disjoint $\sqcup$ union) match-apply graphs including the new instances with the starting match-apply model.

Definition 13. *Transformation Step Semantics*
Let $[l :: R] \in Transformation_t^s$ be a Transformation, where $l \in Layer_t^s$ is a Layer and R a list. The transformation step relation $\overset{trstep}{\rightarrow} \subseteq MAM_t^s \times TR_t^s \times MAM_t^s$ *is defined as follows:*

$$\frac{}{\langle m,[]\rangle \xrightarrow{trstep} m} \qquad \frac{\left\langle m,\langle\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset\rangle,l\right\rangle \xrightarrow{layerstep} m'',\ \langle m'',R\rangle \xrightarrow{trstep} m'}{\langle m,[l::R]\rangle \xrightarrow{trstep} m'}$$

where $\{m, m', m''\} \subseteq MAM_t^s$ are match-apply models.

A model transformation is a sequential application of transformation layers to a match-apply model containing the source model and an empty apply model. The transformation output is the apply part of the resulting match-apply model.

Definition 14. *Model Transformation*
Let $m_s \in MODEL^s$ and $m_t \in MODEL^t$ be models and $tr \in Transformation_t^s$ be a transformation. A model transformation $\overset{transf}{\rightarrow} \subseteq MODEL^s \times Transformation_t^s \times MODEL^t$ is defined as follows:

$$m_s, tr \xrightarrow{transf} m_t \Leftrightarrow \langle V, E, \tau, m_s, \emptyset, \emptyset\rangle, tr \xrightarrow{trstep} \langle V, E, \tau, m_s, m_t, Bl\rangle$$

We now prove two important properties about DSLTrans' transformations.

Proposition 1. *Confluence*
Every model transformation is confluent regarding typed graph equivalence.

Proof. (Sketch) We want to prove that for every model transformation $tr \in Transformation_t^s$ having as input a model $m_s \in MODEL^s$, if $m_s, tr \xrightarrow{transf} m_t$ and $m_s, tr \xrightarrow{transf} m_t'$ then $m_t \cong m_t'$. Note that we only have to prove typed graph equivalence between m_t and m_t' because the identifiers of the objects produced by a model transformation are irrelevant.
If we assume $\neg(m_t \cong m_t')$ then this should happen because of non-determinism points in the rules defining the semantics of a transformation: 1) in definition 11 g_Δ is non-deterministic up to typed graph equivalence, which does not contradict the proposition; 2) in definition 12 transformation rule tr is chosen non-deterministically from layer l. Thus, the order in which the transformation rules are treated is non-deterministic. However, the increments to the transformation by each rule of a layer are united using $\sqcup$, which is commutative and thus renders the transformation result of each layer deterministic. Since there are no other possibilities of non-determinism points in the semantics of a transformation, $\neg(m_t \cong m_t')$ provokes a contradiction and thus the proposition is proved. $\square$

Proposition 2. *Termination*
Every model transformation terminates.

Proof. (Sketch) Let us assume that there is a transformation which does not terminate. In order for this to happen there must exist a section of the semantics of that transformation which induces an algorithm with an infinite amount of steps. We identify three points of a transformation's semantics where this can happen: 1) if definition 13 induces an infinite amount of steps. The only possibility for this to happen is if the transformation has an infinite amount of layers, which is a contradiction with definition 9; 2) if definition 12 induces an infinite amount of steps. The only possibility for this to happen is if a layer has an infinite amount of transformation rules, which is a contradiction with definition 8; 3) if the result of the $match_{tr}(m)$ function in definition 10 is an infinite set of match-apply graphs. The match-apply graph m is by definition finite, thus the number of isomorphic subgraphs of m is infinite only if the transitive closure of containment edges of m is infinite. The only possibility for this to happen is if the graph induced by the containment edges of m has cycles, which contradicts definition 1. Since there are no more points in the semantics of a transformation that can induce an infinite amount of steps, the proposition is proved. □

4 Conclusions and Future Work

We have presented DSLTrans, a turing incomplete transformation language with a mathematical underpinning which guarantees transformation termination and confluence by construction. With this language, we have introduced interesting abstractions such as layers, backward and indirect links. An important side effect of DSLTrans not being a turing complete language is the fact that verification of properties about our transformations are possible.
Our efforts are now focused on the optimization of the efficiency of the transformation engine itself, and we will perform a thorough validation on the usability aspects of this language in several transformation use cases.

References

1. Biermann, E., Ermel, C., Taentzer, G.: Precise Semantics of EMF Model Transformations by Graph Transformation. In: Busch, C., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 53–67. Springer, Heidelberg (2008)
2. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Heidelberg (2006)
3. Ehrig, H., Ehrig, K., de Lara, J., Taentzer, G., Varró, D., Varró-Gyapay, S.: Termination criteria for model transformation. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 49–63. Springer, Heidelberg (2005)
4. Levendovszky, T., Prange, U., Ehrig, H.: Termination criteria for dpo transformations with injective matches. Electronic Notes in Theoretical Computer Science 175(4), 87–100 (2007)
5. Lúcio, L., Barroca, B., Amaral, V.: A technique for automatic validation of model transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6394, pp. 136–150. Springer, Heidelberg (2010)
6. Mens, T., Van Gorp, P.: A taxonomy of model transformation. Electronic Notes in Theoretic Computer Science 152, 125–142 (2006)
7. Plumpf, D.: Termination of graph rewriting is undecidable. Fundamenta Informaticae 33(2), 201–209 (1998)
8. Solar Group. Dsltrans plug-in (2010), `http://solar.di.fct.unl.pt/twiki/pub/BATICCCS/ReleaseFiles/dsltrans.october,zip`

Translator Generation Using ART

Adrian Johnstone and Elizabeth Scott

Royal Holloway, University of London, Egham, Surrey, TW20 0EX, UK
{a.johnstone,e.scott}@rhul.ac.uk

Abstract. ART (Ambiguity Resolved Translators) is a new translator generator tool which provides fast generalised parsing based on an extended GLL algorithm and automatic generation of tree traversers for manipulating abstract syntax. The input grammars to ART comprise modular sets of context free grammar rules, enhanced with regular expressions and annotations that describe disambiguation and tree modification operations using the TIF (Tear-Insert-Fold) formalism. ART generates a GLL parser for the input grammar along with an *output grammar* whose derivation trees are the abstract trees specified by the TIF tree modification operations.

1 Introduction

ART is an integrated generalised parser generator and tree rewriter. ART supports the traditional applications for parser generators (such as compiler front end generation) by providing high performance parsing coupled with parser generation times that are typically less than the time needed to compile the generated code. ART is the first full-scale implementation of both the GLL algorithm [2] and the TIF formalism [3].

Existing generalised parser generators typically use a variant of bottom-up GLR parsing; for example, in ASF+SDF [6], Stratego [1] and Elkhound[4]; even Bison has a partial GLR mode. We have described elsewhere improvements to the GLR algorithm that provide worst-case cubic performance using binarised SPPF's but recognise that users find shift-reduce automata hard to understand. GLL is a generalisation of recursive descent that also provides worst-case cubic performance with linear performance on LL(1) parts of a grammar: in practice we find that GLL runs approximately as fast as our BRNGLR parsers.

An ART-generated parser or treewalker, Π, performs the following steps (i) lexical analysis of string characters or tree labels into *tokens*; (ii) parsing of those tokens against the grammar to form a *Shared Packed Parse Forest* (SPPF) of derivations; (iii) disambiguation of the potentially many derivations into a single derivation tree; (iv) restructuring of that derivation tree using the TIF operators to form a *restructured derivation tree* (RDT); (v) semantics evaluation.

This paper is mostly concerned with step (iv). The GLL algorithm used in step (ii) was presented at LDTA09 [2] and we shall discuss disambiguation in a future presentation.

On each run, ART produces a parser or treewalker for its input grammar, along with a *TIF Transformed Grammar* (TTG) which completely describes the family of trees that can be constructed by that parser or tree walker. A TTG is itself a TIF grammar, and can thus be used as input to ART, usually after TIF operators specifying the next phase of tree reworking have been added.

The idea is that parsing and tree transformation can be broken down into a series of distinct phases, allowing separation of concerns, but using a single notation and conceptually identical processing at each stage. Our goal is to have a system which is theoretically well founded, but which software engineers and casual users find approachable. The TIF operators are based on the notions of tearing and folding trees using three simple local transforms, coupled with the ability to insert arbitrary tree fragments.

Consider the construction of a C++ to ANSI-C converter such as CFront or of an intermediate language converter such as CIL [5] which simplifies and normalises C++ into a core sub-language in which, for instance, all loops are represented by a single form. We envisage writing such converters as a number of distinct phases, for instance one to handle loop constructs and another to rework data scoping. This separation of concerns between phases allows better testing and in principle reduces the opportunities for unexpected interactions between tree restructurings. We illustrate the process here as a chain of parser/tree walkers $\Pi_0, \Pi_1, \ldots$ with associated TTG's $\Gamma_1, \Gamma_2, \ldots$. At each stage, we envisage the software engineer intervening by adding TIF annotations and by factoring, deleting or multiplying out rules to form a Γ_i'. This process is marked by dashed lines.

$$\begin{tikzcd}
\boxed{\Gamma_0} \arrow[r, dashed] & \boxed{\Gamma_0'} \arrow[r] & \boxed{\text{ART}} \arrow[r] \arrow[d] & \boxed{\Gamma_1} \arrow[r, dashed] & \boxed{\Gamma_1'} \arrow[r] & \boxed{\text{ART}} \arrow[r] \arrow[d] & \boxed{\Gamma_2} \arrow[r, dashed] & \boxed{\Gamma_2'} \arrow[r] & \boxed{\text{ART}} \arrow[r] \arrow[d] & {} \\
\boxed{\sigma} \arrow[rr] & & \boxed{\Pi_0} \arrow[rr] & & \boxed{\tau_1} \arrow[r] & \boxed{\Pi_1} \arrow[rr] & & \boxed{\tau_2} \arrow[r] & \boxed{\Pi_2} \arrow[r] & {}
\end{tikzcd}$$

The initial inputs are a concrete grammar Γ_0 (say, the ANSI-C standard grammar) and a string σ to be parsed. The translator designer adds TIF annotations to specify the first intermediate form and semantic actions to be executed over trees from that intermediate form. ART then generates a GLL parser/TIF RDT builder Π_0 which processes σ to produce an RDT τ_1. ART also produces the TTG Γ_1 which specifies the full range of τ intermediate forms that can arise from Π_0. Further annotation and reworking of Γ_1 to Γ_1' provides the input to the next stage.

There are significant open questions concerning change management with this approach. As things stand, if a change is made to the grammar is an early phase, the contingent changes to TIF annotations will have to be manually propagated throughout the chain. Ideally we would have some meta-TIF notation that could specify the TIF annotations and thus reapply them automatically. At present we are attempting to gain experience with the 'manual' approach before trying to abstract a meta-notation.

Part of the motivation for ART comes from our experience of manually constructing simplified equivalent grammars for complex languages such as the C-like assembler for Analog Device's digital signal processor line. Our `asm21toc` reverse compiler started off with a detailed context free grammar (CFG) that captured in the syntax many constraints that arise from the hardware of those processors: for instance shifter operations and arithmetic operations use different sets of input and output registers. We then moved to a grammar which formally accepted a larger language in which, amongst other things, any register could supply the operands for any operator. This *widened* grammar was safe to use because any real program had already been parsed for correctness by the tight grammar: the fact that the grammar could accept programs that were strictly illegal was of no consequence because they had already been filtered out. The widened grammar concentrated operand fetch semantics into one rule, rather than the complicated case analysis that we would have had to implement with the 'real' grammar.

An important feature of our approach in ART is that the tool provides not only a parser, but a formal description of the parser's output which is guaranteed to be complete. A common failure mode when writing systems based on tree rearrangement is to forget about some obscure or infrequently used special case. For instance, a tree-walker based code generator must be able to completely tile any possible intermediate tree with target instructions, and proving that every possible case has been covered is hard if we use *ad hoc* mappings. An engineer working with ART is presented with a grammar which completely describes all possible output trees at each stage of a compiler: by ensuring that every production has appropriate rewrites or semantics, complete coverage is assured.

2 Source Syntax, Modularity and Parsing

The ART source syntax mostly follows the conventions of our earlier RDP and GTB toolkits with added support for modularisation and lexical level rules. We have also created the tools `gramex` and `gramconv` which extract grammar rules from electronically readable standards documents and convert them to the source syntax for a variety of tools including Bison, ASF+SDF, GTB and ART; optionally converting EBNF to BNF using a variety of idioms. These tools and extracted grammars for multiple standardisations of Java, C, C++ and Pascal. are available from

`http://www.cs.rhul.ac.uk/research/languages/projects/grammars/index.html`

An ART specification comprises one or more modules with associated import and export lists. In software engineering, modularisation is used to allow separation of concerns, to encapsulate and abstract away from details, and to support reuse; we believe that the engineering of large grammar-based systems also benefits from effective modularisation. A complete example of an ART specification for a tiny language is shown on page 313.

ART grammar rules use conventional CFG syntax augmented with the postfix regular operators `*`, `+` and `?` for Kleene closure, positive closure and optional items respectively, as well as parentheses and the `|` operator to allow alternates to be gathered together. The symbol `#` represents the empty string ϵ. We also provide the form `A@B` which is shorthand for `A (B A)*`.

To support scannerless parsing, terminals come in three forms. The most fundamental is the *character terminal* denoted by a back-quote followed by either a printable character or one of the ANSI-C conventional character escape sequences. These are designed to be used in lexical level specifications and we also allow the shorthands `‘x-‘z` and `\‘x` to represent sets of character tokens in the range `x` through `z` and the set that includes all character tokens except `x`. The range operator may only be applied between operands that are either both digits, both lowercase or both uppercase characters because ART does not expect any other sequences to have a portable ordering.

Whether ART uses a separate lexer, specified in this way, or deploys the GLL algorithm right down to character level (in the manner of scannerless parsing in SDF [7]) is user selectable: there are theoretical and engineering implications which are beyond the scope of this paper but which will be discussed in future presentations.

Case sensitive terminals are demarcated by single quotes, and represent a shorthand for a whitespace nonterminal followed by the sequence of character terminals corresponding to the pattern of the terminal; that is `'while'` is just shorthand for `(artWhiteSpace ‘w ‘h ‘i ‘l ‘e)`.

The nonterminal `artWhiteSpace` is predefined by ART to correspond to the nonprinting characters; if the user provides explicit productions then the internal default is suppressed. ART specifications are modular, as we describe in the next section, and each module may have its own `artWhiteSpace` definition.

Case insensitive terminals are demarcated by double quotes, and represent a shorthand for a whitespace nonterminal followed by the sequence of character terminals corresponding to the mixed-case pattern of the terminal; that is `"while"` is shorthand for

```
(artWhiteSpace (‘w|‘W) (‘h|‘H) (‘i|‘I) (‘l|‘L) (‘e|‘E)).
```

Nonterminals and terminals may be *named* by appending a colon and an alphanumeric identifier. Names are used in semantic expressions to disambiguate multiple instances of a nonterminal, and to identify torn subtrees that will be reinserted into an RDT.

ART modules export a list of nonterminals. Non-exported nonterminals are private to a module and invisible outside of their parent module. A module import list comprises import entities written as `M.X = Y`. This asks for the productions from module `M` whose left hand side is `X` to be copied into the current module but with their left-hand side name changed to `Y`. The simpler form `M.X` copies the productions for `X` with the same local name as in the exporting module, and the form `M` simply copies all rules that are exported from `M` with their original names. When a rule `X ::= Z1 Z2 ... Zk` is exported by module `M` and imported into module `N` via the instruction `M.X = Y`, the copy rule is properly written `N.Y ::= M.Z1 M.Z2 ... M.Zk`, in particular if `Zi=X` then `M.Zi` is different from `N.Y`. In a derivation using this rule the grammar rules for `Zi` in module `M` are used to expand `M.Zi`.

Module re-use sometimes demands that we modify imported nonterminal definitions. Extra rules may be added locally simply by writing them into the importing module. However, we can also remove productions from a nonterminal by using the *deleter* `X \ ::=` α. In this case, all rules of the form `X ::=` α (after suppression of TIF annotations and semantic actions) are deleted from the module containing the deleter.

An attempt to recursively import a module is an error, the dependency graph of modules must form a directed acyclic graph. During module resolution, we say a module M is *complete* if all of the modules in its import list are complete, and all imports to M have been performed and all deleters in M have been applied. In practice this means that we start at the leaves of the dependency DAG and work our way back to the root module.

The order of modularity operations for M is as follows:

1. Construct internal representations of all of the rules in the source text for M.
2. When all modules listed as imports to M are complete, execute each import in M and apply renamings.
3. Apply deleters in M to the resulting set of rules.
4. Construct the export list from M.
5. Mark M as complete.

ART uses GLL-style parsing [2] which is a generalisation of recursive descent using the process management regime from our RIGLR parsers. A feature of GLL is that the parser is defined in terms of a small series of *templates* corresponding to various grammar idioms. Converting ART to produce parsers and tree walkers written in a new programming language requires us to write templates in that language and to implement a small set of support routines. ART generates C++; we plan to implement Java templates next.

The present ART implementation uses RDP to generate its front end: ART has not yet been ported to itself. When a bootstrapped version of ART becomes available, some aspects of the source syntax (such as the trailing ; on productions) will become optional: they are only there to ensure that ART's source syntax is LL(1) and thus admissible by RDP.

A major space component of the generated parsers is the bit strings associated with the selector sets that guard the activation of individual productions. For performance reasons, we do not wish to implement the well known compression techniques used in table driven parsers, but we have observed that for typical programming languages a 75% reduction in space is obtainable simply by storing sets by contents rather than by name because many selector sets have the same contents. For ANSI-C, there are 866 selector sets in the parser but only 218 contents-unique selector sets.

3 Tear-Insert-Fold Annotations

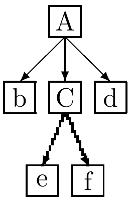
The TIF formalism described in [3] provides four tree manipulation operators. We imagine the tree as being drawn on a piece of paper, and allow for a node to be folded under or over its parent or for complete subtrees to be torn away and then reattached in a different position under the same parent. Essentially we use the derivation tree of a string as a starting point, and then rearrange it into an RDT as specified by the TIF operators.

The four TIF operations are carefully designed to be completely local: we do not allow subtrees to be torn and reattached to any node other than the original parent. This allows us to write an algorithm which generates a specification for the possible output trees in the form of another grammar which we call the TIF Transformed Grammar (TTG): the derivation trees of the TTG are the RDT's of the original grammar.

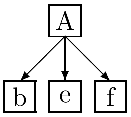
We shall illustrate the TIF operators using the following grammar:

```
A ::= 'b' C 'd';    C ::= 'e' 'f';   X ::= 'y' ;
```

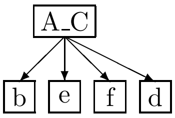
Nonterminal `X` is unreachable from start symbol `A` but will be used in an insertion. The derivation tree for string `befd` is



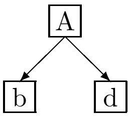
Now, if we add a *fold-under* (^) TIF operator to `C` and `d` in `A` giving `A ::= 'b' C^ 'd'^;` the derivation tree is transformed to this RDT:



which has TTG `A ::= 'b' 'e' 'f';` Similarly, if we add the *fold-over* (^^) operator giving `A ::= 'b' C^^ 'd';` then the derivation tree would be transformed to this RDT:



with TTG `A_C ::= 'b' 'e' 'f' 'd';`, `A_C` is a new nonterminal. The *tear* operator (^^^) removes an entire sub-tree: the rule `A ::= 'b' C^^^ 'd';` yields this RDT:

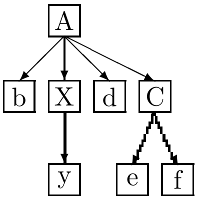


with TTG `A ::= 'b' 'd'; .`

We can also perform arbitrary insertions within rules using the `$` operator which may be applied to terminals, nonterminals that generate singleton languages and *named tears*. The TIF production

```
A ::= 'b' C:tearName^^^ [ $X ] 'd' [ $tearName ];
```

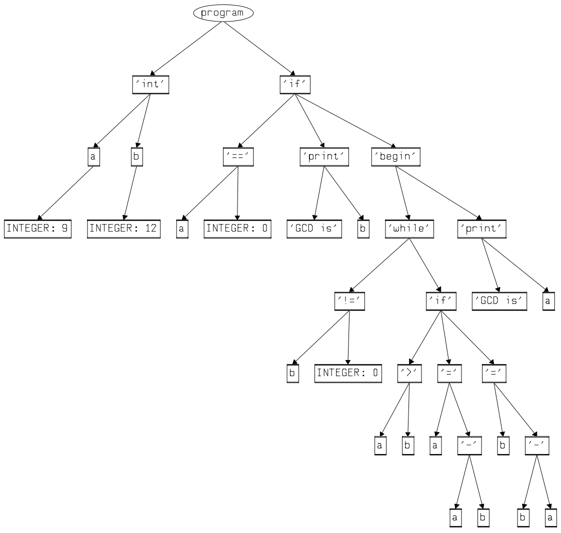
tears the subtree generated b nonterminal C, inserts a tree corresponding to the single derivation in the language of `X` and then re-inserts the torn tree:



with TTG `A ::= 'b' X 'd' C;   C ::= 'e' 'f';   X ::= 'y' ;`

As a more substantial example, here is the source code for Euclid's Greatest Common Divisor algorithm written in the `mini` language originally designed as a tutorial example for RDP together with the RDT used by `mini`'s code generator.

```
int a = 9,
    b = 12;

if a == 0 then
  print("GCD is", b)
else

  begin

    while b != 0 do
      if a>b then
        a = a - b
      else
        b = b - a;

    print("GCD is", a)

  end;
```



The RDT was produced using the following TIF grammar.

```
*mini(declarations statements)(program)
program ::=  #^ | (var_dec | statement | #^ ) ';'^ program^ ;

*declarations(lexer expressions)(var_dec)
var_dec ::= 'int'^^ dec_body  dec_list^ ;
dec_list ::= #^ | ','^ dec_body dec_list^ ;

dec_body ::= id^^ ( '='^ e0 )? ;

*statements(lexer expressions)(statement)
statement ::= id '='^^ e0  |
              'if'^^ e0 'then'^ statement ( 'else'^ statement )? |
              'while'^^ e0 'do'^ statement |
              'print'^^ '('^ ( e0  | stringLiteral ) print_list^ ')'^ |
              'begin'^^ statement stmt_list^ 'end'^;

print_list ::= #^ | ','^ ( e0  | String ) print_list^ ;

stmt_list ::= #^ | ';'^ statement stmt_list^ ;

*expressions(lexer)(e0)
e0 ::= e1 ('>'^^ e1 | '<'^^ e1  | '>='^^ e1 | '<='^^ e1 | '=='^^ e1 | '!='^^ e1)? ;
e1 ::= e2^^ | e1 ('+'^^ | '-'^^) e2 ;
e2 ::= e3^^ | e2 ('*'^^ | '/'^^) e3 ;
e3 ::= e4^^ | '+'^ e3^^ | '-'^^ e3 ;
e4 ::= e5^^ | e5 '**'^^ e4 ;
e5 ::= id^^ | integerLiteral^^ | '('^ e1^^ ')'^ ;

*lexer()(id integerLiteral stringLiteral)
alpha ::= `a..`z | `A..`Z | '_' ;
digit ::= `0..`9;
id ::= alpha (alpha|digit)*;
integerLiteral ::= digit*;
stringLiteral ::= `" (..)* `";
```

4 Some Source-to-Source Conversion Examples

The ANSI-C **do-while** and **for** statements can be expressed using the **while** statement. They are provided for user convenience but mapping them onto the **while** statement means that target code has only to be specified for one type of intermediate form. The ANSI-C production describing iteration statements is:

```
iteration_statement ::=
  'while' '(' expression ')' statement  |
  'do' statement 'while' '(' expression ')' ';'  |
  'for' '(' ( expression )? ';' ( expression )? ';'
                                  ( expression )? ')' statement ;
```

The following TIF rules parse the C `for` and `do` constructs respectively but generate RDT's corresponding to the equivalent `while` statements.

```
mappedForLoop ::=
        'for'^ '('^ expr:init^^^ ';'^ expr:test^^^ ';'^ expr:step^^^ ')'^
        [ $init ';' 'while' '(' $test ')' '{' ] statement [ ';' $step '}' ] ';' ;

mappedDoLoop ::=
        'do'^ '{'^ statement:body^^^ '}'^ 'while'^ '('^ expr:test^^^ ')'^ ';'^
        [ $body ';' 'while' '(' $test ')' $body ';' ] ;
```

We can also use TIF annotations to generate a uniform intermediate form from restricted cases of generic constructs. The following TIF rules generate an RDT from a `for` loop which has a root node labelled `for` and exactly four children, three of them valid expression sub-trees and one an instance of statement.

```
expressionFriendlyForLoop ::=
  'for'^^ '(' Eexpr ';' Eexpr';' Eexpr ')' statement ;

Eexpr ::= expression^^ | # ['true'^^] ;
```

Empty expressions are remapped so that if the programmer chooses to omit one of the control expressions we insert the expression `true` instead of leaving a child labelled with the epsilon symbol (or indeed no child at all by some conventions). We choose `true` because the C semantics for `for` specifies that the default for the step expression is that loop execution continues forever. C always discards the result of the initialisation and step expressions; only the side effects are used. Hence any side-effect free expression is a valid default.

5 Concluding Remarks and Open Issues

ART is fast, powerful and unfinished. There are a variety of open issues that we are experimenting with, and we expect to modify the tool's behaviour in response to user experiences.

At present, ART directly implements EBNF parentheses and the `?` operator by multiplying out. Closures are handled by the auxiliary `gramconv` tool. As a result, ART only need generate parser templates for BNF grammars. We have developed parser templates for EBNF constructs which allow iteration within the GLL parser to directly and efficiently handle closures, but the exact form of the trees to be produced is the subject of further study: it is not clear for instance whether a Kleene closure matching the empty string should yield a node labelled ϵ in the SPPF or simply be suppressed.

We have syntax to support lexical level rules, but the exact form of the lexer/parser divide is not specified. ART can produce GLL parsers which truly run at the character level, but the resulting SPPF's can be very large. Alternatively, ART can interface to DFA style lexers.

The fold operators in the TIF formalism are inspired by RDP's fold operators. RDP has been used in a wide variety of industrial and research projects over the last 15 years, and we have confidence that the basic notions of folding are useful and comfortable for engineers. In detail, ART's folds work differently in the case where we have chains of fold operators, that is when we fold a rule which also has folds on its own right hand side. In RDP, a fold-under operator could promote a fold over operator which then reached up and over the original parent node. We have outlawed this behaviour in ART by ensuring that fold under operators take priority over fold overs. Interestingly, we have never found an instance of this construct in any real RDP grammar.

ART can perform insertions of nonterminals which generate singleton languages, that is languages with only one string. ART builds the derivation tree for that single sentence, and inserts it. In the TIF formalism as described in [3] an insertion may be made of a (N, s) pair in which the derivation of string s in the grammar whose start symbol is N is inserted. ART's present limitation to singleton languages is a restricted version of this: we intend to implement the full semantics in a future version.

Finally, we note the lack of change management capability we need to design a TIF metalanguage which described the annotations to be applied: this could then be interpreted by ART as part of the generation of Γ_i.

References

1. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/xt 0.17. a language and toolset for program transformation. Sci. Comput. Program. 72(1-2), 52–70 (2008)
2. Johnstone, A., Scott, E.: GLL parsing. In: Proc. 9th Workshop on Language Descriptions, Tools and Applications LDTA 2009 (2009)
3. Johnstone, A., Scott, E.: Tear-Insert-Fold grammars. In: LDTA 2010 Tenth Workshop on Language Descriptions, Tools and Applications (2010)
4. McPeak, S., Necula, G.C.: Elkhound: A fast, practical GLR parser generator. In: Duesterwald, E. (ed.) CC 2004. LNCS, vol. 2985, pp. 73–88. Springer, Heidelberg (2004)
5. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)
6. van den Brand, M.G.J., Heering, J., Klint, P., Olivier, P.A.: Compiling language definitions: the ASF+SDF compiler. ACM Transactions on Programming Languages and Systems 24(4), 334–368 (2002)
7. van den Brand, M.G.J., Heering, J., Klint, P., Olivier, P.A.: Compiling language definitions: the ASF+SDF compiler. ACM Transactions on Programming Languages and Systems 24(4), 334–368 (2002)

Empirical Language Analysis in Software Linguistics

Jean-Marie Favre[1], Dragan Gasevic[2], Ralf Lämmel[3], and Ekaterina Pek[3]

[1] OneTree Technologies, Luxembourg
[2] Athabasca University, Canada
[3] Universität Koblenz-Landau, Germany

Abstract. Software linguistics is the science of software languages. In this short paper, we sketch the general discipline of software linguistics, but our focus is on one part of it: empirical analysis of software languages. Such analysis is concerned with understanding language usage on the grounds of a corpus. In this short paper, we sketch a survey on empirical language analysis, and we argue that the research method of content analysis is needed for a thorough survey.

1 Introduction

Software Language Engineering (SLE) or "Software Languages are Software too". Software language descriptions and processors are pieces of software. Hence, all kinds of Software Engineering concepts and techniques can be adopted to software languages. SLE is about the systematic design, implementation, deployment, and evolution of software languages [14]. Clearly, software language descriptions have particular properties, when compared to other kinds of artifacts in software engineering. Hence, traditional software engineering life-cycles, methods, set of qualities and constraints must be genuinely adapted. If we think about the distinction of software languages vs. natural languages, then Software Language Engineering can be compared to the established field of *Natural Language Engineering* [6,10].

Software Linguistics (SL) or "Software Languages are Language too". SLE practices should be informed by scientific knowledge. In the case of natural languages, linguistics is the science of languages [5]. Hence, it is worth to see which concepts, research methods, perhaps even techniques or results from the field of linguistics could be adopted to the study of software languages. In this manner, we obtain "Software Linguistics". The term *Software Linguistics* was introduced by Misek-Falkoff in 1982 [18]. This term and the whole concept of adopting linguistics for software (programming) languages has not seen much interest. We refer to [8,17] for some controversy. We note that Software Linguistics should not be confused with Computational Linguistics—the former is "linguistics for software languages"; the latter is (simply speaking) "computing for linguistics" (for natural languages).

Towards a survey on empirical software language analysis. A common form of software linguistics is empirical analysis of software language where one is concerned with understanding usage of software languages on the grounds of a corpus. We are interested in any kind of software language. Such analysis has been carried out for many software languages and many aspects of language usage. In this short paper, we begin a corresponding survey of a collection of publications on the subject of empirical software language analysis. This is work in progress, and we discuss how we expect to use the research method of content analysis [15] for the continuation of the project.[1]

Road-map. In Sec. 2, we contextualize the present paper in reference to linguistics for natural and software languages. In Sec. 3, we begin the survey of a collection of papers on empirical language analysis. In Sec. 4, we discuss the employment of the research method of content analysis for completing the current efforts on surveying empirical language analysis in a systematic manner. In Sec. 5, we conclude the paper.

2 Broader Research Context: Software Linguistics

Empirical software language analysis is an important part of Software Linguistics. A survey on such work helps understanding and improving the state of the art in empirical software language analysis—also specifically with regard to scientific methodology. Such a surveying effort also exercises the adoption of regular linguistics to software.

In this section, we would like to contribute to a broader understanding and definition of Software Linguistics—through references to (Natural) Linguistics. We have found that the mature, scientific framework provided by linguistics can be reused for software languages—even though many techniques related to natural languages may not be (directly) applicable. Much can be reused beyond the classical separation of different levels such as syntax, semantics and pragmatics. In fact, one can systematically mine Software Linguistics from resources such as "The Cambridge encyclopedia of language" [5]. A few examples are given below.

Comparative linguistics studies, compares and classifies languages according to their features using either a quantitative or qualitative approach. It aims at identifying patterns that are recurrent in different languages, but also differences and relationships between languages. Comparative linguistics may also apply to software languages. For instance, "Programming Linguistics" [7] compares programming languages in terms of commonalities, relationships, and differences while discussing basic matters of syntax, semantics and styles. "The comparison of programming languages: A linguistic approach" [12] goes deeper into linguistic aspects. We also refer to [2,19].

Historical linguistics studies the history and evolution of languages—often with the goal of identifying language families, that is, languages that derive from a common ancestor. Part of this research compensates for the lost origin of natural languages. In the case of software languages, history is often well documented. Consider, for example, the History of Programming Languages (HOPL) conference series. HOPL focuses on programming languages rather than software languages in general. Also, HOPL does

[1] There is the SourceForge project `http://toknow.sourceforge.net/` that is designated to this surveying effort on empirical analysis of software languages. The project hosts (identifies) a paper collection, in particular. In the present paper, references to the papers of the collection use angle brackets as in "⟨Knuth71⟩".

not stipulate systematic linguistics studies; a typical paper relies on historical accounts by the primary language designers. Some reports, though, provide a large set of qualitative information regarding the evolution of dialects of languages, e.g., [1]. The *impact* of the evolution of software languages on software evolution, though real, is not well understood [9], and deserves more research inspired by linguistics.

Geo-linguistics studies the intersection of geography and linguistics. Studies can, for example, take into account the distribution of languages or dialects over countries, continents, and regions. We encounter a related "clustering" dimension for software languages in Sec. 3. Also, in [19], the emergence of dialects of Lisp are considered in terms of geographical zones.

Socio-linguistics studies languages as social and societal phenomena. The design of software languages and dialects is often directly linked to such phenomena, too. For instance, in [19] Steele and Gabriel conclude "Overall, the evolution of Lisp has been guided more by institutional rivalry, one-upsmanship, and the glee born of technical cleverness that is characteristic of the hacker culture than by sober assessments of technical requirements". The field of socio-linguistics for software remains largely unexplored, but see [3] for a related account.

Corpus linguistics deals with all aspects of designing, producing, annotating, analyzing and sharing corpora. Producing a useful, natural linguistics corpus could be an effort that goes far beyond what individual researchers or teams can do. There are international associations who support sharing, e.g., the European Language Resources Association (ELRA).[2] One should assume that empirical research on the usage of software languages also involves efforts on 'software corpus engineering/linguistics'. Such software corpus linguistics should be simplified by the fact that software language artifacts are inherently digitally stored. However, the survey of Sec. 3 shows that corpora are too often unavailable or unreproducible.

3 Towards a Survey on Empirical Language Analysis

In the following, we describe the beginning of a survey on empirical language analysis. In particular, we identify first research questions, and we provide a corresponding coding scheme à la content analysis [15].

3.1 Paper Collection

At the time of writing, we have accumulated 52 papers on empirical analysis of software languages. As an illustration, Fig. 1 shows the language distribution for the full collection. For reasons of brevity and maturity of all metadata, all of the subsequent tables will focus on a *selective collection* of 17 papers. Both the full and the selective collections are described online; see footnote 1.

[2] ELRA website: `http://www.elra.info/`

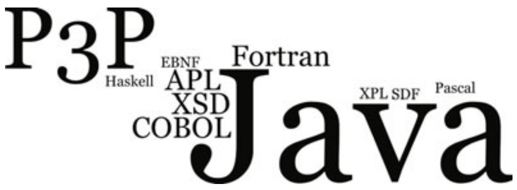


Fig. 1. Tag cloud for the language distribution for the underlying paper collection

3.2 Research Questions

i) Each paper in the collection involves a corpus of a chosen software language. What are the characteristics of those corpora; refer to Sec. 3.4? ii) Each empirical analysis can be expected to serve some objective. What are those objectives for the collection of papers; refer to Sec. 3.5? iii) Each empirical analysis can be expected to leverage some actual (typically automated) analyses on the corpus. What are those analyses for the collection of papers; refer to Sec. 3.6?

3.3 Terminology

We use the term (software) *corpus* to refer to a collection of *items* that are expressed in the language at hand. (These items may be valid or invalid elements of the language in a formal sense.) Items originate from possibly several *sources*. We use the term *source* to refer to different kinds of physical or virtual sources that contribute items. For instance, a corpus may leverage an open-source repository as a source to make available, or to retrieve the items—based on an appropriate search strategy. A paper may provide a *corpus description* that identifies sources and explains the derivation of the actual corpus (say, item set) from the sources.

3.4 Corpus Characteristics

Fig. 2 provides metadata that we inferred for the corpora of the selective paper collection.[3] We capture the following characteristics of the software corpora: the software language of the corpus, numbers of sources and items (with a suitable unit), the online *accessibility* of the sources (on a scale of *none*, *partial*, and *full*), and the *reproducibility* of the corpus (on a scale of *none*, *approximate*, *precise*, and *trivial*). We say that reproducibility is *trivial*, if the corpus is available online—in one piece; reproducibility is *precise*, if the sources and the corpus description suffice to reproduce the corpus precisely by essentially executing the corpus description. Otherwise, we apply a judgement call, and use the tags *approximate* or *none*. For instance, the inability to reproduce a ranking list of a past web search *may* be compensated for by a new web search, and hence, reproducibility can be retained at an approximate level. In future work, we would like to go beyond the characteristics that we have sketched here.

[3] A cell with content "?" means that the relevant data could not be determined.

	language	sources	items	unit	accessibility	reproducibility
⟨AlvesV05⟩	SDF	8	27	grammars	partial	approximate
⟨BaxterFNRSVMT06⟩	Java	17	56	projects	full	precise
⟨ChevanceH78⟩	COBOL	1	50	programs	none	none
⟨CollbergMS04⟩	Java	1	1132	JAR files	full	approximate
⟨CookL82⟩	Pascal	1	264	programs	none	none
⟨CranorESMC08⟩	P3P	3	?	policies	full	approximate
⟨GilM05⟩	Java	4	14	projects	full	precise
⟨HageK08⟩	Haskell	1	68000	compilations	none	approximate
⟨Hahsler04⟩	Java	1	988	projects	full	approximate
⟨Hautus02⟩	Java	?	9	packages	full	approximate
⟨KimSNM05⟩	Java	2	2	programs	full	approximate
⟨Knuth71⟩	Fortran	7	440	programs	none	none
⟨LaemmelKR05⟩	XSD	2	63	schemas	partial	none
⟨LaemmelP10⟩	P3P	1	3227	policies	full	trivial
⟨ReayDM09⟩	P3P	1	2287	policies	full	approximate
⟨SaalW77⟩	APL	6	32	workspaces	none	none
⟨Visser06⟩	XSD	9	9	schemas	full	approximate

Fig. 2. Corpus characteristics for selective paper collection

3.5 Objectives of the Papers

Based on our (preliminary) analysis of the paper collection, we propose the following (preliminary) list of objectives for empirical language analysis; see Fig. 3 for corresponding metadata for the selective paper collection.

Language adoption. The objective is to determine whether the language is used, and with what frequency. Typically, some scope applies. For instance, we may limit the scope geographically, or on the time-line.

Language habits. The objective is to understand the usage of the language in terms of syntactically or semantically defined terms. For instance, we may study the coverage of the language's diverse constructs, or any other, well-defined metrics for that matter. This objective may be addressed with substantial measurements and statistical analysis.

Language taming. The objective is to impose extra structure on language usage so that habits can be categorized in new ways. For instance, we may equip the language with patterns or metrics that are newly introduced or adopted from other languages. In some cases, the empirical effort towards addressing the objective of language taming may also qualify as effort that attests to the objective of language habits.

User feedback. The objective is to compile data of any kind that helps the language user to better understand or improve programs. For instance, we may carry out an analysis to support the proposal of a new pattern that should help with using the language more effectively. This objective could be seen as a more specific kind of language taming.

	⟨AlvesV05⟩	⟨BaxterFNRSVMT06⟩	⟨ChevanceH78⟩	⟨CollbergMS04⟩	⟨CookL82⟩	⟨CranorESMC08⟩	⟨GilM05⟩	⟨HageK08⟩	⟨Hahsler04⟩	⟨Hautus02⟩	⟨KimSNM05⟩	⟨Knuth71⟩	⟨LaemmelKR05⟩	⟨LaemmelP10⟩	⟨ReayDM09⟩	⟨SaalW77⟩	⟨Visser06⟩
language adoption						●			●						●		
language habits		●		●	●	●			●					●	●	●	
language taming	●						●			●	●		●				●
user feedback										●							
language evolution				●			●										
user behavior								●									
implementor feedback			●	●	●							●					

Fig. 3. Objectives of the selected publications

Language evolution. The objective is to gather input for design work on the next version of the language. For instance, we may try to detect indications for missing constructs.

User behavior. The objective is to understand the usage of the language and its tools in a way that involves users or user experiences directly—thereby going beyond the narrow notion of corpus consisting only of "programs". For instance, we may analyse instances of compiler invocations with regard to problems of getting programs to compile eventually.

Implementor feedback. The objective is to understand parameters of language usage that help language implementors to improve compilers and other language tools. For instance, we may carry out an analysis to suggest compiler optimizations.

Without going into detail here, the available data caters for various observations. For instance, we realize that research on language adoption is generally not exercised for programming languages. It appears that online communications but not scientific publications are concerned with such adoption.[4,5,6]

3.6 Analyses of the Papers

Based on our (preliminary) analysis of the paper collection, we have come up with a simple hierarchical classification of (typically automated) analyses that are leveraged in the empirical research projects; see Fig. 4 for the classification; see Fig. 5 for corresponding metadata for the selective paper collection.

The presented classification focuses on prominent forms of static and dynamic analysis. In our paper collection, static analysis is considerably more common, and there

[4] The TIOBE Index of language popularity: `http://www.tiobe.com/tpci.htm`

[5] Another web site on language popularity: `http://langpop.com/`

[6] Language Popularity Index tool: `http://lang-index.sourceforge.net/`

Static analysis	Source code or other static entities are analyzed.
Validity	The validity of items in terms of syntax or type system is analyzed.
Metrics	Metrics are analyzed.
Size	The size of items is analyzed, e.g., in terms of lines of code.
Complexity	The complexity of items is analyzed, e.g., the McCabe complexity.
Structural properties	Example: the depth of inheritance hierarchy in OO programs.
Coverage	The coverage of language constructs is analyzed.
Styles	The usage of coding styles is analyzed.
Patterns	The usage of patterns, e.g., design patterns, is analyzed.
Cloning	Cloning across items of the corpus is analyzed.
Bugs	The items are analyzed w.r.t. bugs that go beyond syntax and type errors.
Dynamic analysis	Actual program runs are analyzed.
Profiles	Execution frequencies of methods, for example, are analyzed.
Traces	Execution traces of method calls, for example, are analyzed.
Dimensions of analysis	Orthogonal dimensions applicable to analyses.
Evolution	An analysis is carried out comparatively for multiple versions.
Clustering	The corpus is clustered by metadata such as country, team size, or others.

Fig. 4. Classification of analyses

	⟨AlvesV05⟩	⟨BaxterFNRSVMT06⟩	⟨ChevanceH78⟩	⟨CollbergMS04⟩	⟨CookL82⟩	⟨CranorESMC08⟩	⟨GilM05⟩	⟨HageK08⟩	⟨Hahsler04⟩	⟨Hautus02⟩	⟨KimSNM05⟩	⟨Knuth71⟩	⟨LaemmelKR05⟩	⟨LaemmelP10⟩	⟨ReayDM09⟩	⟨SaalW77⟩	⟨Visser06⟩
validity						●		●						●	●		
metrics	●	●		●	●			●	●	●			●	●		●	●
coverage			●	●	●	●	●					●	●	●		●	
styles				●			●			●			●	●			
patterns							●		●								
cloning											●			●			
bugs						●								●	●		
profiles			●					●				●					
traces																	
evolution						●		●					●				
clustering						●			●						●		

Fig. 5. Analyses of the selected publications

is a substantial variety of different analyses. We also indicate two additional dimensions for analyses. An analysis is concerned with *evolution*, when different versions of items, sources, or languages are considered. The dimension of *clustering* generalizes geo-linguistics of Sec. 2. By no means, our classification scheme is complete. For instance, we currently miss characteristics regarding data analysis (e.g., in terms of the involved statistical methods), and the presentation of research results (e.g., in terms of the leveraged tables, charts, etc.).

4 Outlook on Research Methodology

The survey is work in progress. We are in the process of assessing the feasibility of such a survey, discovering research questions, and studying applicable research methodology. Ultimately, we aim at reproducible research results and assessment of validity and limitations. In this discussion section, we briefly hint at the utility of content analysis for the continuation of this research.

4.1 Content vs. Meta-analysis

We consider two research methods commonly used in social and health sciences, namely, *content analysis* [15] and *meta-analysis* [11]. Content analysis is a qualitative research method commonly used for systematically studying large amounts of communication content such as news, articles, books, videos, or blogs. The key characteristic is that the analyzed content is *categorized* by researchers. Systematic literature surveys [4] often leverage content analysis.

Meta-analysis is another research method used for a systematic analysis of research literature. Unlike content analysis, meta-analysis is a quantitative method, and it is focused on research studies that have highly related research hypotheses. The main goal of a meta-analysis is to aggregate and statistically analyze findings of several research studies. Meta-analysis uses stricter inclusion criteria than content analysis: measured outcomes, and sufficient data to calculate effect sizes. These specifics of meta-analysis challenge its application to empirical software engineering [13]. As a result, we focus on content analysis for the ongoing survey.

4.2 Adoption of Content Analysis

Let us synthesize the main steps for applying content analysis for studying literature on empirical software language engineering; here we follow the advice of [4]. (Content analysis could serve as a convenient research method for studying not only research publications, but also other types of communications such as online discussions about languages, news articles, or bug report histories.)

1. Formulate research questions. In Sec. 3, we listed research questions related to corpus characteristics, objectives and analyses. We may add questions, for example, about i) the correlation between different factors (e.g., languages vs. objectives), ii) the adopted research method, iii) the statistical techniques that are leveraged, and iv) the presentational tools (figures, charts) etc. that are used in the papers.

2. Formulate inclusion criteria for papers. These criteria are derived from the research questions. So far, we have been using these criteria: i) the paper discusses usage of a software language, and ii) the paper reports empirical results on such usage based on a corpus. These constraints may need to be made more precise to avoid inclusion of "irrelevant" papers.

3. Collect data. That is, we need to discover the relevant literature. Our preliminary collection has been obtained in an ad-hoc manner. We harvested several papers from the literature work that went into our prior, related research: ⟨LaemmelKR05⟩, ⟨LaemmelP10⟩. We examined first-degree and second-degree references. While searching those papers on the web, we encountered a few more "obvious" candidates.

A systematic approach commences as follows. First, we need to determine the *sources* from which the papers will be collected. To this end, we may select digital libraries (e.g., ACM, IEEE, Science Direct, and Springer), and other online sources (DBLP[7]), where the papers will be searched. Next, we need to define the *search strategy*. In particular, we need methodologically defined keywords including constraints on their combination so that the search strategy is reproducible and amenable to assessments. Ultimately, the search strategy needs to be executed so that all those papers are collected that satisfy the inclusion criteria.

4. Evaluate data. The collected data is evaluated in terms of some classification scheme, which is commonly called *coding scheme*, and which consists of a set of categories that describe the studied domain. In some cases, a (part of a) coding scheme can be defined up-front. For instance, we designed the corpus characteristics of the papers up-front; refer to Sec. 3.4. In general, categories may emerge during evaluation. For instance, we perceived the objectives for empirical language analysis only post-priori; refer to Sec. 3.5. There is a systematic process for developing a coding scheme based on the iteration of certain steps of discovery and testing by other researchers [20].

Each paper in the collection has to be coded eventually. Coding is typically done by two researchers (so-called raters) independently. Two authors of the present paper have acted as raters in some limited manner. For a proper execution, it is inevitable to keep track of original codes of the researchers and the reconciled ones. This is needed for the computation of a measure of accuracy of the coding scheme.

As the process of data collection and coding might be rather labor-intensive, content analysis may leverage some more automated approaches. For example, text mining is one possible approach [16], and off-the-shelf tools—such as AeroText and SPSS—provide corresponding support. It will be interesting to see how well text mining works in our domain.

5. Analyse data. Frequencies of each category are to be calculated and reported. The frequencies describe the level of coverage of a certain phenomenon of interest studied in the domain. As these frequencies are obtained through a systematic research process, they can nicely reflect on the main types of problems the research has been focused on, and probably discover some patterns about missing research. Such analysis also helps giving relatively objective answers to the research questions.

6. Publish analysis. The publication must describe all methodological steps, and include all data needed for reproducibility. For instance, inclusion of the discovered number of papers (per used source), and included papers are mandatory. Similarly, details of data evaluation and data analysis need to be included. For instance, measures of

[7] `http://www.informatik.uni-trier.de/~ley/db/`

accuracy of the coding scheme are to be included. Further, a qualitative discussion of the surveyed papers must be provided.

5 Concluding Remarks

We have described the beginning of a literature survey on empirical language analysis. We have presented first results of the emerging survey and discussed details of the underlying research method. We hope to get other software language engineers and empirical software engineers as well as software linguists or even classical linguists involved in this effort.

In the work on the survey, so far, we have found it inspirational to consult linguistics (for natural languages). In particular, linguistics provides input for the categorization needed in the survey, and it suggests underrepresented areas of empirical language analysis. The other interesting insight has been that a survey on empirical language analysis, in itself, also calls for an empirical method, and we have found that content analysis provides a good fit to produce a first survey on empirical language analysis.

References

1. Barbara, M.B., Ryder, G., Soffa, M.L.: The impact of software engineering research on modern progamming languages. ACM Transactions on Software Engineering and Methodology 14(4), 431–477 (2005)
2. Baron, N.S.: How to Make Your Way through the New Language Maze - Computer Languages: a Guide For The Perplexed. Doubleday (1986)
3. Bertil Ekdahl, L.J.: The difficulty in communicating with computers. In: Interactive Convergence: Critical Issues in Multimedia, ch. 2. Inter-Disciplinary Press (2005)
4. Cooper, H.M.: The integrative research review: A systematic approach. Applied social research methods series, vol. 2. Sage, Thousand Oaks (1984)
5. Crystal, D.: The Cambridge Encyclopedia of Language, 2nd edn. Cambridge University Press, Cambridge (2005)
6. Cunningham, H.: A definition and short history of language engineering. Journal of Natural Language Engineering 5, 1–16 (1999)
7. David Gelernter, S.J.: Programming Linguistics. MIT Press, Cambridge (1990)
8. Dijkstra, E.W.: On the foolishness of "natural language programming". In: Program Construction, International Summer Schoo, London, UK, pp. 51–53. Springer, Heidelberg (1979)
9. Favre, J.-M.: Languages evolve too! changing the software time scale. In: IEEE (ed.) 8th Interntational Workshop on Principles of Software Evolution, IWPSE (September 2005)
10. Georgiev, H.: Language Engineering. Continuum (2007)
11. Glass, G.V., McGaw, B., Smith, M.L.: Meta-analysis in social research. Sage, Beverly Hills (1981)
12. Goodenough, J.: The comparison of programming languages: A linguistic approach. In: ACM/CSC-ER (1968)
13. Kitchenham, B., Al-Khilidar, H., Babar, M.A., Berry, M., Cox, K., Keung, J., Kurniawati, F., Staples, M., Zhang, H., Zhu, L.: Evaluating guidelines for reporting empirical software engineering studies. Empirical Software Engineering 13(1), 97–121 (2008)
14. Kleppe, A.: Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Addison-Wesley Professional, Reading (2008)

15. Krippendorff, K.: Content Analysis: an Introduction to Its Methodology, 2nd edn. Sage, Thousand Oaks (2004)
16. Leech, N.L., Onwuegbuzie, A.J.: Qualitative data analysis: A compendium of techniques and a framework for selection for school psychology research and beyond. School Psychology Quarterly 23(4), 587–604 (2008)
17. Mahoney, M.: HOPL II - closing panel: The history of programming: Does our present past have a future? ACM SIGPLAN Notices 31(11), 15–37 (1996)
18. Misek-Falkoff, L.D.: The new field of ”software linguistics”: An early-bird view. In: ACM (ed.) ACM SIGMETRICS Workshop on Software Metrics (1982)
19. Steele Jr., G.L., Gabriel, R.P.: The evolution of Lisp. ACM SIGPLAN Notices, 231–270 (1993)
20. Taylor-Powell, E., Renner, M.: Analyzing qualitative data. In: Evaluation in Social Work: The Art and Science of Practice, ch. 13, University of Wisconsin Extension Program, Development and Evaluation Unit (2003)

Interactive Disambiguation of Meta Programs with Concrete Object Syntax

Lennart C.L. Kats[1], Karl T. Kalleberg[2], and Eelco Visser[1]

[1]Software Engineering Research Group, Delft University of Technology, The Netherlands
`l.c.l.kats@tudelft.nl, visser@acm.org`
[2]KolibriFX, Oslo, Norway
`karltk@kolibrifx.com`

Abstract. In meta-programming with concrete object syntax, meta programs can be written using the concrete syntax of manipulated programs. Quotations of concrete syntax fragments and anti-quotations for meta-level expressions and variables are used to manipulate the abstract representation of programs. These small, isolated fragments are often ambiguous and must be explicitly disambiguated with quotation tags or types, using names from the non-terminals of the object language syntax. Discoverability of these names has been an open issue, as they depend on the (grammar) implementation and are not part of the concrete syntax of a language. Based on advances in interactive development environments, we introduce *interactive disambiguation* to address this issue, providing real-time feedback and proposing quick fixes in case of ambiguities.

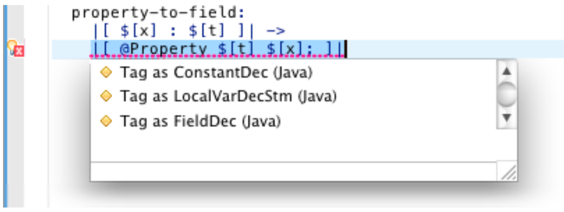
1 Introduction

Meta programs analyze, transform, and generate programs. Examples include compilers, interpreters, and static analysis tools. Most frequently, meta programs operate on the abstract syntax of an object language, using a structured representation of programs rather than a textual representation of their source code. Using a structured representation ensures well-formedness, enables compositionality of transformations, and makes it easier to support type safety and hygiene. However, manipulating the abstract syntax through an API can get tedious, and larger structures are often hard to recognize.

Meta-programming with concrete object syntax [15] as a surface syntax for the abstract representation is, for a great number of situations, a best of both worlds between a textual and an abstract syntax representation: the meta program is written using the familiar concrete syntax of the object language, while at the meta level, all operations are done on a structured representation of the object program. Concrete object syntax can be syntactically checked as meta programs are compiled. This technique is now supported by many meta-programming systems [2,3,4,7,12].

A prevailing problem with embedding concrete object syntax inside a meta-language is that the syntax of the combined meta-and-object language is usually highly ambiguous when the embedding employs a single pair of quotation and anti-quotation symbols. For example, a quoted Java code fragment `|[ i = 2 ]|` can either be an assignment expression, part of a local variable declaration, or even an annotation element initializer.

Two approaches have been proposed to address ambiguity in meta programs, each with their own trade offs and limitations. Perhaps the most straightforward approach



```
property-to-field:
  |[ $[x] : $[t] ]| ->
  |[ @Property $[t] $[x]; ]|
```

Fig. 1. Screenshot of a quick fix dropdown menu, listing three possible tags to disambiguate a Java quotation. The menu can be triggered by clicking on the error icon shown in the left margin, or by using a keyboard shortcut. Selecting a suggestion fixes the ambiguity.

is to use tagged quotation and anti-quotation symbols, e.g. writing `Expr |[ i = 2 ]|` using the tag `Expr` to indicate that the quotation contains an expression. The other approach is to use type information of the meta-programming language to attempt to select the intended interpretation of a concrete syntax quotation [5,13]. For example, for an embedding of Java in Java, a statement `Expr assign = |[ i = 2 ]|;` can be disambiguated based on the declared type `Expr`, while the quotation itself does not have to be explicitly tagged.

A pressing problem that both approaches share is a lack of *discoverability* of quotation tags and types. Meta-programmers may be intimately familiar with the concrete syntax of a language, but may not be well-grounded in the specific names of non-terminals in the syntax definition and the corresponding tag and type names. Having to know these names adds to the learning curve of meta-programming. Furthermore, as object languages evolve, or as additional object languages are added to a meta program, new ambiguities can be introduced for existing code that has not yet, or insufficiently, been explicitly disambiguated. Neither of the two approaches provides developers with adequate feedback if the developer must decide how to fix such an ambiguity.

In this paper we propose *interactive disambiguation* as a *complementary* approach to tag-based and type-based disambiguation that addresses the concern of discoverability. Our work builds on advances in interactive development environments (IDEs). Modern IDEs aid in discoverability of language features and APIs by providing features such as context-aware code completion and quick fixes. Quick fixes provide a facility to quickly fix common errors by selecting a fix from a list of suggestions. In this paper we propose to use quick fixes to present developers a list of candidate type or tag names for ambiguous concrete syntax fragments, allowing them to selectively fix problematic ambiguities and quickly discover possible fixes (illustrated in Figure 1). Our interactive disambiguation approach is *fully language independent* and does not have to be adapted for a specific meta-programming language or its type system.

2 Meta-programming with Concrete Object Syntax

In this section we recapitulate the general method for supporting concrete object syntax in meta languages and describe the problem of ambiguity. This method, as described in [15,6], is independent of the meta language used and relies on composition of the

```
action-to-java-method:
  |[ action $[Id:name] {
       $[Statement*:s*]
     }
  ]| ->
  |[ public void $[Id:name]() {
       $[Stm*:<statements-to-java> s*]
     }
  ]|
```

Fig. 2. A rewrite rule that uses concrete object syntax notation to rewrite a WebDSL action to a Java method. $s*$ is a meta variable containing a list of statements. $name$ contains an identifier.

syntax of the meta language and the object language. Based on this composition, the meta language can use *quotations* of object-level code to match and construct code fragments in the object language. In turn, quotations can include *anti-quotations* that escape from the object language code in order to include variables or expressions from the meta language. As an example, Figure 2 shows a Stratego [4] rewrite rule that uses quotations (indicated by `|[...]|`) and anti-quotations (indicated by `$[...]`) to rewrite a WebDSL [16] action definition to a Java method.

Grammar composition. Some meta-programming systems, such as Jak [3] and Meta-AspectJ [8], have been specifically designed for a fixed object language. These systems use a carefully handcrafted grammar or parser for the combined meta and object language. Other systems are more flexible and can be configured for different object languages by combining the grammar for the meta and object languages, and generating a corresponding parser using a parser generator. Building flexible meta-programming systems using traditional parser generators is very difficult, because their grammars are restricted to LL or LR properties. This means that conflicts arise when the grammar of the meta language and the object language are combined [6], and these must be resolved before the meta-and-object language parser can be constructed. A further impediment to language composition found in traditional parsers is the use of a separate scanner, requiring the use of a single lexical syntax definition for the combined language.

By using a combination of SDF for syntax definition and SGLR for parsing [6,14], any object language can be embedded in any meta language [15]. SGLR supports the full class of context-free grammars, which is closed under composition. This makes it possible to compose languages simply by combining grammar modules. *Mixin grammars* can combine languages by introducing productions for quotation and anti-quotation of an object language to a meta language. Mixin grammars can be written by hand or automatically generated using a tool.

As an example of a mixin grammar, Figure 3 shows an excerpt of a grammar that embeds Java into the Stratego program transformation language. Quotation productions have the form q_1 *osort* q_2 `->` *msort* and specify that a quotation of object-language non-terminal *osort*, surrounded by (sequences of) symbols q_1 and q_2, can be used in place of meta-language non-terminal $msort$. We sometimes refer to q_1 and q_2 collectively as the *quoting symbols*. In most of our examples, q_1 is `|[` and q_2 is `]|`, or a variation thereof.

```
module Stratego-Java
imports Stratego Java[1]
exports context-free syntax
% Quotations %
  "|[" ClassDec "]|" -> Term {cons("ToMetaExpr")}
  "|[" BlockStm "]|" -> Term {cons("ToMetaExpr")}
  "|[" CompUnit "]|" -> Term {cons("ToMetaExpr")}
% Anti-quotations %
  "$[" Term "]" -> ClassDec {cons("FromMetaExpr")}
  "$[" Term "]" -> BlockStm {cons("FromMetaExpr")}
  "$[" Term "]" -> CompUnit {cons("FromMetaExpr")}
```

Fig. 3. A mixin grammar for embedding object language Java into host language Stratego. For selected Java non-terminal, the mixin defines productions for quoting and anti-quoting. `ClassDec`, `BlockStm`, `CompUnit` are defined by the `Java` grammar.

```
     "CompUnit" "|[" BlockStm "]|" -> Term {cons("ToMetaExprTagged1")}
"Java:CompUnit" "|[" BlockStm "]|" -> Term {cons("ToMetaExprTagged2")}
"$[" "BlockStm" ":" Term "]" -> BlockStm   {cons("FromMetaExprTagged")}
```

Fig. 4. Productions with tagged quoting symbols

Conversely, anti-quotation productions have the form q_1 *msort* q_2 `->` *osort*. They specify that an anti-quotation of meta-language non-terminal *msort*, using quoting symbols q_1 and q_2, can be used in place of object-language non-terminal *osort*.

In our example we combine a single meta language with a single object language. It is also possible to add additional object languages or embeddings and extensions inside object languages. Using nestable quotations and anti-quotations, meta and object language expressions can be arbitrarily nested.

Ambiguity. As the meta language and object language are combined, *ambiguities* can arise in quotations and anti-quotations. Quotations and anti-quotations are ambiguous if they can be parsed in more than one way, leading to multiple possible abstract syntax representations. Ambiguities can also occur if the same quoting symbols are used for (anti-)quotation of multiple non-terminals. Such ambiguities can be avoided by using quoting symbol tags that indicate the kind of non-terminal or by using type information from the meta language [5,13]. Both approaches use names based on the non-terminals in a syntax definition for the object language. Without loss of generality, we focus on a combination of tag-based disambiguation with interactive disambiguation in this paper.

As an example of tagged quoting symbols, Figure 4 shows tagged productions tag (anti-)quotations for the `CompUnit` non-terminal. We indicate the kind of tag in the constructor of these productions. For untagged quotation productions we use `ToMetaExpr`, for productions with a non-terminal name we use `ToMetaExprTagged1` and for productions that also include a language prefix we use `ToMetaExprTagged2`. The last

[1] This example uses plain imports to combine the meta and object languages (Java and Stratego). To avoid name clashes between non-terminals of the two grammars, actual mixin grammars use parametrized imports, so that all symbols are postfixed to make them uniquely named.

```
|[
   public class X {
     // ...
   }
]|
```

```
CompUnit |[
  public class X {
     // ...
  }
]|
```

```
|[
   package org.generated;
   public class X {
     // ...
   }
]|
```

Fig. 5. Quotations of a Java compilation unit. From left to right: an ambiguous quotation, a quotation that is disambiguated by tagging, and a quotation that is already unambiguous without tagging.

category enables distinction between non-terminals with the same name that are defined in different object languages. For tagged anti-quotations we distinguish `FromMetaExpr` and `FromMetaExprTagged`.

Figure 5 shows an illustration of both untagged and tagged quotations. The quotation on the left is ambiguous, as it can represent a single class, a class declaration statement, or a compilation unit, each represented differently in the abstract syntax. The quotation in the middle makes it *explicit* that the intended non-terminal is a compilation unit, resolving the ambiguity. The quotation on the right is already unambiguous, because only complete Java compilation units can include a package declaration, and does not have to be explicitly disambiguated. Similar to quotations, anti-quotations can be ambiguous if they can represent multiple possible non-terminals within the context of a quotation.

3 Interactive Disambiguation of Concrete Object Syntax

In this section we describe how ambiguities in concrete object syntax can be interactively resolved by analyzing ambiguities and providing quick fix suggestions. We describe different classes of ambiguities and give an algorithm for automatically determining disambiguation suggestions for a given parse forest and grammar.

3.1 Classes of Ambiguities

At the grammar level, there are a number of different classes of ambiguities. In this paper we focus on ambiguities in quotations and anti-quotations. These ambiguities are inherent to the use of mixin grammars, as languages are woven together and fragments must be parsed with limited syntactic context. Disambiguation with tags or types can resolve these ambiguities. Other forms of ambiguities can be caused by the meta or object language, such as with the C language that notoriously overloads the $*$ operator for multiplication and pointer dereference. Such ambiguities must be retained if they are part of the object language design, otherwise they should be resolved at the grammar level. Ambiguities can also arise by the combination of the two languages if the syntax between the meta and object language overlap. These cannot always be resolved by type-based disambiguation [13], but can only be avoided by carefully selecting sensible quoting symbols in such a way that they do not overlap with the meta language and object language. Ideally, the symbols are chosen to be aesthetically pleasing characters or character combinations that never occur in either the object or meta language.

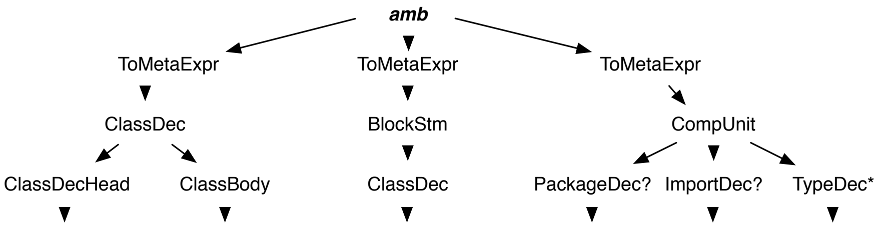


Fig. 6. The parse forest for the quotation `|[ public class X {} ]|`

Quotations are ambiguous when they can be parsed in more than one way, according to one or more object languages. To illustrate interactive disambiguation suggestions for quotations, consider again the untagged quotation in the left-hand side of Figure 5. Recall that this fragment could represent a single class, a class declaration statement, or a compilation unit. Using a generalized parser such as SGLR, the parser constructs a parse forest that branches at the point such an ambiguity, containing all possible subtrees for the ambiguous expression. Figure 6 illustrates the parse forest for our example, with at the top a special "amb" tree node that has the three possible interpretations as its children. The gist of our technique is to analyze the different possible parse trees, and have the developer select which alternative they intended.

In the mixin grammar for the embedded Java language (shown in Figure 3), there are three untagged productions that produce the three interpretations of our example. The tagged productions of Figure 4 parse the same object language non-terminal, but include distinguishing tags. These tags can be used to disambiguate the example: when one of the tags `ClassDec`, `BlockStm`, or `CompUnit` is added, there is only one possible interpretation of the quotation. By providing quick fix suggestions that automatically insert one of these three tags, meta-programmers can consider the three options and decide which is the interpretation they intended. In the event that the fragment could also be parsed using a different object language that happens to use the same tag names, the prefixed tags such as `Java:CompUnit` are proposed instead.

Anti-quotations can be disambiguated much like quotations. However, because they always occur in the context of a quotation, there is no need for language-prefixed quoting tags. For anti-quotations we also distinguish *local ambiguity*, where a single anti-quotation can be parsed in multiple ways, and *non-local ambiguity*, where a larger area of the quotation can be parsed in multiple ways. Non-local ambiguities arise as anti-quotations productions typically reduce to multiple possible non-terminals, whereas quotation productions typically reduce to only one, such as `Term` in Figure 3.

Figure 7 (left) shows a local ambiguity. The anti-quotation `$[`x`]` may be interpreted as an identifier or as the signature of the quoted class. The remainder of the quotation is unambiguous, making it trivial to identify the cause of the ambiguity in the parse forest.

Figure 7 (right) shows a non-local ambiguity. For this example, the entire body of the quotation can be interpreted in multiple ways: it can be either a class y with modifier x, or a package/import/type declaration x followed by a class y. For non-local ambiguities it is harder to identify the cause of the ambiguity, as the quotation expressions are no longer a direct subtree of the "amb" node as they are in Figure 6.

```
CompUnit |[
  class $[x] {
    // ...
  }
]|
```

```
CompUnit |[
  $[x] class Y {
    // ...
  }
]|
```

Fig. 7. Example of a local ambiguity (left) and a non-local ambiguity (right). In the first only the anti-quotation `$[x]` is ambiguous, in the other the entire contents of the quotation is ambiguous.

3.2 Automatic Disambiguation Suggestions

In this subsection we describe an algorithm to automatically collect disambiguation suggestions. We implemented the algorithm using Stratego and published the implementation and source code online at [1]. A prototype currently integrates into the Spoofax language workbench [10].

Figure 8 shows pseudocode for the disambiguation suggestions algorithm. At the top, the `CollectSuggestionsTop` function is the main entry point, which gets the parse forest and grammar as its input and returns a set of disambiguation suggestions as its output. For each outermost ambiguous subtree amb, it uses the `CollectSuggestions` function to find local disambiguation suggestions.

The `CollectSuggestions` function produces a set of disambiguation suggestions by inspecting each subtree of the amb tree node (line 2). For each branch, it searches for the outermost meta-expressions that are not yet completely tagged (line 3). For each meta-expression it determines the production $prod$ that was used to parse it (line 4), and its left-hand and right-hand side non-terminals (line 5, 6). For SGLR parse trees, the production is encoded directly in the tree node, allowing it to be easily extracted. Only meta-expressions that are a direct child of amb (local ambiguities) and meta-expression subtrees that do not have any tag (non-local ambiguities) are considered for suggestions (line 7).[2] For the selected meta-expressions, a set of possible disambiguation suggestions is collected (line 8). These suggestions take the form of tagged meta-expression productions (line 9) that contain the same left-hand and right-hand side non-terminals as the production $prod$ (line 10). Of course, we only include quotation productions if the current expression is a quotation, and anti-quotation productions if it is an anti-quotation (line 11). After all corresponding suggestions are collected, the complete set is filtered using the `FilterAmbiguousSuggestions` function (line 13).

The `FilterAmbiguousSuggestions` function filters out any suggestions that are ambiguous with respect to each other. This is useful if two object languages both match a meta-expression *and* they use the same quotation tag `X`. In those cases, inserting the tag `X` would not resolve the ambiguity, and a tag with a language prefix of the form `Lang:X` must be proposed instead. For suggestions with quoting symbols q_1,q_2 (line 3, 4), the function only returns those for which there is no other suggestion with the same quoting symbols (line 5, 6).

[2] A special case is the `ToMetaExprTagged1` constructor, used for tagged quotations without a language prefix. Suggestions are only provided for *local* ambiguities with this constructor.

```
CollectSuggestionsTop($tree, grammar$)
1  $results \leftarrow \{\}$
2  foreach outermost subtree $amb$ in $tree$ where $amb$ has the form amb(...)
3     $results \leftarrow results \cup$ CollectSuggestions($amb, grammar$)
4  return $results$

CollectSuggestions($amb, grammar$)
 1  $results \leftarrow \{\}$
 2  foreach child subtree $branch$ in $amb$
 3     foreach outermost subtree $expr$ in $amb$ where IsMetaExprTaggable($expr$)
 4           $prod \leftarrow$ the production for $expr$
 5           $lsort \leftarrow$ the non-terminal at left-hand side of $prod$
 6           $rsort \leftarrow$ the non-terminal at right-hand side of $prod$
 7           if $expr = branch \lor \neg$IsMetaExprTagged($prod$) then
 8              $results \leftarrow results \cup \{\ (expr, prod') \mid prod' \in$ productions of $grammar$
 9                                                 $\land$ IsMetaExprTagged($prod'$)
10                                                 $\land\ prod'$ has the form ($q_1$ $lsort$ $q_2$ -> $rsort$)
11                                                 $\land\ prod'$ and $prod$ have the same construc-
12                                                    tor prefix To or From $\}$
13  return FilterAmbiguousSuggestions($results$)

IsMetaExprTaggable($t$)
1  if $t$ has FromMetaExpr, ToMetaExpr, or ToMetaExprTagged1 constructor
2  then return true
3  else return false

IsMetaExprTagged($p$)
1  if $p$ has FromMetaExprTagged, ToMetaExprTagged1 or ToMetaExprTagged2 constructor
2  then return true
3  else return false

FilterAmbiguousSuggestions($suggestions$)
1  return $\{\ (prod, expr) \mid (prod, expr) \in suggestions$
2                              $\land\ prod$ has the form ($q_1$ $lsort$ $q_2$ -> $rsort$)
3                              $\land \neg\exists\ (expr', prod') \in suggestions$:
4                                    $prod'$ has the form ($q_1$ $lsort'$ $q_2$ -> $rsort'$) $\}$
```

Fig. 8. Pseudo-code for collecting suggested quotation symbols

3.3 Presentation of Suggestions

Interactive disambiguation is based on the notion of *quick fixes*, small program transformations that can be triggered by the developer in case of a code inconsistency or code smell. Quick fixes are non-intrusive: as developers write their program, errors or warnings are marked inline, but it is up to the developer to decide when and if to address the problems. For interactive disambiguation, quick fixes allow meta-programmers to write concrete syntax for expressions first, allowing the parser to decide whether or not it is ambiguous, proposing appropriate quick fixes as necessary.

The `CollectSuggestionsTop` function is executed each time the result of parsing the meta program is ambiguous. The quickfix menu is populated with the results, and any ambiguity can be addressed by adding the tag name or by inserting the type into the context of the quotation. In order to avoid spurious suggestions for multiple ambiguities, we only provides suggestions for the *outermost* expressions (`Collect-Suggestions`, line 8), allowing meta-programmers to incrementally fix any remaining ambiguities.

While we emphasize interactivity, it should be noted that the technique does not necessarily require an IDE. Quotation alternatives can also be displayed as part of the build process and used with generic text editors that may not interactively parse and analyze the meta language source code.

In our implementation we cache operations such as collecting productions from the grammar for efficiency, while in the algorithm described here we abstract from these optimizations. Experience with the prototype tells us that the performance overhead of the suggestions algorithm is very low, as it only does a depth-first traversal of each ambiguity and a few hash table lookups.

4 Discussion and Conclusions

In this paper we combined interactive and tag-based disambiguation to reduce quotation noise in meta-programs with concrete syntax. Developers only need to quote where absolutely necessary, and are interactively helped to introduce appropriate quotation symbols where required.

Interactive disambiguation can also be combined with type-based disambiguation, assisting in cases where type-based disambiguation is inadequate, as multiple type-based interpretations are type correct. These cases particularly arise when combining the technique with type inference, as seen with Meta-AspectJ [8], or when forgoing quoting symbols that distinguish between the meta and the object language, as observed by Vinju [13]. Both works propose heuristics as a solution in these cases. Interactive disambiguation can let the programmer interactively, and thus more predictably, resolve such ambiguities statically. Alternatively it can assist when programs are not yet type consistent, providing suggestions for inserting type declarations or type casts. Stratego is largely untyped, ruling out type-based disambiguation for our present prototype. A typed variant of Stratego [11] might be a suitable testbed for experiments combining interactive disambiguation and type inference. On the dynamic side, we have had promising experimental results using *runtime disambiguation*, where the decision of the correct interpretation of a meta-expression is delayed until run-time, when the actual values of meta-level expressions are known. Based on a static analysis of the meta-program, it is possible to determine which quotations can safely be disambiguated at runtime.

We performed a preliminary evaluation of our approach using existing source files that embed Java in Stratego, from the Dryad Java compiler [9]. The sources use a total of 55 concrete syntax quotations of a wide variety of different Java language constructs. Most are small quotations, but a few contain complete compilation units, used for compilation and for unit testing. We stripped all existing disambiguation tags from the sources, and by following the interactive disambiguation suggestions, were able to successfully disambiguate the files. We then introduced WebDSL [16] as an additional object language as a form of an evolution scenario. This introduced new ambiguities, as some expressions such as `Expr |[ $[Expr:`x`] == $[Expr:`y`] ]|` would be a valid quotation for either language. Again, applying the quick fixes helped the transition and resolved the ambiguities by introducing language-prefixed tags.

We have found interactive disambiguation to be a practically useful technique, complementary approach to both tag-based [6], and type-based disambiguation [5,13], and independent of the meta and object language and their type system.

Acknowledgements. This research was supported by NWO/JACQUARD projects 612.063.512, *TFA: Transformations for Abstractions*, and 638.001.610, *MoDSE: Model-Driven Software Evolution*.

References

1. The interactive disambiguation project (2010), `http://strategoxt.org/Spoofax/InteractiveDisambiguation`
2. Arnoldus, J., Bijpost, J., van den Brand, M.: Repleo: a syntax-safe template engine. In: GPCE, pp. 25–32 (2007)
3. Batory, D., Lofaso, B., Smaragdakis, Y.: JTS: Tools for Implementing Domain-Specific Languages. In: Conference on Software Reuse, p. 143 (1998)
4. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A language and toolset for program transformation. SCP 72(1-2), 52–70 (2008)
5. Bravenboer, M., Vermaas, R., Vinju, J.J., Visser, E.: Generalized type-based disambiguation of meta programs with concrete object syntax. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 157–172. Springer, Heidelberg (2005)
6. Bravenboer, M., Visser, E.: Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In: OOPSLA, pp. 365–383 (2004)
7. Cordy, J.R., Halpern-Hamu, C.D., Promislow, E.: TXL: A rapid prototyping system for programming language dialects. Comp. Lang., Syst. & Struct. 16(1), 97–107 (1991)
8. Huang, S.S., Zook, D., Smaragdakis, Y.: Domain-specific languages and program generation with Meta-AspectJ. Transactions on Software Engineering Methodology 18(2) (2008)
9. Kats, L.C.L., Bravenboer, M., Visser, E.: Mixing source and bytecode: a case for compilation by normalization. In: OOPSLA, pp. 91–108 (2008)
10. Kats, L.C.L., Visser, E.: The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In: OOPSLA, pp. 444–463 (2010)
11. Lämmel, R.: Typed generic traversal with term rewriting strategies. Journal of Logic and Algebraic Programming 54(1), 1–64 (2003)
12. van Deursen, A., Heering, J., Klint, P. (eds.): Language Prototyping. An Algebraic Specification Approach. AMAST Series in Computing, vol. 5. World Scientific, Singapore (1996)
13. Vinju, J.J.: Type-driven automatic quotation of concrete object code in meta programs. In: Guelfi, N., Savidis, A. (eds.) RISE 2005. LNCS, vol. 3943, pp. 97–112. Springer, Heidelberg (2006)
14. Visser, E.: Syntax Definition for Language Prototyping. PhD thesis, University of Amsterdam (September 1997)
15. Visser, E.: Meta-programming with concrete object syntax. In: Batory, D., Blum, A., Taha, W. (eds.) GPCE 2002. LNCS, vol. 2487, pp. 299–315. Springer, Heidelberg (2002)
16. Visser, E.: WebDSL: A case study in domain-specific language engineering. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) Generative and Transformational Techniques in Software Engineering II. LNCS, vol. 5235, pp. 291–373. Springer, Heidelberg (2008)

Evaluating a Textual Feature Modelling Language: Four Industrial Case Studies

Arnaud Hubaux[1], Quentin Boucher[1], Herman Hartmann[3], Raphaël Michel[2], and Patrick Heymans[1]

[1] PReCISE Research Centre, Faculty of Computer Science, University of Namur, Belgium
{ahu,qbo,phe}@info.fundp.ac.be
[2] CETIC Research Centre, Belgium
raphael.michel@cetic.be
[3] Virage Logic, High Tech Campus, The Netherlands
herman.hartmann@viragelogic.com

Abstract. Feature models are commonly used in software product line engineering as a means to document variability. Since their introduction, feature models have been extended and formalised in various ways. The majority of these extensions are variants of the original tree-based graphical notation. But over time, textual dialects have also been proposed. The *textual variability language* (TVL) was proposed to combine the advantages of both graphical and textual notations. However, its benefits and limitations have not been empirically evaluated up to now. In this paper, we evaluate TVL with four cases from companies of different sizes and application domains. The study shows that practitioners can benefit from TVL. The participants appreciated the notation, the advantages of a textual language and considered the learning curve to be gentle. The study also reveals some limitations of the current version of TVL.

1 Introduction

Feature models (FMs) were introduced as part of the FODA (Feature Oriented Domain Analysis) method 20 years ago [1]. They are a graphical notation whose purpose is to document variability, most commonly in the context of software product line engineering (PLE) [2]. Since their introduction, FMs have been extended and formalised in various ways (e.g. [3,4]) and tool support has been progressively developed [5]. The majority of these extensions are variants of FODA's original tree-based graphical notation. Figure 1 shows an example of graphical tree-shaped FM that describes the variability of an eVoting component. The *and*-decomposition of the root feature (*Voting*) implies that all its sub-features have to be selected in all valid products. Similarly, the *or*-decomposition of the *Encoder* feature means that at least one of its child features has to be selected, and the *xor*-decomposition of the *Default VoteValue* feature means that one and only one child has to be selected. Cardinality-based decompositions can also be defined, like for *VoteValues* in the example. In this case, the decomposition type implies that at least two, and at most five sub-features of *VoteValues* have to be selected. Finally, two `<requires>` constraints impose that the feature corresponding to the default vote value (*Yes* or *No*) is part of the available vote values.

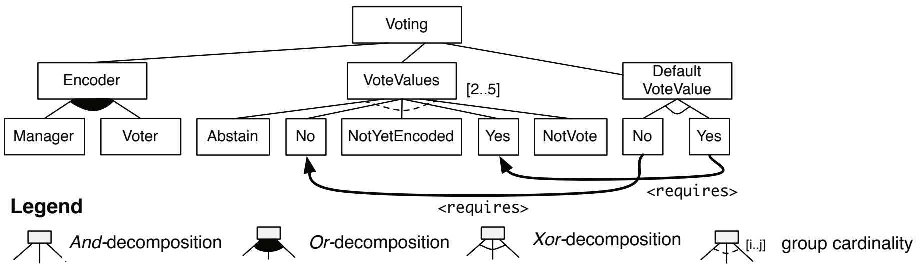


Fig. 1. FM of the PloneMeeting voting component

Over time, textual FM dialects have also been proposed [6,7,8,9], arguing that it is often difficult to navigate, edit and interpret large graphical FMs. The need for more expressiveness was a further motivation for textual FMs since adding constructs to a graphical language can quickly start harming its readability. Although advanced techniques have been suggested to improve the visualisation of graphical FMs (e.g. [10,11]), these techniques remain tightly bound to particular modelling tools and are hard to integrate in heterogeneous tool chains [12]. Finally, our experience shows that editing functionalities offered by such tools are usually pretty limited.

Based on these observations, we proposed TVL [8,13], a textual FM dialect geared towards software architects and engineers. Its main advantages are that (1) it does not require a dedicated editor—any text editor can fit—(2) its C-like syntax makes it both intuitive and familiar, and (3) it offers first-class support for modularity. However, TVL is meant to complement rather than replace graphical notations. It was conceived to help designers during variability modelling and does not compete, for instance, with graphical representations used during product configuration.

The problem is that empirical evidence showing the benefits and limitations of existing approaches, be they graphical or textual, is cruelly missing [8]. The goal of this paper is to collect evidence that demonstrates whether TVL is actually fit for practice, which is translated into the following research questions:

RQ1. *What are the benefits of TVL for modelling product line (PL) variability, as perceived by model designers?*
RQ2. *What are the PL variability modelling requirements that are not fulfilled by TVL?*

It is important to understand that the goal of this research is neither to compare TVL to other languages, nor to assess capabilities of TVL other than its ability to model variability, nor to compare graphical and textual approaches. Instead, this research aims at identifying the benefits and limitations of TVL.

To answer these research questions, we conducted a *controlled field experiment* following a *sequential explanatory strategy* [14]. It consists of a *quantitative data analysis* followed by a *qualitative* one. The quantitative analysis is meant to collect data while the qualitative analysis assists in explaining the outcomes of the quantitative analysis. Quantitative data on TVL is collected via evaluation forms based on a set of quality criteria inspired from the evaluation of programming languages. The main motivation for this is that TVL is in many respects similar to a declarative constraint programming language. TVL was evaluated by five participants working for four companies of different sizes (from 1 to 28 000 employees), domains (hardware and software) and business models (proprietary and free software). Furthermore, TVL was evaluated by participants with different backgrounds in modelling and programming languages. The interviews carried out during the qualitative analysis helped us (1) collect evidence that practitioners can benefit from TVL, (2) identify the types of stakeholders who can reap benefits from a language like TVL, and (3) elicit requirements that are not fulfilled by TVL.

The remainder of the paper is structured as follows. Section 2 looks into related work on feature-based variability modelling. Section 3 recalls the essence of TVL. Section 4 describes the research method and the four cases, whilst Section 5 presents the results of the quantitative and qualitative analyses. Section 6 analyses these results and Section 7 discusses the threats to validity.

2 Related Work

This section studies related work respectively dedicated to graphical and textual approaches to feature modelling.

2.1 Graphical Feature Models

Most common FM languages are graphical notations based on FODA which was introduced by Kang *et al.* [1] twenty years ago. Since this original proposal, several extensions have been proposed by various authors [15]. Most of these graphical notations are meant to be accessible to non-technical stakeholders. However, working with large-scale FMs can become challenging with such notations. Given that a FM is a tree on a two dimensional surface, there will inevitably be large physical distances between features, which makes it hard to navigate, search and interpret them. Several tools have been developed to help modellers [16,17,18,19]. Most of them use directory tree-like representations of FMs to reduce physical distances between some features and provide collapse/expand functionalities. More advanced user interfaces and visualisation techniques have also been proposed to attenuate the aforementioned deficiencies (e.g. [10,11]), but tools have their own drawbacks. First, building FMs can become time consuming as tools often require lots of clicks and drag-and-drops to create, place or edit elements. Second, they rely on software to visualise a model, meaning that without this software, like for instance on paper, blackboard or on a random computer, they will not work. Furthermore all those tools tend to have poor interoperability, which prevents effective collaboration. Besides all those considerations, some constructs like cross-tree constraints or attributes cannot be easily accommodated into those graphical representations.

2.2 Textual Feature Models

Various textual FM languages have been proposed for a number of purposes. Their claimed advantages over graphical notations are: they do not require dedicated modelling tools and well-established tools are available for text-based editing, transformation, versioning... Furthermore, textual information and textual models can be easily exchanged, for instance by email.

To the best of our knowledge, FDL [6] was the first textual FM language. It is the only such language to have a formal semantics. It also supports basic *requires* and *excludes* constraints and is arguably user friendly, but it does not include attributes, cardinality-based decompositions and other advanced constructs.

The AHEAD [7] and FeatureIDE [18] tools use the GUIDSL syntax [7], where FMs are represented through grammars. The syntax is aimed at the engineer and is thus easy to write, read and understand. However, it does not support decomposition cardinalities, attributes, hierarchical decomposition of FMs and has no formal semantics.

The SXFM file format is used by SPLOT [20] and 4WhatReason [21]. While XML is used for metadata, FMs are entirely text-based. Its advantage over GUIDSL is that it makes the tree structure of the FM explicit through indentation. However, except for the hierarchy, it has the same deficiencies as GUIDSL.

The VSL file format of the CVM framework [22,23] supports many constructs. Attributes, however, cannot be used in constraints. The Feature Modelling Plugin [16] as well as the FAMA framework [24] use XML-based file formats to encode FMs. Tags make them hard to read and write by engineers. Furthermore, none of them proposes a formal semantics. The Concept Modelling Language (CML) [9] has been recently proposed but to the best of our knowledge is still a prototype and is not yet fully defined or implemented.

3 TVL

Starting from the observation that graphical notations are not always convenient and that existing textual notations have limited expressiveness, formality and/or readability, we proposed TVL [8,13], a textual alternative targeted to software architects and engineers. For conciseness, we can only recall here the basic principles of the language. More details about its syntax, semantics and reference implementation can be found in [13].

The following model will be used to illustrate some TVL constructs. It is a translation of the graphical model presented in Figure 1, which is an excerpt of the complete FM we built for PloneMeeting, one of the case studies. It captures the variability in the voting system that governs the discussion of meeting items[1]. Note that the default vote value is specified here as an attribute rather than a feature.

```
root Voting { // define the root feature
 enum defaultVoteValue in {yes, no}; //attribute is either yes or no
 (defaultVoteValue == yes) -> Yes; //yes requires Yes in VoteValues
 (defaultVoteValue == no) -> No; // no requires No in VoteValues
 group allOf { // and-decomposition
     Encoder { group someOf {manager, voter} },
     VoteValues group [2..*] { // <2..5> cardinality
         Yes,
         No,
         Abstain
         NotYetEncoded,
         NotVote,
     }
 }
}
```

[1] The complete model is available at `http://www.info.fundp.ac.be/~acs/tvl`

TVL can represent FMs that are either trees or directed acyclic graphs. The language supports standard decomposition operators [1,3]: *or*- , *xor*-, and *and*-decompositions. For example, the *and*-decomposition of the `Voting` is represented from lines `05` to `12` (`group allOf {...}`). The *xor*-decomposition of the *Encoder* is represented at line `06` (`group oneOf {...}`). Generic cardinality-based decompositions [25] can also be defined using a similar syntax (see line `07`). Five different types of feature attributes [26] are supported: integer (`int`), real (`real`), Boolean (`bool`), structure (`struct`) and enumeration (`enum`). The domain of an attribute can be restricted to a predefined set of values using the `in` keyword. For instance, the set of available values of the enumerated attribute `defaultVoteValue` is restricted to `yes` and `no` (see line `02`). Attributes can also be assigned fixed or calculated values using the `is` keyword. Furthermore, the value of an attribute can differ when the containing feature is selected (`ifIn:` keyword) or not selected (`ifOut:` keyword). Several standard operators are available for calculated attributes (e.g. arithmetic operations). Their value can also be computed using aggregation functions over lists of attributes. Calculated attributes are not illustrated in the example.

In TVL, constraints are attached to features. They are Boolean expressions that can be added to the body of a feature. The same guards as for attributes are available for constraints. They allow to enable (resp. disable) a constraint depending on the selection (resp. non-selection) of its containing feature. Line `05` is an example of (unguarded) constraint where the assignment of the `yes` value to the `defaultVoteValue` attribute requires the selection of the `Yes` feature.

TVL offers several mechanisms to reduce the size of models and modularise them. We only touch upon some of them here and do not illustrate them in the example. First, custom types can be defined and then used in the FM. This allows to factor out recurring types. It is also possible to define structured types to group attributes that are logically linked. Secondly, TVL supports constants that can be used inside expressions or cardinalities. Thirdly, `include` statements can be used to add elements (e.g. features or attributes) defined in an external file anywhere in the code. Modellers can thus structure the FM according to their preferences. The sole restriction is that the hierarchy of a feature can only be defined at one place (i.e. there is only one group structure for each feature). Finally, features can have the same name provided they are not siblings. Qualified feature names must be used to reference features whose name is not unique. Relative names like `root`, `this` and `parent` are also available to modellers.

4 Research Method

This section describes the settings of the evaluation, its goals and the four cases along with the profiles of the companies and participants involved in the study. The section ends with a description of the experiment's protocol.

4.1 Objectives

The overall objective of this paper is to evaluate the ability of TVL to model the variability of a PL as perceived by software engineers. The criteria that we use to measure the quality of TVL are inspired and adapted from [27,28]. Originally, these criteria were defined to evaluate the quality of programming languages. We have selected programming language criteria because TVL resembles a declarative constraint programming language whose constructs have been tailored for variability modelling. Finally, TVL should ultimately be integrated in development environments like eclipse or advanced text editors like emacs or vim. TVL is thus likely to be assimilated to a programming language by developers. We outline the quality criteria relevant to our study below.

Quality criteria adapted from [27,28].	
C1 Clarity of notation	The meaning of constructs should be unambiguous and easy to read for non-experts.
C2 Simplicity of notation	The number of different concepts should be minimum. The rules for their combinations should be as simple and regular as possible.
C3 Conciseness of notation	The constructs should not be unnecessarily verbose.
C4 Modularisation	The language should support the decomposition into several modules.
C5 Expressiveness	The concepts covered by the language should be sufficient to express the problems it addresses. Proper syntactic sugar should also be provided to avoid convoluted expressions.
C6 Ease and cost of model portability	The language and tool support should be platform independent.
C7 Ease and cost of model creation	The elaboration of a model should not be overly human resource-expensive.
C8 Ease and cost of model translation	The language should be reasonably easy to translate into other languages/formats.
C9 Learning experience	The learning curve of the language should be reasonable for the modeller.

No specific hypothesis was made, except that the participants had the appropriate expertise to answer our questions.

4.2 Cases

The evaluation of TVL was carried out with five participants coming from four distinct companies working in different fields (two in computer hardware manufacture, one in meeting management and one in document management). Table 2 summarises the profiles of the five participants involved in the evaluation as well as the company and project they work for. For each participant, we indicate his position, years of experience in software engineering, his fields of expertise, the modelling and programming languages he used for the last 5 years, his experience with PLE and FMs, and the number of years he actively worked on the selected project. For the experience with languages, PLE and FMs, we also mention the frequency of use, i.e. *intensive/regular/casual/evaluation*. By evaluation, we mean that the language or concept is currently being evaluated in the company. The four cases are described below as well as the motivations of the participants to use TVL.

Table 2. Profiles of the five participants

Criteria	**PloneMeeting**	**PRISMAprepare**	**CPU calculation**	**OSGeneric**	
Company	*GeezTeem*	*OSL Namur S.A.*	*NXP Semiconductors*	*Virage Logic*	
#Employees	1	70	28 000 (worldwide)	700 (worldwide)	
Location	Belgium	Belgium	The Netherlands	The Netherlands	
Type of software	Open source	Proprietary	Proprietary	Proprietary	
Project kickoff date	January 2007	May 2008	June 2009	2004	
Project stage reached	Production	Production	Development	Production	
Project version	1.7 build 564	4.2.1	July 6, 2009	September 30, 2009	
Number of features	193	152	19	60	
Position	Freelance	Product Line Manager	Senior Scientist	Senior Software Architect	Development Manager
Years of experience in SE	12+ years	31+ years	20+ years	15+ years	20+ years
Fields of expertise	WA, CMS	PM	PM, SPI, SR	McS, RPC, RT	ES, RT
Modelling languages	UML (regular)	None	UML (casual), DSCT (evaluation)	UML (casual)	UML (regular)
Programming languages	Python (intensive), Javascript (regular)	C++ (casual)	C (casual), Prolog (regular), Java (evaluation)	C (intensive)	C (casual), C++ (casual), Visual Basic (casual), Python (casual)
Experience with PLE	2 years (evaluation)	1 year (casual)	3 years (intensive)	4 years (regular)	4 years (intensive)
Experience with FDs	2 years (evaluation)	1 year (casual)	3 years (intensive)	4 years (casual)	4 years (intensive)
Project participation	3 years	2 years	1 year	6 years	2 years

Legend

CMS - Content Management System, **DSCT** - Domain Specific Components Technologies, **ES** - Embedded System, **McS** - Multi-core System,
PM - Project Management, **RPC** - Remote Procedure Call, **RT** - Real Time, **SE** - Software Engineering, **SPI** - Software Process Improvement,
SR - Software Reliability, **WA** - Web Applications.

PloneMeeting

Description. PloneGov [29] is an international Open Source (OS) initiative coordinating the development of eGovernment web applications. PloneGov gathers hundreds of public organizations worldwide. This context yields a significant diversity, which is the source of ubiquitous variability in the applications [30,31,32]. We focus here on PloneMeeting, PloneGov's meeting management project developed by GeezTeem. PloneMeeting is built on top of Plone, a portal and content management system (CMS) written in Python.

The current version of PloneMeeting extensively uses UML for code generation. However, the developer is not satisfied by the limited editing functionalities and flexibility offered by UML tools. To eliminate graphical UML models, he developed *appy.gen*[2]. *appy.gen* allows to encode class and state diagrams, display parameters, portlet creation, and consistency rules in pure Python and enables the automated generation of full-blown Plone applications.

PloneMeeting is currently being re-engineered with *appy.gen*. A major challenge is to extend *appy.gen* to explicitly capture variation points and provide systematic variability management. We collaborate with the developer to specify the FM of PloneMeeting.

Motivation. The initial motivation of the developer to engage in the evaluation was to assess the opportunity of using FMs for code generation. He has not used FMs to that end so far because, in his words, *"graphical editing functionalities (typically point-and-click) offered by feature modelling tools are cumbersome and counterproductive"*. The textual representation of FMs is therefore more in line with his development practice than their graphical counterpart.

PRISMAprepare

Description. Océ Software Laboratories S.A. (OSL) [33], is a company specialized in document management for professional printers. One of their main PLs is *Océ PRISMA*, a family of products covering the creation, submission and delivery of printing jobs. Our collaboration focuses on one sub-line called PRISMAprepare, an all-in-one tool that guides document preparation and whose main features are the configuration of printing options and the preview of documents.

In the current version of PRISMAprepare, mismatches between the preview and the actual output can occur, and, in rare cases, documents may not be printable on the selected printer. The reason is that some incompatibilities between document options and the selected printer can go undetected. For example, a prepared document can require to staple all sheets together while the target printer cannot staple more than 20 pages together, or does not support stapling at all. The root cause is that only some constraints imposed by the printers are implemented in the source code, mostly for time and complexity reasons.

Consequently, OSL decided to enhance its PL by automatically generating configuration constraints from printing description files. The objective is twofold: (1)

[2] Available online at `http://appyframework.org/gen.html`

build complete constraint sets and (2) avoid the manual maintenance of cumbersome constraints. So far, we have designed a first FM of the variability of the printing properties.

Motivation. Feedback was provided by the *product line manager* of PRISMAprepare. OSL is currently evaluating different modelling alternatives to express the variability of its new PL and generate the configuration GUI. The evaluation of TVL is thus part of their exploratory study of FM languages. Additionally, most of the software engineers dealing with FMs are developers, hence their interest for a textual language.

CPU calculation

Description. NXP Semiconductors [34] is an international provider of Integrated Circuits (IC). ICs are used in a wide range of applications like automotive, infotainment and navigation systems. They are the fundamental pieces of hardware that enable data processing like video or audio streams. ICs typically embed several components like CPU, memory, input and output ports, which all impose constraints on their possible valid combinations.

This study focuses on the FM that models the variability of a video processing unit and study the impact it has on the CPU load. The FM, which is still under development, is meant to be fed to a software configurator. Thereby, it will support the customer during the selection of features while ensuring that no hardware constraint is violated (e.g. excessive clock speed required by the features). The FM also allows the user to strike an optimal price/performance balance, where the price/performance ratio is computed from attributes attached to features and monitored within the configuration tool.

Motivation. The evaluation was performed by the developer who originally participated in the creation of the FM with a commercial tool, before and independently from this experiment. Prolog was used for defining the constraints. The major problem is that the time needed to implement the calculation over attributes was deemed excessive compared to the time needed to design the whole FM. This lead the company to consider TVL as an alternative. Furthermore, the developer's knowledge of a declarative language like Prolog motivated him to test a textual constraint modelling language.

OSGeneric

Description. Virage Logic [35] is a supplier of configurable hardware and software to a broad variety of customers such as the Dolby Laboratories, Microsoft and AMD. Its variability-intensive products allows its customers to create a specific variant for the manufacturing of highly tailorable systems on chip (SoC). OSGeneric (Operating System Generic) is a PL of operating systems used on SoCs. The produced operating systems can include both proprietary and free software. Each SoC can embed a large variety of hardware and contain several processors from different manufacturers.

Motivation. The evaluation was performed by two participants: the lead software architect of OSGeneric and the software development manager. Their PL is currently modelled with a commercial tool. The participants are now considering other variability modelling techniques. Their motivation for evaluating TVL lies in using a language that (1) is more suited for engineers with a C/C++ background, (2) has a lower learning curve than the commercial tool and (3) makes use of standard editors.

4.3 Experiment Protocol

In this experiment, TVL was evaluated through interviews with the five participants of the four companies. Interviews were conducted independently from each other, except for Virage Logic where the two participants were interviewed together. Two researchers were in charge of the interviews, the synthesis of the results and their analysis. For each interview, they followed the protocol presented in Figure 2.

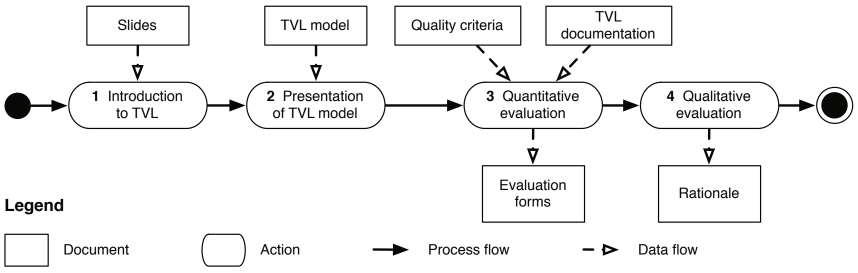


Fig. 2. Interview protocol

The protocol starts with a short introduction to TVL (*circa* 20 minutes) that aims at giving the participants an overview of the language. At this stage, the participants are not exposed to details of the language. The goal of the second step is to provide the participants with a real TVL model. To keep the effort of the participants moderate, the appointed researchers designed, for each company and prior to the interviews, TVL models that respectively correspond to the configuration menus of PloneMeeting and PRIMSAprepare, and the FMs of the CPU calculation and OSGeneric. The presentation of the TVL model was limited to 30 minutes to keep the participants focused on the understanding of the model and avoid untimely discussions about the quality of the language. During that step, the participants are exposed to more details of the language and discover how their PL can be modelled using TVL. Alternative design decisions are also discussed to demonstrate the expressiveness of TVL.

During the third step, the participants fill out the evaluation form presented in Table 3. The evaluation scale proposed to the participants is: ⊞ the participant is strongly satisfied; **+** the participant is rather satisfied; ○ the participant is neither satisfied nor dissatisfied; **-** the participant is rather dissatisfied; ⊟ the participant is completely dissatisfied; N/A the participant is not able to evaluate the criterion.

Table 3. Results of the evaluation of TVL

Criterion		**PloneMeeting**	**PRISMAprepare**	**CPU calculation**	**OSGeneric**	
C1	Clarity of notation	+	+	○	⊞	+
C2	Simplicity of notation	+	⊞	+	⊞	+
C3	Conciseness of notation	⊞	⊞	+	⊞	⊞
C4	Modularisation	○	+	+	+	+
C5	Expressiveness	-	○	⊞	⊞	⊞
C6	Ease and cost of model portability	+	⊞	⊞	⊞	⊞
C7	Ease and cost of model creation	+	+	⊞	○	○
C8	Ease and cost of model translation	⊞	⊞	+	○	+
C9	Learning experience	+	⊞	+	+	+

The results of the evaluation are then discussed during the fourth step. The qualitative information collected during this last phase is obtained by asking, for each criteria, the rationale that lead the participant to give his mark. On average, these two last steps lasted two hours in total.

5 Results

Table 3 synthesises the evaluation of TVL performed by the participants of GeezTeem, OSL, NXP and Virage Logic. Note that we kept the evaluations of the two Virage Logic participants separate, has indicated by the two columns under OSGeneric.

To facilitate the explanation, we group the criterion into five categories: *notation*, *modularisation*, *expressiveness*, *ease and cost*, and *learning experience*. Note that the collaborations with OSL, NXP and VirageLogic are protected by non-disclosure agreements. Therefore, specific details of the models are not disclosed.

Notation [C1-C3]. The participants unanimously appreciated the notation and the advantages of text in facilitating editing (creating, modifying and copy/pasting model elements). The NXP and VirageLogic participants liked the compactness of attributes and constraints and the fact that attributes were explicitly part of the language rather than an add-on to a graphical notation.

The GeezTeem participant appreciated the ability of the language to express constraints very concisely. He reports that *appy.gen*, his website generator, offers two major ways of specifying constraints. First, guards can be used to make the value of an attribute depend on the value of another attribute. Secondly, Python methods can be used to express arbitrary constraints. These mechanisms can rapidly lead to convoluted constraints that are hard to maintain and understand. Additionally, developers struggle to maintain these constraints across web pages. The participant reports that at least 90% of the constraints (both within and across pages) implemented in classical *appy.gen* applications could be more efficiently expressed in TVL.

The OSL participant was particularly satisfied to see that TVL is not based on XML. He reported that their previous attempts to create XML-based languages were not very satisfactory because of the difficulty to write, read and maintain them. He also reported that the model represented in the language is much more compact than anything he could have produced with existing graphical representations.

The NXP participant was concerned about the scalability of the nested structure, i.e. the tree-shaped model, offered by TVL. He also reports that people used to graphical notations who already know FMs might prefer classical decomposition operators (*and*, *or*, *xor*) rather than their TVL counterparts (`allOf`, `someOf`, `oneOf`). Finally, the participants from NXP and Virage Logic were confused by the fact that the `->` symbol can always replace `requires` but not the other way around. In their opinion, a language should not offer more than one means to express the same thing.

One of the Virage Logic participants reports that attributes might be hard to discern in large models. He suggested to declare them in an Interface Description Language (IDL) style by prefixing the attribute declaration with the `attribute` keyword.

Modularisation [C4]. The ability to define a feature at one place and extend it further in the code was seen as an undeniable advantage as it allows to distribute the FM among developers. The Virage Logic participants both discussed the difference between the TVL `include` mechanism and an *import* mechanism that would allow to specify exactly what parts of an external TVL model can be imported but also what parts of a model can be exported. In their opinion, it would improve FM modularisation and module reuse since developers are already used to import mechanisms.

Apart from the `include` mechanism, TVL does not support *model* specialisation and abstraction (as opposed to constructs, e.g. class and specialisation). In contrast, the developer of *appy.gen* considers them as fundamental. Their absence is one of the reasons that lead them to drop UML tools. Along the same lines, the OSL participant argued that the `include` should be augmented to allow *macro* definitions. By *macro*, the participant meant parametrized models similar to parametrized types, e.g. Java generics. A typical use case of that mechanism would be to handle common variability modelling patterns.

Expressiveness [C5]. The GeezTeem participant expressed that TVL is sufficiently expressive to model variability in most cases. However, he identified several constructs missed by TVL that would be needed to model PloneMeeting. First, TVL does not offer *validators*. In his terms, a validator is a general-purpose constraint mechanism that can constrain the formatting of a field. For instance, validators are used to specify the elements that populate a select list, to check that an email address is properly formatted or that a string is not typed in where the system expects an integer. Secondly, he makes intensive use of the specialisation and abstraction mechanisms available in Python, which have no TVL equivalents. These mechanisms are typically used to refine already existing variation points (e.g. add an attribute to a meeting item) or to specify abstract variation points that have to be instantiated and extended when generating the configuration menu (e.g. an abstract meeting attendee profile is built and it has to be instantiated before being available under the *Encoder* feature in Figure 1). Thirdly, multiplicities are used to specify the number of instances, i.e. clones, of a given element. Cloning is a fundamental aspect of *appy.gen* as many elements can be cloned and configured differently in Plone applications. This corresponds to *feature* cardinalities (as opposed to *group* cardinalities), which have already been introduced in FM [3], but are currently not supported in TVL. Besides offering more attributes types, *appy.gen* also allows developers to add parameters to attributes, e.g., to specify whether a field can be edited or requires specific read/write permissions. Type parameters are mandatory in *appy.gen* to support complete code generation. Finally, in order to be able to display web pages in different languages, *i18n* labels are attached to elements. *i18n* stands for internationalisation and is part of Plone's built-in translation management service. Translations are stored in key/value pairs. A key is a label in the code identifying a translatable string; the value is its translation. For instance, the `meeting_item_i18n` element will be mapped to *Meeting Item* (English) and *Point de discussion* (French). In most cases, several labels are attached to an element (e.g. a human-readable name and a description).

The OSL participant also pointed out some missing constructs in TVL. First, default values which are useful in their projects for things like page orientation or paper dimensions. Secondly, feature cloning is missing. In PRISMAprepare, a document is normally composed of multiple *sheets*, where sheets can be configured differently and independently from one another. Thirdly, optionality of attributes should be available. For instance, in TVL, the *binding margin* of a page was specified as an attribute determining its size. If the document does not have to be bound, the *binding margin* attribute should not be available for selection.

The NXP and VirageLogic participants also recognized that feature cloning and default features were missing in the language. Additionally, they miss the specification of error, warning and information messages directly within the TVL model. These messages are not simple comments attached to features but rather have to be considered as guidance provided to the user that is based on the current state of the configuration. For instance, in the NXP case, if the selected video codec consumes most of the CPU resources, the configurator should issue a warning advising the user to select another CPU or select a codec that is less resource-demanding. Since they are a needed input for a configurator, they argued that a corresponding construct should exist in TVL.

Ease and cost [C6-C8]. The OSL participant reports that improvements in terms of *model creation* are, in his words, very impressive compared to the graphical notation that was used initially [3]. And since TVL is formally defined, he does not foresee major obstacles to its translation into other formalisms.

The NXP and VirageLogic participants report that, no matter how good the language is, the process of model creation is intrinsically very complex. This means that the cost of model creation is always high for real models. Nevertheless, they observed that the mechanisms offered by TVL facilitate the transition from variability elicitation to a formal specification, hence the neutral score.

Learning experience [C9]. All the participants agreed that the learning curve for software engineers with a good knowledge of programming languages was rather gentle.

Software engineers who use only graphical models might need a little more time to feel comfortable with the syntax. In fact, the NXP and VirageLogic participants believe that people in their teams well versed in programming languages would give a ⊞ whereas those used to modelling languages would give a ○, hence their average + score.

6 Findings

This sections builds upon the analysis by looking at the results from three different angles: (1) the constructs missed by TVL, (2) the impact of stakeholder profiles on the use of TVL, and (3) the tool support that is provided and still to provide.

6.1 Language Constructs

The analysis revealed that extensions to the catalogue of constructs provided by TVL would be appreciated by the participants. We summarise below those that neither make the language more complex nor introduce extraneous information (such as, e.g., behavioural specifications) into the language.

Attribute cardinality. Traditionally, the cardinality of an attribute is assumed to be $\langle 1..1 \rangle$ (`oneOf`), i.e. one and only one element in its domain can be selected. However, the translation of select lists in PloneMeeting would have required enumerations with cardinalities. For instance, in Figure 3, the vote encoders would typically be captured in a select list, which should be translated into an enumeration with cardinality $\langle 1..2 \rangle$. The absence of cardinality for enumerations forced us to translate them has features. Yet, these select lists typically allow multiple selections, i.e. they require a cardinality like $\langle 1..n \rangle$ (`someOf`). Additionally, optional attributes, like the binding margin, would require a $\langle 0..1 \rangle$ (`opt`) cardinality. Technically, arbitrary cardinalities for attributes are a simple extension of the decomposition operators defined for features. Their addition to TVL will thus be straightforward.

Cloning. All the participants expressed a need for cloning in FMs. They have not been introduced in TVL because of the theoretical problems they yield, *viz.* reasoning about potentially infinite configuration spaces and managing clone-specific constraints. Feature cardinalities will thus be proposed in TVL when all the reasoning issues implied by cloning will be solved. This is work in progress.

Default values. The main advantage of default values is to speed up product configuration by pre-defining values (e.g. the default page orientation). The participant from OSL argued that, if their applications were to include a TVL-based configuration engine, then TVL should contain default values. This would avoid having default values scattered in the source code, thereby limiting the maintenance effort.

Extended type set. Far more types are needed in PloneMeeting than TVL offers. Our experience and discussions with the developers show that only some of them should be built in the language. For this reason, only the `String`, `Date` and `File` types will be added to TVL.

Import mechanism. In addition to the `include` mechanism, the participants requested a more fine grained import mechanism. This will require to add scoping to TVL.

Labels and messages. In the PloneMeeting case, the participant showed that labels have to be attached to features to provide the user with human-readable feature names and description fields, and also to propose multilingual content.

Both the NXP and VirageLogic participants use messages to guide the configuration of their products. Constraints on features determine the messages that are displayed based on the current configuration status.

Specialisation. In two cases, specialisation mechanisms appeared to be desirable constructs. They would typically allow to overload or override an existing FM, for example, by adding or removing elements from previously defined `enum`, `struct` and feature groups, or by refining cardinalities, e.g. from $\langle 0..*\rangle$ to $\langle 2..3\rangle$.

The extensions that could have been added to TVL but that, we thought, would make it deviate too much from its purpose are the following.

Abstraction. Although particularly useful for programming languages (e.g. abstract classes in object-oriented programming), we do not see the interest of using abstract FMs. FMs are, by definition, not meant to be instantiated but to be configured.

Method definition. TVL provides a static description of the variability of a PL. Methods, even declarative, would embed behavioural specifications within the language. This is neither an intended purpose nor a desired property of the language. Any method manipulating the FM should be defined externally.

Typed parameters. In PloneMeeting, typed parameters are extensively used. However, many of these parameters are very technical and case-specific, e.g. editable or searchable fields. Unless we come across more use cases, we will not add parameters to attributes. For cases like PloneMeeting, these parameters can be encoded with the `data` construct.

6.2 Stakeholder Profiles

As shown in Table 2, the participants had fairly different profiles. Our population consists of two developers, one designer and two project managers. Their experience with PLs and FMs also differ. Two participants are intensive users, one is a regular and the other two are still in the transition phase, i.e. moving from traditional to PL engineering.

Interestingly, these profiles did not have a noticeable influence on the marks given to the notation (C1-C3), ease and cost (C6-C8), and learning experience (C9). They all preferred and attribute grammar-like syntax to a markup-based language like XML, usually considered too verbose, difficult to read and tricky to edit. Furthermore, the C-like syntax was deemed to preserve many programming habits—like code layout, the development environment, etc.

Deviations appear at the level of modularisation (C4) and expressiveness (C5). One way to interpret it is that OSL and PloneMeeting are still in the transition phase. This means that they are not yet confronted to variability modelling tools in their daily work. They are more familiar with traditional modelling languages like UML and programming languages like C++ and Python. Compared to these languages, FMs, and TVL in particular, are more specific and thus far less expressive. Furthermore, in the PloneMeeting case, the participant developed its own all-in-one website configuration and generation tool, which embeds a domain specific language for statechart and class diagram creation.

These observations lead us to conclude that:

stakeholder profiles do no impact the evaluation of the notation. Developers clearly appreciated the textual alternative proposed by TVL. The ease and efficiency of use are the main reasons underlying that preference. Thanks to their knowledge of programming languages, developers naturally prefer to write code than draw lines and boxes. This is usually seen as a time-consuming and clerical task, even with proper tool support. Surprisingly, the participants who were used to graphical models also positively evaluated TVL. They not only liked the language but also the convenience offered by the textual edition interface.

stakeholder profiles influence the preference for the configuration model. The developers looked for ways to reuse the TVL model for configuration purposes. Their suggestion was to remove the unnecessary elements from the TVL model (unselected features and attributes) and directly use it as a product specification. The advantage of this approach is that comparison between products could be immediately achieved with a simple *diff*. In contrast, the designers were in favour of a graphical model, e.g. a tree-shaped FM, or more elaborate configuration interfaces like configuration wizards or tables.

6.3 Tool Support

At the moment, TVL only comes with a parser that checks syntactic and type correctness, as well as a configuration engine that supports decision propagation limited to Boolean values. We have also developed plugins for tree editors, namelly NotePad++ (Windows), Smultron (MacOS) and TextMate (MacOS). These plugins provide basic syntax highlighting and collapse/expand mechanisms to hide/show pieces of code.

Besides textual editors, out-of-the-box versioning tools like CVS or Subversion already support the collaborative editing of TVL models as any other text file, as reported by the OSL and Virage Logic participants. The interpretation of a change made to a TVL line is as easy as it is for programming code. By simply looking at the log, one can immediately see who changed what and when. In contrast, graphical models usually require dedicated tools with their own encoding, which makes interoperability and collaboration difficult.

The configuration capabilities of TVL have recently been applied to re-engineer the configuration menu of PloneMeeting. This resulted in a prototype that demonstrates how it is possible to use an application-specific web interface as frontend for a generic TVL-based configurator. Although very limited in functionality, the prototype gave the participant a better overview of the benefits of TVL. Surprisingly, the PloneMeeting participant was not interested in generating *appy.gen* code from a TVL model because of the Python code that would still have to be edited after generation. However, generating a TVL model from *appy.gen* code would greatly simplify constraint specification and validation. Tedious and error-prone Python code would no longer have to be maintained manually, and most of the constraints that are only available in the head of developers would become trivial to implement. Put simply, TVL would be used here as a domain-specific constraint language. We could not produce similar prototypes for the other cases because configuration interfaces were not available and/or access to the code was not granted.

A functionality not provided by the TVL tools but requested by the participants is *code completion* of language constructs, feature names and attributes. Another important functionality would be the *verification of the scope of features* in constraints. Since a constraint can contain any feature in the FM, it might rapidly become hard to identify whether the referenced feature is unique or if a relative path to it has to be given. The on-the-fly suggestion of alternative features by the editor would facilitate constraint definition and make it less error-prone. By extension, the on-the-fly checking of the satisfiability of the model would avoid wasting time later on debugging it. The downside of such checks is that they can be resource-intensive and should thus be carefully scheduled and optimized.

7 Threats to Validity

The evaluation was performed with four PLs and five participants, providing a diversity of domains and profiles. Yet, their diversity did not have a significant influence on the results since we observed a substantial overlap between their comments. Therefore, we believe that the results are valid for wide a range of organizations and products [36].

The TVL models were prepared in advance by the two researchers and later checked by the participants. Consequently, the expertise of the researchers might have influenced the models and the evaluation of the participants. In order to assess this potential bias more precisely, we will have to compare models designed by participants to models designed by the two researchers. However, TVL is arguably simpler than most programming languages and the modelling task was felt to be rather straightforward. As as consequence, we do not expect this to be a problem for our evaluation. Furthermore, when the participants questioned the design decisions, alternative solutions were discussed based on the constructs available in the language—even those not disclosed during the presentation.

The limited hands-on experience with TVL might have negatively influenced the evaluation of the expressiveness, notation and modularisation of the language, and positively impacted the evaluation of the ease and cost and learning experience. That situation resembles the setting of an out-of-box experience [37]. This gives valuable insight as to how software engineers perceive TVL after a one-hour training and how fast they can reach a good understanding of the fundamental concepts.

A more specific problem was the unavailability of proper documentation and the limited access granted to the codebase in the case of OSL, NXP and Virage Logic. This made the modelling of those cases more difficult.

In the case of OSL, the development team is still in the SPL adoption phase. This could be a threat as the participant has only been exposed to FMs for reviewing. Therefore, he might have focused on comparing the textual and graphical approaches rather than evaluating the language itself. Along the same lines, the PloneMeeting participant was already reluctant to use graphical FMs and might have evaluated the textual approach rather than TVL itself. In any case, we believe that the feedback received was more shaped by the expectations and requirements of the participants than by the preference for a textual representation over a graphical one.

More generally, one limitation of our study is the relatively small size of the subsystems we could deal with during the experiment.

8 Conclusion

Effective representations of FMs are an open research question. Textual alternatives have been proposed, including TVL, a textual variability modelling language meant to overcome a number of known deficiencies observed in other languages. Yet, evidence of the suitability of TVL in practice was missing. In this paper, we systematically evaluated the language by conducting an empirical evaluation on four industrial product lines.

Our evaluation of TVL showed that practitioners positively evaluated the notation and that efficiency gains are envisioned in terms of model comprehension, design and learning curve. However, they suggested some extensions like attribute cardinalities, feature cloning, default values and guidance messages that can be used during product configuration.

In the future we will focus on integrating the recommended extensions into TVL. Furthermore, the prototype implementation of the TVL parser and reasoner needs to be extended to better support on-the-fly verification of model consistency. To assess these new extensions, live evaluations through modelling sessions are envisaged. To better assess the pros and cons of variability modelling languages, comparative evaluations are planned, too.

Acknowledgements

We would like to thank Gaëtan Delannay from GeezTeem, Jacques Flamand from OSL, Hans den Boer and Jos Hegge from Virage Logic for the time they dedicated to our evaluation, and Bart Caerts from NXP Semiconductors for his contribution to the development of the CPU-load calculation variability model. This work is sponsored by the Interuniversity Attraction Poles Programme of the Belgian State, Belgian Science Policy, under the MoVES project, the Walloon Region under the NAPLES and the FIRST DOC.A programmes.

References

1. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-21, SEI, Carnegie Mellon University (November 1990)
2. Pohl, K., Bockle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, Heidelberg (2005)
3. Czarnecki, K., Helsen, S., Eisenecker, U.W.: Formalizing cardinality-based feature models and their specialization. Software Process: Improvement and Practice 10(1), 7–29 (2005)
4. Schobbens, P.Y., Heymans, P., Trigaux, J.C., Bontemps, Y.: Generic semantics of feature diagrams. Computer Networks, Special Issue on Feature Interactions in Emerging Application Domains, 38 (2006)

5. pure-systems GmbH: Variant management with pure: variants. Technical White Paper (2006), `http://www.pure-systems.com/fileadmin/downloads/pv-whitepaper-en-04.pdf`
6. van Deursen, A., Klint, P.: Domain-specific language design requires feature descriptions. Journal of Computing and Information Technology 10, 2002 (2002)
7. Batory, D.S.: Feature models, grammars, and propositional formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)
8. Boucher, Q., Classen, A., Faber, P., Heymans, P.: Introducing TVL, a text-based feature modelling language. In: VaMoS 2010, University of Duisburg-Essen, pp. 159–162 (January 2010)
9. Czarnecki, K.: From feature to concept modeling. In: VaMoS 2010, University of Duisburg-Essen, 11 Keynote (January 2010)
10. Nestor, D., O'Malley, L., Sikora, E., Thiel, S.: Visualisation of variability in software product line engineering. In: VaMoS 2007 (2007)
11. Cawley, C., Healy, P., Botterweck, G., Thiel, S.: Research tool to support feature configuration in software product lines. In: VaMoS 2010, University of Duisburg-Essen, pp. 179–182 (January 2010)
12. Dordowsky, F., Hipp, W.: Adopting software product line principles to manage software variants in a complex avionics system. In: SPLC 2009, San Francisco, CA, USA, pp. 265–274 (2009)
13. Classen, A., Boucher, Q., Faber, P., Heymans, P.: The TVL Specification. Technical report, PReCISE Research Centre, Univ. of Namur (2009)
14. Shull, F., Singer, J., Sjøberg, D.I.K.: Guide to Advanced Empirical Software Engineering. Springer-Verlag New York, Inc., Secaucus (2007)
15. Schobbens, P.Y., Heymans, P., Trigaux, J.C., Bontemps, Y.: Feature Diagrams: A Survey and A Formal Semantics. In: RE 2006, pp. 139–148 (September 2006)
16. Antkiewicz, M., Czarnecki, K.: Featureplugin: feature modeling plug-in for eclipse. In: OOPSLA 2004, pp. 67–72 (2004)
17. Beuche, D.: Modeling and building software product lines with pure::variants. In: SPLC 2008: Proceedings of the 2008 12th International Software Product Line Conference, Washington, DC, USA, p. 358. IEEE Computer Society, Los Alamitos (2008)
18. Kästner, C., Thüm, T., Saake, G., Feigenspan, J., Leich, T., Wielgorz, F., Apel, S.: FeatureIDE: A tool framework for feature-oriented software development. In: Proceedings of ICSE 2009, pp. 311–320 (2009)
19. Krueger, C.W.: Biglever software gears and the 3-tiered spl methodology. In: OOPSLA 2007: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion, pp. 844–845. ACM, New York (2007)
20. Mendonca, M., Branco, M., Cowan, D.: S.p.l.o.t. - software product lines online tools. In: Proceedings of OOPSLA 2009, pp. 761–762 (2009)
21. Mendonça, M.: Efficient Reasoning Techniques for Large Scale Feature Models. PhD thesis, University of Waterloo (2009)
22. Reiser, M.O.: Core concepts of the compositional variability management framework (cvm). Technical report, Technische Universität Berlin (2009)
23. Abele, A., Johansson, R., Lo, H., Papadopoulos, Y., Reiser, M.O., Servat, D., Torngren, M., Weber, M.: The cvm framework - a prototype tool for compositional variability management. In: Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2010), University of Duisburg-Essen, pp. 101–105 (January 2010)
24. Benavides, D., Segura, S., Trinidad, P., Cortés, A.R.: Fama: Tooling a framework for the automated analysis of feature models. In: Proceedings of VaMoS 2007, pp. 129–134 (2007)
25. Riebisch, M., Böllert, K., Streitferdt, D., Philippow, I.: Extending feature diagrams with uml multiplicities. In: IDPT 2002 (2002)

26. Czarnecki, K., Bednasch, T., Unger, P., Eisenecker, U.W.: Generative programming for embedded software: An industrial experience report. In: Batory, D., Blum, A., Taha, W. (eds.) GPCE 2002. LNCS, vol. 2487, pp. 156–172. Springer, Heidelberg (2002)
27. Pratt, T.W.: Programming Languages: Design and Implementation, 2nd edn., 604 pages. Prentice-Hall, Englewood Cliffs (1984)
28. Holtz, N., Rasdorf, W.: An evaluation of programming languages and language features for engineering software development. Engineering with Computers 3, 183–199 (1988)
29. PloneGov. (June 2010), `http://www.plonegov.org/`
30. Delannay, G., Mens, K., Heymans, P., Schobbens, P.Y., Zeippen, J.M.: Plonegov as an open source product line. In: OSSPL 2007, collocated with SPLC 2007 (2007)
31. Hubaux, A., Heymans, P., Benavides, D.: Variability modelling challenges from the trenches of an open source product line re-engineering project. In: SPLC 2008, Limerick, Ireland, pp. 55–64 (2008)
32. Unphon, H., Dittrich, Y., Hubaux, A.: Taking care of cooperation when evolving socially embedded systems: The plonemeeting case. In: CHASE 2009, collocated with ICSE 2009 (May 2009)
33. Océ Software Laboratories, S.A (June 2010), `http://www.osl.be/`
34. NXP Semiconductors (June 2010), `http://www.nxp.com/`
35. Virage Logic (June 2010), `http://www.viragelogic.com/`
36. Yin, R.K.: Case Study Research: Design and Methods, 3rd edn. Applied Social Research Methods, vol. 5. Sage Publications, Inc., Thousand Oaks (2002)
37. Sangwan, S., Hian, C.K.: User-centered design: marketing implications from initial experience in technology supported products. In: Press, I.C.S. (ed.) Engineering Management Conference, vol. 3, pp. 1042–1046 (2004)

Extending DMM Behavior Specifications for Visual Execution and Debugging

Nils Bandener, Christian Soltenborn, and Gregor Engels

University of Paderborn, Warburger Straße 100, 33098 Paderborn, Germany
{nilsb,christian,engels}@uni-paderborn.de

Abstract. Dynamic Meta Modeling (DMM) is a visual semantics specification technique targeted at behavioral languages equipped with a metamodel defining the language's abstract syntax. Given a model and a DMM specification, a transition system can be computed which represents the semantics of that model. It allows for the investigation of the model's behavior, e.g. for the sake of understanding the model's semantics or to verify that certain requirements are fulfilled. However, due to a number of reasons such as tooling and the size of the resulting transition systems, the manual inspection of the resulting transition system is cumbersome.

One solution would be a visualization of the model's behavior using animated concrete syntax. In this paper, we show how we have enhanced DMM such that visual execution and debugging can be added to a language in a simple manner.

1 Introduction

One challenge of today's software engineering is the fact that software systems become more and more complex, making it hard to produce systems which work correctly under all possible executions. As a result, the *Object Management Group* (OMG) has proposed the approach of *Model-Driven Architecture* (MDA). The main idea of MDA is to start with an abstract, platform-independent *model* of the system, and to then refine that model step by step, finally generating platform-specific, executable code.

In this process, *behavioral* models (e.g., UML Activities) play an increasingly important role; they allow to model the system's desired behavior in an abstract, visual way. This has a couple of advantages, one of the most important being that such visual models can be used as a base for communication with the system's stakeholders (in contrast to e.g. Java code).

However, to get the most usage out of behavioral models, their semantics has to be defined precisely and non-ambiguously; otherwise, different interpretations of a model's meaning may occur, leading to all kinds of severe problems. Unfortunately, the UML specification [15] does not fulfill that requirement: The semantics of the behavioral models is given as natural text, leaving room for different interpretations.

One solution would be to specify the semantics of these behavioral languages with a *formal* semantics, i.e., some kind of mathematical model of the language's behavior. A major advantage of such a specification is that it can be processed automatically, e.g. for verifying the specification for contradictory statements.

Dynamic Meta Modeling (DMM) [9,13] is a semantics specification technique which results in semantics specifications that are not only formal, but also claim to be easily understandable. The only prerequisite for using DMM is that the syntax of the language under consideration is defined by means of a metamodel.

In a nutshell, DMM works as follows: In a first step, the language engineer creates a so-called *runtime metamodel* by enhancing the syntax metamodel with concepts needed to express states of execution of a model. For instance, in the case of the UML, the specification states that "the semantics of UML Activities is based on token flow". As a result, the runtime metamodel contains a Token class.

The second step consists of creating *operational rules* which describe how instances of the runtime metamodel change through time. For instance, the DMM specification for UML Activities contains a rule which makes sure that an Action is executed as soon as Tokens are sitting on all incoming ActivityEdges of that Action.

Now, given a model (e.g., a concrete UML Activity) and an according DMM specification, a transition system can be computed, where states are instances of the runtime metamodel (i.e., states of execution of the model), and transitions are applications of the operational rules. The transition system represents the complete behavior of the model under consideration and can therefore be used for answering all kinds of questions about the model's behavior.

However, investigating such a transition system is a difficult and cumbersome task for a number of reasons. First of all, we have seen that the states of the transition system are instances of the runtime metamodel, which can be pretty difficult to comprehend. Additionally, due to the so-called *state explosion problem*, the transition systems tend to be pretty large.

One solution for (at least partly) solving this problem would be to show the execution of a model in the model's own *concrete syntax*. This has two major benefits:

– It is significantly easier to find interesting states of execution, e.g., situations where a particular Action is executed.
– Investigating the states of the transition system only is an option for advanced language users, i.e., people who are at least familiar with the language's metamodel. In contrast, visualizing the model execution in concrete syntax is much easier to comprehend.

In this paper, we show how we extended the DMM approach to allow for exactly that. We will show how the language engineer (i.e., the person who defines a modeling language) can make her language visualizable and debuggable by adding a couple of simple models containing all information necessary to visualize a model's execution, and how this information is used to extend existing visual editors at runtime for the sake of showing the model execution. As a result, the language engineer can make the language under consideration visualizable and debuggable without writing a single line of code.

Structure of paper. In the next section, we will give a short introduction to DMM, and we will briefly introduce the components we used to implement model visualization in a generic way. Based on that, Sect. 3 will show what information the language engineer has to provide, and how this information is specified by means of certain *configuration models*. Finally, Sect. 3 will briefly explain how we integrated our approach into the existing tooling. Section 5 will discuss work related to our approach, and finally, Sect. 6 will conclude and point out some future work.

2 Dynamic Meta Modeling

This section introduces the foundations needed for the understanding of the main section 3. It gives a very brief introduction to Dynamic Meta Modeling (DMM).

As already mentioned in the introduction, DMM is a language dedicated to the specification of behavioral semantics of languages whose syntax is defined by means of a metamodel. The general idea is to enhance the syntax metamodel with information needed to express states of execution of a model; the enhanced metamodel is called *runtime metamodel*. The actual behavior is then specified by means of operational rules which are typed over the runtime metamodel.

Given such a runtime metamodel and a set of DMM rules, a transition system can be computed. This is done as follows: First, a *semantic mapping* is used to map an instance of the syntax metamodel into an instance of the runtime metamodel; that model will then serve as the initial state of the transition system to be computed. Now, all matching DMM rules are applied to the initial state, resulting in a number of new states. This process is repeated until no new states are discovered. The resulting transition system represents the complete behavior of a particular model. It can then e.g. be analyzed with model checking techniques (see [10]). An overview of the DMM approach is depicted as Fig. 1.

Let us demonstrate the above using the language of UML Activities. A careful investigation of the UML specification reveals that the semantics of Activities is based on tokens flowing through the Activity; as consequence, two runtime states of the same Activity differ in the location of the flowing tokens.

In fact, the semantics is significantly more difficult: Tokens do not actually flow through an Activity. Instead, they are only *offered* to ActivityEdges. Only if an Offer arrives at an Action (and that Action is being executed), the Token owning the Offer moves. Figure 2 shows an excerpt of the runtime metamodel for UML Activities; elements depicted in white are part of the syntax metamodel, gray elements belong to the runtime part of the metamodel.

Now for the specification of the semantics' dynamic part: As mentioned above, this is done by operational rules. A DMM rule basically is an annotated object diagram. It *matches* a state if the rule's object structure is contained in that state. If this is the case, the rule is *applied*, i.e., the modifications induced by

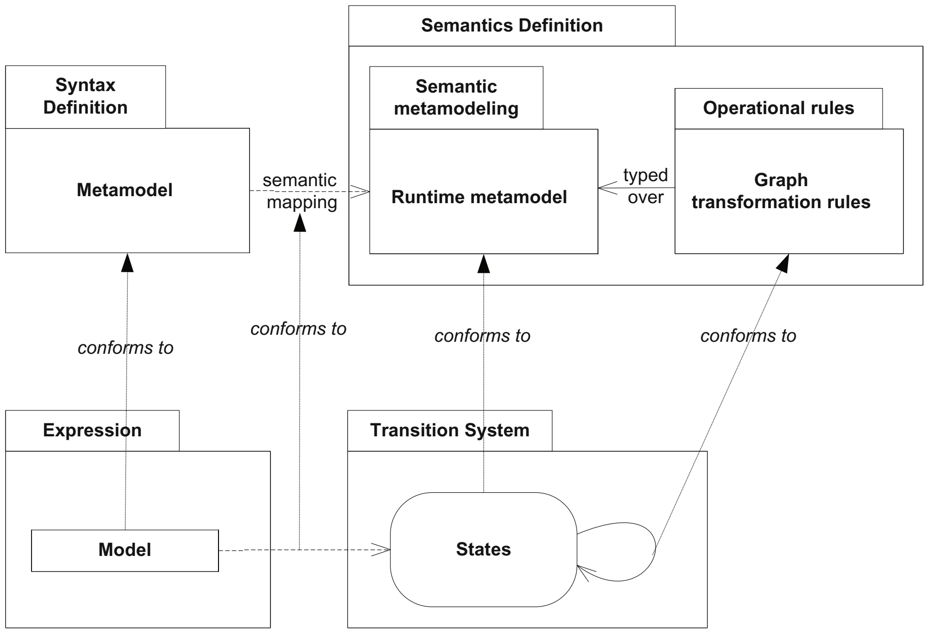


Fig. 1. Overview of the DMM approach

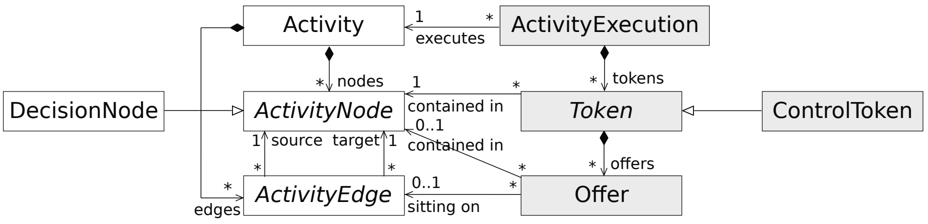


Fig. 2. Excerpt of the runtime metamodel

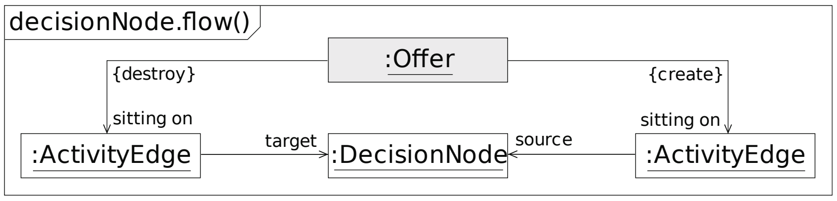


Fig. 3. DMM Rule decisionNode.flow()

the annotations are performed on the found object structure, leading to a new state.

Figure 3 shows a simple DMM rule with name decisionNode.flow(). Its semantics is as follows: The rule matches if the incoming ActivityEdge of a DecisionNode carries an Offer. If this is the case, the Offer is moved to one of the outgoing edges. Note that usually, a DecisionNode has more than one outgoing edge. In this case, the rule's object structure is contained in the state more than once; every occurence consists of the incoming ActivityEdge, the DecisionNode, the Offer, and one of the outgoing ActivityEdges. As a result, we will end up with a new state for every outgoing edge, and the states will differ in the location of the Offer (which will sit on the according outgoing edge of that state). In other words: We end up with one new state for every possible way the Offer can go.

Note that DMM specifications usually do not aim at making models executable – if this would be the case, rule 3 would need to evaluate the guards of the DecisionNode's outgoing edges to determine the edge to which the Offer has to be routed. Instead, the transition system representing the model's behavior contains all possible executions of that model. This allows to analyze the behavior of Activities which are modeled rather informally (e.g., if the guards are strings such as "Claim valid"). See [10] for an example of such an analysis.

Technically, DMM rules are *typed graph transformation rules* (GTRs) [18]. DMM supports a couple of advanced features: For instance, *universally quantified structures* can be used to manipulate all occurrences of a node within one rule; *negative application conditions* allow to describe object structures which prevent a rule from matching; additionally, DMM allows for the usage and manipulation of attributes.

The main difference to common GTRs is the fact that DMM rules come in two flavors: *bigstep rules* and *smallstep rules*. Bigstep rules basically work as common GTRs: They are applied as soon as they match as described above. In contrast, smallstep rules have to be explicitly *invoked* by a bigstep rule or another smallstep rule;[1] as long as there are smallstep rules which have been invoked, but are yet to be processed, bigstep rules cannot match.

To actually compute a transition system from a model and a DMM semantics specification, the model as well as the set of DMM rules are translated into a graph grammar suitable for the graph transformation tool GROOVE [17]. Slightly simplified, this is done as follows: The model is translated into a GROOVE state graph which serves as the initial state of the transition system to be computed. Furthermore, each DMM rule is translated into an according GROOVE rule; features not directly supported by GROOVE are translated into structures within the GROOVE rules which make sure that the DMM rule's behavior is reflected by the GROOVE rule. For instance, to be able to handle invocation of rules, an actual *invocation stack* is added to the initial

[1] Note that the invocation of a smallstep rule might fail in case it is invoked, but the object structure required by the invoked rule is not contained in the current state, resulting in the rule not matching; a DMM semantics specification potentially leading to such situations is considered to be broken.

state, and the GROOVE rules have structures which manipulate that invocation stack and make sure that smallstep rules can only match if an according invocation is on top of the stack.

3 Visual Model Execution

With DMM, we can define the semantics of a modeling language, calculate execution states of model instances, and apply further analytical methods to it. Yet, the basic concept of DMM provides no means to visualize the execution and the therein occurring states of a model; a feature which is feasible for monitoring, better understanding, or debugging a model—especially if it is written in a visual language.

Thus, we have developed a tool for visually executing and debugging models with DMM-specified semantics, the *DMM Player* [1]. From a technical perspective, the tool is a set of Eclipse plug-ins, which is able to process models and DMM semantics specified with the *Eclipse Modeling Framework* (EMF) [5]. The visualization is realised using the *Graphical Modeling Framework* (GMF) [7], which is the standard way of providing visual editors for EMF-based models. As there are already numerous existing GMF editors for behavioral models which—however—do not support displaying runtime-information such as tokens or active states, the DMM Player also provides means to augment existing editors by such elements.

Leaving those pesky technical details behind, we will now focus on the underlying concepts which make the visualization of a model execution possible. Using UML Activities as a running example, we start with the fundamental question on how to visualize execution states and how the augmentation of existing visualizations can be specified in a model-driven way. Beneath the graphical dimensions, we will also have to consider the time axis when visualizing the execution. This will be covered in Sect. 3.2. Section 3.3 covers means of controlling the execution path when external choices are necessary. Section 3.4 introduces concepts that make debugging of models in the presented environment possible. Finally, in Sect. 3.5 we will demonstrate our approach on another language, i.e., UML Statemachines.

3.1 Visualizing Runtime Information

In order to visualize the behavior of a model, i.e., the development of its runtime state over time, it is obviously essential to be able to visualize the model's runtime state at all. However, this cannot be taken for granted. While certain visual languages have an inherent visual syntax for runtime information—such as Petri nets [16] visualizing the state using tokens on places—many visual languages only support the definition of the static structure—such as UML activity diagrams.

The specification for activity diagrams only informally describes the semantics using concepts such as tokens, which are comparable to the tokens used by Petri nets, and offers, which act as a kind of path finder for tokens. The specification does not provide any runtime information support in the formally specified metamodel and also does not give any guidelines on how to visualize runtime information. This is where DMM comes into play.

As we have seen in the previous section, the core concept of DMM translates this informal description into the runtime metamodel, which formally defines an abstract syntax for runtime states of models in the particular language. With the DMM Player, we have developed a set of concepts and techniques to define a concrete syntax for those runtime states. Similar to the enhancement by runtime information in the abstract syntax, the concept allows for building on the concrete syntax of the static part of models in order to create the concrete syntax for runtime states. The enhanced concrete syntax is defined using a completely declarative, model-based way.

In order to create such a visualization with the DMM Player, three ingredients are needed: An idea on how the concrete syntax should look like, the DMM runtime metamodel, and an existing extensible visualization implementation for the static structure of the particular language.

Our implementation of the DMM Player allows for extending GMF-based editors, as GMF offers all required extension mechanisms. The particular implementation is described in Sect. 4. In the following, we will focus on the concepts, which are—while being partially inspired by—independent from GMF. Essentially, this means that definitions for an enhanced concrete syntax may be also used in conjunction with other frameworks. This of course requires an implementation interpreting the DMM artifacts for these particular frameworks.

The first concern is how the runtime information should be visualized in concrete syntax. Beneath the obvious question on the shape or appearance of runtime information, it may also be necessary to ask what runtime objects should be included in the visualization at all. Certain runtime information may be only useful in certain contexts. In the example of UML activity diagrams, the visualization of tokens is certainly essential; we visualize tokens—aligned with the visualization in Petri nets—as filled black circles attached to activity nodes. An example for such a diagram can be seen in Fig. 4. For debugging of models and semantics, visualized offers may also be useful; offers are visualized as hollow circles. As multiple tokens and offers may be in action at once, an arrow visualizes which offers are owned by which tokens.

The diagram in Fig. 4 also shows boxes labeled with the letters EX. These boxes indicate that the particular node is currently executing. We will not go any further into the semantics of these boxes, though.

Having an idea on how the concrete syntax should look, we combine it with the formal structure of the runtime metamodel to create a so called *diagram augmentation model* which associates certain parts of the runtime metamodel with visual shapes. The word "augmentation" in the name of the model refers to the fact that it is used by the DMM Player to augment the third ingredient, the preexisting static diagram visualization, with runtime information.

Diagram augmentation models use the metamodel which is partially pictured in Fig. 5. The class DiagramAugmentationModel is the root element, i.e. each augmentation model contains exactly one instance of this class. The attribute diagramType is used to associate a particular diagram editor with the augmentation model - this diagram editor will then be used for displaying runtime states.

The actual elements to be visualized are determined by the classes AugmentationNode and AugmentationEdge. More precisely, as both classes are abstract, subclasses of these classes must be used in an instance of the meta model. The subclasses determine the implementation type of the visualization.

The way the elements are integrated into the existing diagram is determined by the references between AugmentationNode and AugmentationEdge on the one side and EReference and EClass on the other side. The latter two classes stem from the Ecore metamodel, which is the EMF implementation of the meta-metamodel standard MOF. They represent elements from the DMM runtime metamodel the visualization is supposed to be based on.

In the case of an AugmentationNode, the following references need to be set: The reference augmentationClass determines the class from the runtime metamodel which is visualized by this particular node; however, this is not sufficient, as the class needs to be somehow connected to elements that already exist in the visualization of the static structure. For instance, a token is linked to an activity node and should thus be visually attached to that node. This connection is realized by the reference named references; it must point to an EReference object which emanates from the referenced augmentationClass or one of its super classes. The EReference object in turn must point towards the model element the augmenting element should be visually attached to. Thus, this model element must stem from the static metamodel and must be visualized by the diagram visualization to be augmented. The reference containment is only relevant if the user shall be able to create new elements of the visualized type directly in the editor; those elements will be added into the containment reference specified here.

Figure 6 shows the part of the augmentation model for UML activities which specifies the appearance of control tokens, which are a subclass of tokens. The references link specifies that the reference named contained_in determines to which ActivityNode objects the new nodes should be attached to. The references link is part of the class Token. However, as the link augmentationClass specifies the class ControlToken as the class to be visualized, this particular ShapeAugmentationNode will not come into effect when other types of tokens occur in a runtime model. Thus, other augmentation nodes may be specified for other tokens.

3.2 Defining the Steps of Executions

Being able to visualize individual runtime states, creating animated visualizations of a model's behavior is straightforward. Sequentially applying the rules of the DMM semantics specification yields a sequence of runtime states which can be visualized with a brief pause in-between, thus creating an animation.

However, the sequence of states produced by DMM is not necessarily well suited for a visualization. In many cases, subsequent states only differ in parts that are not visualized. These parts are primarily responsible for internal

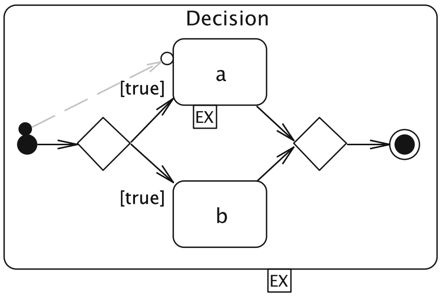


Fig. 4. An UML activity diagram with additional runtime elements

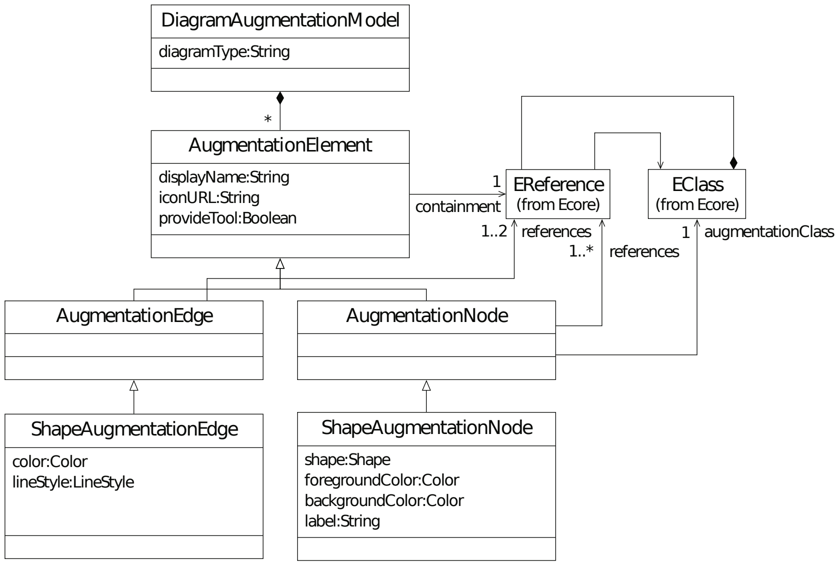


Fig. 5. Excerpt of the metamodel for diagram augmentation models

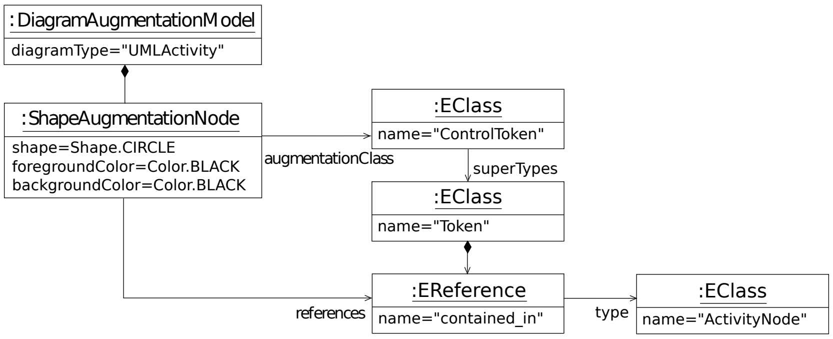


Fig. 6. Excerpt of the augmentation model for UML activities

information which is specific to the particular implementation of the semantics, but is not relevant for the behavior of the final model. Including these steps in the animation would cause strange, irregular pauses between visual steps.

Furthermore, DMM semantics may produce states with temporary inconsistencies. These states also result from implementation details of the particular DMM semantics specification. DMM rules may modify a model using several consecutive invocations of other rules; this is necessary if the semantics of a language element is too complex to be described within one rule. Each invoked rule produces a new state, which however might be wrong—when viewed from pure semantics perspective without implementation details. In the case of the particular implementation the state is of course still correct, as the subsequently invoked rules correct this inconsistency, thus making it a temporary inconsistency.

Figure 7 shows an example for a temporary inconsistency. In the first state, an offer has reached the final node. The activity diagram semantics now demands that the corresponding token should be moved to the node reached by the offer. The DMM implementation of the semantics however creates a new token on the target node before removing the original token from its location. This creates the exhibited temporary inconsistency with two tokens being visible at once.

The visualization of temporary inconsistencies might be interesting for the developer of the DMM semantics implementation; for a user only interested in viewing the behavior of a model, such states should not be visualized.

Thus, we need a way of selecting the states that should be displayed to the user. There is a number of different approaches to that problem which we will briefly discuss in the following.

If the visualization of temporary inconsistencies is desired and only the aforementioned problem of steps without visual changes needs to be addressed, a very simple solution is obvious: Using the diagram augmentation model, it is possible to determine what elements of the runtime model are visualized. The DMM Player can use this information to scan the consecutive states for visual changes; only if changes are detected, a visual step is assumed and thus promoted to the user interface.

If temporary inconsistencies are to be avoided in the visualization, other measures need to be applied. A simple and straight-forward approach would be to visualize only the state when the application of a Bigstep rule has been finished; application in this context means that the changes by the Bigstep rule and by its invocations have been performed. As temporary inconsistencies are typically raised by an invocation and again fixed by a consecutive invocation, temporary inconsistencies will be fixed when all invocations have been finished and thus the application of a bigstep rule has been finished.

Yet, the structuring concept of Bigstep and Smallstep rules has not been designed for visualization purposes; thus, it is also possible to find cases in which a state produced by a Smallstep rule should be visualized while the application of the invoking Bigstep rule has not been finished yet. Just restricting the visualization to states left by Bigstep rules is thus too restrictive.

An obvious solution would be the explicit specification of all rules that should trigger a visual step. This is, however, also the most laborious solution, as each rule of the semantics specification needs to be checked. In the case of the DMM-based UML activity semantics specification, this means the inspection of 217 rules.

Our solution for now is a combination of the approaches. Thus, in addition to the specification of individual rules triggering visual steps, the DMM Player also allows for the specification of all Bigstep rules. Furthermore, it is possible to specify whether the visualization should be triggered before or after the application of the particular rules.

For creating a suitable animated visualization with the DMM UML activity semantics, it is sufficient to trigger a visual step after the application of all Bigstep rules and after the application of only one further Smallstep rule, which takes the responsibility of moving a token to the new activity node that has accepted the preceding offer.

3.3 Controlling Execution Paths

A limitation of the behavior visualization using an animated sequence of states is its linearity. In some cases, the behavior of a model may not be unambiguously defined. For instance, this is the case in the activity diagram we have seen before in Fig. 4; the left decision node has two outgoing transitions. Both are always usable as indicated by the guard [true]. In a transition system, such a behavior is reflected by a fork of transitions leading from one state to several distinct states. In an animation, it is necessary to choose one path of the fork. At first sight, it is evident that such a choice should be offered to the user.

The DMM Player can offer this choice to the user by pausing the execution and visualizing the possible choices; after the user has made a choice, execution continues.

However, there are cases in which it is not feasible for the user to choose the path to be used for every fork in the transition system. This is primarily the case for forks caused by concurrency in the executed model. Even though a linear execution does not directly suffer from state space explosion, concurrency might require a decision to be made before most steps of a model execution.

As the semantics of concurrency can be interpreted as an undefined execution order, it is reasonable to let the system make the decision about the execution order automatically. Forks in the transition system which are caused by model constructs with other semantics—such as decision nodes—should however support execution control by user interaction.

The problem is now to distinguish transition system forks that should require user interaction from others. More precisely, as a single fork can both contain transitions caused by decision nodes and by concurrency, it is also necessary to identify the portions of a fork that are supposed to form the choices given to the user.

A basic measure for identifying the model construct that caused a fork or a part of it is considering the transformation rules that are used for the transitions forming the fork. In the case of the DMM semantics for UML activities, the transitions which cause the forks at decision nodes are produced by the Bigstep rule decisionNode.flow() (see also Fig. 3). Forks which consist of transitions caused by other rules can be regarded as forks caused by concurrency.

Just considering the rules causing the transitions is not sufficient, though. Concurrency might lead to forks which consist of decisionNode.flow() transitions belonging to different decision nodes in the model. If each of those decision node has only one active outgoing transition, there is no choice to be made by the user but just choices purely caused by concurrency.

Thus, it is necessary to group the transitions at a fork by the model element they are related to. This model element can be identified by using the rule match associated to the particular transition (see Sect. 2 for a brief explanation of rule matching and application). The match is a morphism between the nodes of the particular DMM rule and elements from the model. Generally, one node from a rule that is supposed to trigger a user choice can be used to identify the related model element. In the case of the DMM activity semantics, this is the :DecisionNode element itself.

With these components, we can build an algorithm for identifying the instances of transition system forks in which the user should be asked for a choice: Group the transitions that are possible from the current state by the rule and by the elements bound to the grouping node defined for the particular rule. If there is no grouping node or the rule is not supposed to cause user choices, the particular transition forms a group of size one. Now one of these groups is arbitrarily chosen by the software; this reflects possible concurrency between the single groups. If the chosen group contains more than one transition, the user is requested to make a choice. Otherwise the single transition in the arbitrarily chosen group will be used to gather the next state.

This algorithm enables us to ensure that the user is only required to make choices regarding single instances of certain model constructs, such as decision nodes. A remaining problem is how to give the user an overview over the possible choices. We can again utilize nodes from the rules that are supposed to trigger user choices. Those rules contain as a matter of principle always a node which represents the different targets which can be reached while the aforementioned grouping node stays constantly bound to the same model element. If the model element represented by the target node is part of the diagram, this diagram element can be used for identifying the different choices.

In the case of UML activities, this is the target ActivityEdge node in the rule decisionNode.flow(). The DMM Player can now use these model elements to visualize the possible choices using generic marker signs as is depicted in Fig. 8. The user may easily select one of the choices using the context menu of one of these model elements.

3.4 Debugging Concepts

As we have seen up to now, the main objective of the DMM Player is the visualization of a model's behavior by means of animated concrete syntax. This

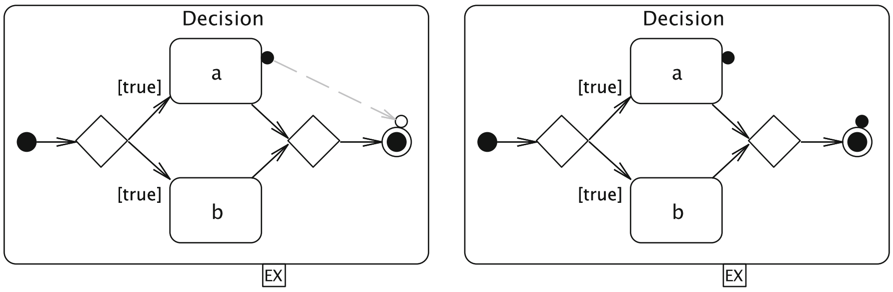


Fig. 7. A sequence of states exhibiting a temporary inconsistency

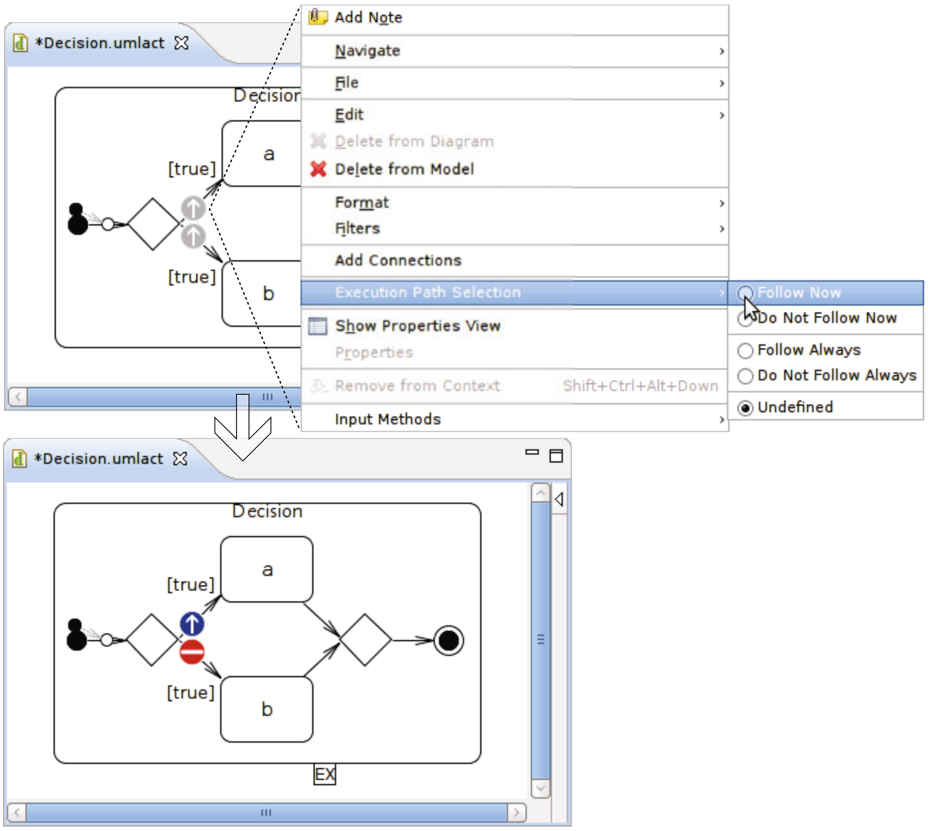


Fig. 8. UI for choosing execution paths

is very useful when trying to understand the details of a model's behavior, for instance if the model contains flaws. One feature which would obviously be of great use in such scenarios is the possibility to stop the execution of a model as soon as certain states of execution are reached. In other words: The transformation of concepts known from debuggers for classical programming languages to the DMM world would stand to reason.

The main concept of classical debuggers is the breakpoint; in source code, this is a line marked in such a way that the program execution is suspended just before the statements on that line are executed. When the execution is suspended, the values of program variables can be inspected by the user. Transferred to DMM, rules are the main units of execution; thus, the DMM Player supports setting breakpoints on DMM rules. Before (or, configurable, after) a rule is applied, the DMM Player suspends the execution. Using the property editors supplied by GMF, the user may now inspect the state of the model.

A variant of breakpoints are watchpoints. These cause the execution to be suspended when the condition defined by the watchpoint becomes true for variables in the executed program. *Property rules* can be seen as an analogical concept in the graph transformation and DMM world. Property rules do not modify the state of a model, and so they do not change a language's semantics. Their only purpose is to recognize certain states by matching them. Using a rule breakpoint, it is possible to suspend execution as soon as a property rule matches.

3.5 Example: UML Statemachines

To further demonstrate the usefulness of our approach, within this section we provide a second language for which we have applied our approach. Despite its visual similarity to UML Activities, we decided to use UML Statemachines. We would have preferred to use UML Interactions; unfortunately, we had issues with the GMF editor for Interactions as provided by the Eclipse UML2 tools [8], which we used in the preliminary version 0.9.

Let us briefly discuss the language of UML Statemachines. Syntactically, a Statemachine mainly consists of states and transitions between those states. At every point in time, a Statemachine has at least one active state. There are different kinds of states, the most important ones being the *Simple state* and the *Complex state* (the latter will usually contain one or more states). The semantics of transitions depends on their context: For instance, an unlabeled transition from a complex state's border to another state models that the complex state can be left while any of its inner state(s) is active. More advanced concepts like *history nodes* allow to model situations where, depending on different past executions, the Statemachine will activate different states.

A sample Statemachine is depicted as Fig. 9 (note that this figure already contains runtime information). The first active state will be state *A1*. From this state, either state *A2* or *A3* will be activated. In case of state *A3*, the *Complex State 1* will be entered. The state marked *H** is a so-called *deep history state*; it makes sure that in case state *Complex State 1* is activated again, all states which were active when that state was left are reactivated.

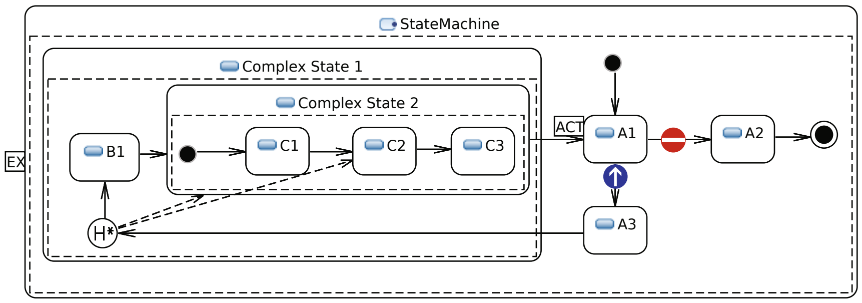


Fig. 9. Example state of Statemachine execution

We now want to briefly investigate the DMM semantics specification of UML Statemachines. As we have seen before, states of execution[2] of a Statemachine are determined by the active states. As a consequence, the runtime metamodel of Statemachines contains the concept of a Marker which references the currently active states (and will be moved by according DMM operational rules). To remember the last active states in case a complex state is left that contains a history state, the runtime metamodel introduces the HistoryMarker.

Next, we want to discuss how the execution of a Statemachine is visualized. In Fig. 9, we have already seen a Statemachine augmented with runtime information. Active states can be recognized by an attached box with the label *ACT*. These boxes represent the Marker instances from the runtime metamodel.

Further runtime information can be seen around the deep history state H*. The dashed arrows pointing away from that state signify the states that will be activated as soon as the complex state containing the history state is entered again. Thus, these arrows represent HistoryMarker instances. The part of the augmentation model that realizes the arrows representing history markers can be seen in Fig. 10. The ShapeAugmentationEdge instance specifies the class to be additionally visualized, i.e., the HistoryMarker and its references which determine the end points of the visualized edge.

We are now ready to explain the runtime state of the Statemachine which can be seen in Fig. 9. The currently active state is *A1*. Since from that state, either state *A2* or *A3* can be reached, the DMM player has already asked for a user decision – as the icons show, the user has decided to follow the transition leading to state *A3*.

Moreover, the visualization reveals that *Complex State 1* had already been active in the past. This is because there do exist HistoryMarker edges. The edges point to the states which had been active within state *Complex State 1* before it was left through the transition between *Complex State 2* and *A1* (i.e., *Complex State 2* and, within that state, *C2*). Therefore, after two further execution steps, these states will be set active again.

[2] Note that *state* is overloaded here; as before, *state of execution* refers to the complete model.

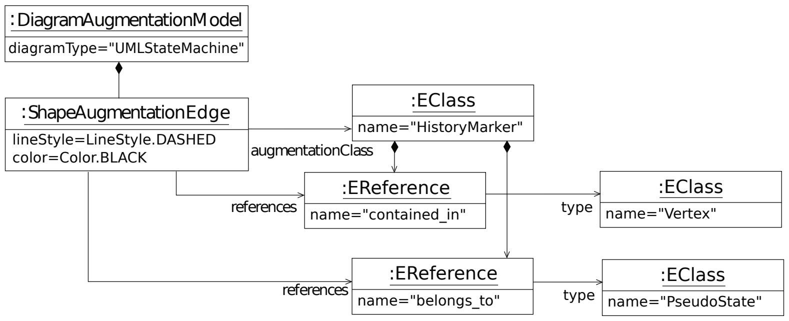


Fig. 10. Excerpt of the augmentation model for UML Statemachines

4 Implementation

This section will give insights into our implementation of the concepts described in the last section. However, our explanations are rather high-level – the reader interested in more technical details is pointed to [1]. A high-level view of the DMM Player's architecture is provided as Fig. 11.

As mentioned above, the DMM Player builds upon Eclipse technologies; still, the concepts of DMM are completely technology-independent. The implementation can be divided into two mostly independent parts: The diagram augmentation and the model execution. Both are connected by the EMF [5] model just using its standard interfaces; the model execution process changes the model. The diagram augmentation part listens for such changes and updates the visualization accordingly.

The model execution part utilizes EProvide [19], a generic framework for executing behavioral models inside of Eclipse. EProvide decouples the actual execution semantics and the method to define them using two layers:

On the first layer, EProvide allows to configure the *semantics description language*, which provides the base for the actual definition. The DMM Player registers DMM as such a language. The second layer defines the actual *execution semantics* for a language using one of the languages from the first layer. Thus, a DMM semantics definition—such as the UML activities definition—is defined at this level.

EProvide essentially acts as an adaptor of the semantics description languages to the Eclipse UI on one side and EMF-based models which shall be executed on the other side. The DMM Player code receives commands from EProvide along with the model to be executed and the semantics specification to be used. The most important command is the step, i.e., the command to execute the next atomic step in the given model. The DMM implementation realizes that step by letting the backing graph transformation tool GROOVE [17] perform the application of the according rule, and by translating the manipulations of the GROOVE rule back to the EMF model which is visualized.

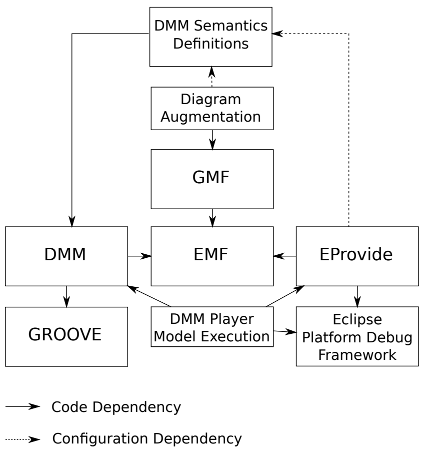

DMM Semantics Definitions
Diagram Augmentation
GMF
DMM
EMF
EProvide
GROOVE
DMM Player Model Execution
Eclipse Platform Debug Framework
Code Dependency
Configuration Dependency

Fig. 11. Architecture of DMM tooling

The advanced features, such as the definition of visual steps—which actually combines multiple steps into single ones—, the user control of execution paths, and the debugging functionality, are realized directly by the DMM Player. The EProvide module MODEF [3] also offers debugging functionality which, however, could not be directly utilized, as it makes a quite strong assumption. It is assumed that the model's state can be deduced from one single model element. Since this is not the case with DMM, where a state is a complete model, we needed to bypass this module.

The implementation of the DMM debugging facilities makes use of the Eclipse Debugging Framework. At every point in time, the DMM Player keeps track of the GROOVE rule applied in the last step to derive the current state, as well as the rules matching that new state. If a breakpoint or watchpoint is reached, the execution is suspended as desired.

The diagram augmentation part of the DMM Player uses interfaces of GMF [7] for extending existing diagram editors. GMF offers quite extensive and flexible means for customizing editors using extension points and factory and decorator patterns.

GMF uses a three-layer architecture to realize diagram editors: Based on the abstract syntax model on the lowest level, a view model is calculated for the mid layer. The view model is simply a model representation of the graph to be visualized, i.e., it models nodes that are connected by edges. On the third layer, the actual UI visualization components are created for the elements from the second layer. Thus, specific elements get a specific look.

The DMM Player hooks into the mapping processes between the layers; between the abstract syntax model and the view model, it takes care that the elements defined in the augmentation model are included in the view model. Between view model and the actual UI, it chooses the correct components and thus the correct appearance for the augmenting elements.

Thus, the DMM Player provides a completely declarative, model driven way of augmenting diagram editors; there is no need to writing new or altering existing source code. Figure 8 shows screenshots of the activity diagram editor that comes with the Eclipse UML2 Tools which has been augmented by runtime elements using the DMM Player.

The DMM Player is designed to be generically usable with any DMM semantics specification. Thus, it offers extension points and configuration models that just need to be adapted in order to use a semantics specification with the DMM Player.

5 Related Work

The scientific work related to ours can mainly be grouped into two categories: Visualization of program execution and animation of visual languages. For the former, we only want to mention eDOBS [12], which is part of the Fujaba tool suite. eDOBS can be used to visualize a Java program's heap as an object diagram, allowing for an easy understanding of program states without having to learn Java syntax; as a result, one of the main usages of eDOBS is in education. In contrast to that, the concrete syntax of such eDOBS visualizations is fixed (i.e., UML object diagrams), whereas in our approach, the modeler has to come up with his own implementation of the concrete syntax, but is much more flexible in formulating it.

In the area of graph grammars and their applications, there are a number of approaches related to ours: For instance, in [14,2,11], the authors use GTRs to specify the abstract syntax of the language under consideration and operations allowed on language instances. The main difference to our approach is that in [14,2], the actual semantics of the language for which an editor/simulator is to be modeled is not as clearly separated from the specification of the animation as in DMM, where the concrete syntax just reflects what are in fact model changes caused solely by (semantical) DMM rules. In the Tiger approach [11], a GEF [6] editor is generated from the GTRs such that it only allows for edit operations equivalent to the ones defined as GTRs; however, Tiger does not allow for animated concrete syntax.

Another related work is [4]; the DEViL toolset allows to use textual DSLs to specify abstract and concrete syntax of a visual editor as well as the language's semantics. From that, a visual editor can be generated which allows to create, edit, and simulate a model. The simulation uses smooth animations based on *linear graphical interpolation* as default; only the animation of elements which shall behave differently needs to be specified by the language engineer.

There is one major difference from all approaches mentioned to ours: As we have seen in Sect. 3.1, DMM allows for the easy reuse of existing (GMF based) editors. As a result, the language engineer only has to create the concrete syntax for the runtime elements not contained in the language's syntax metamodel, in contrast to the above approaches, where an editor always has to be created from scratch; reusing and extending an existing editor at runtime is not possible.

6 Conclusions

Visually executing a behavioral model as animated concrete syntax is an intuitive, natural way to understand the model's behavior. In this paper, we have shown how we have extended our DMM approach to allow for exactly that.

For this, we have first given a brief overview of DMM and the involved technologies in Sect. 1. Based on that, and using the running example of UML Activities, we have explained in Sect. 3 how the information needed to visually execute a model is added to DMM specification, and we have shown how this information is used to reuse existing GMF editors for the animation task by adding the according functionality to the editors at runtime, transparently to the user. Finally, we have discussed work related to ours in Sect. 5.

We believe that with the concepts and techniques described in this paper, we have achieved the next step towards a comprehensive toolbox for engineering behavioral visual languages. In a next step, we will integrate the described techniques more tightly into our DMM workflow. An obvious such integration could work as follows: As mentioned in Sect. 2, models equipped with a DMM semantics specification can be analyzed using model checking techniques. Now, if the result of the model checker is a counterexample (i.e., if a property does *not* hold for the model under consideration), the DMM Player can be used to visualize that counterexample, visually showing under which circumstances the property is violated.

Another area of our research is motivated by the fact that different people working with DMM might want to see different amounts of detail while simulating a model. For instance, the language engineer probably wants to see temporary inconsistencies while developing the semantics of a language, whereas these states should be hidden from end users (as we have argued in Sect. 3.2). Moreover, there might even be people which are only interested in an even higher view of a model's behavior; for instance, they might not care about the location of the offers. To suite the needs of these different kinds of users, we plan to extend our approach such that the augmentation and rulestep models can be *refined*. This would allow to start with a specification of the visualization which reveals all execution details, and then to refine that specification step by step, each refinement fulfilling the information needs of a different kind of language users.

References

1. Bandener, N.: Visual Interpreter and Debugger for Dynamic Models Based on the Eclipse Platform. Diploma thesis, University of Paderborn (2009)
2. Bardohl, R., Ermel, C., Weinhold, I.: GenGED – A Visual Definition Tool for Visual Modeling Environments. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 413–419. Springer, Heidelberg (2004)
3. Blunk, A., Fischer, J., Sadilek, D.A.: Modelling a Debugger for an Imperative Voice Control Language. In: Reed, R., Bilgic, A., Gotzhein, R. (eds.) SDL 2009. LNCS, vol. 5719, pp. 149–164. Springer, Heidelberg (2009)

4. Cramer, B., Kastens, U.: Animation Automatically Generated from Simulation Specifications. In: Proceedings of VL/HCC 2009. IEEE Computer Society, Los Alamitos (2009)
5. Eclipse Foundation: Eclipse Modeling Framework, `http://www.eclipse.org/modeling/emf/` (online accessed 9-1-2010)
6. Eclipse Foundation: Graphical Editing Framework, `http://www.eclipse.org/gef/` (online accessed 9–15–2010)
7. Eclipse Foundation: Graphical Modeling Framework, `http://www.eclipse.org/modeling/gmf/` (online accessed 5–5–2009)
8. Eclipse Foundation: UML2 Tools, `http://www.eclipse.org/modeling/mdt/?project=uml2tools` (online accessed 9–15–2010)
9. Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: Dynamic Meta-Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In: Evans, A., Caskurlu, B., Selic, B. (eds.) UML 2000. LNCS, vol. 1939, pp. 323–337. Springer, Heidelberg (2000)
10. Engels, G., Soltenborn, C., Wehrheim, H.: Analysis of UML Activities using Dynamic Meta Modeling. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 76–90. Springer, Heidelberg (2007)
11. Ermel, C., Ehrig, K., Taentzer, G., Weiss, E.: Object Oriented and Rule-based Design of Visual Languages using Tiger. In: Proceedings of GraBaTs 2006. ECEASST, vol. 1, European Association of Software Science and Technology (2006)
12. Geiger, L., Zündorf, A.: eDOBS – Graphical Debugging for Eclipse. In: Proceedings of GraBaTs 2006. ECEASST, vol. 1, European Association of Software Science and Technology (2006)
13. Hausmann, J.H.: Dynamic Meta Modeling. Ph.D. thesis, University of Paderborn (2005)
14. Minas, M., Viehstaedt, G.: DiaGen: A Generator for Diagram Editors Providing Direct Manipulation and Execution of Diagrams. In: Proceedings of VL 1995. IEEE Computer Society, Los Alamitos (1995)
15. Object Management Group: UML Superstructure, Version 2.3, `http://www.omg.org/spec/UML/2.3/` (online accessed 9–15–2010)
16. Petri, C.A.: Kommunikation mit Automaten. Ph.D. thesis, University of Bonn (1962)
17. Rensink, A.: The GROOVE Simulator: A Tool for State Space Generation. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 479–485. Springer, Heidelberg (2004)
18. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation. Foundations, vol. 1. World Scientific Publishing Co., Inc., River Edge (1997)
19. Sadilek, D.A., Wachsmuth, G.: Prototyping Visual Interpreters and Debuggers for Domain-Specific Modelling Languages. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 63–78. Springer, Heidelberg (2008)

Analysing the Cognitive Effectiveness of the BPMN 2.0 Visual Notation

Nicolas Genon[1], Patrick Heymans[1], and Daniel Amyot[2]

[1] PReCISE, University of Namur, Belgium
{nge,phe}@info.fundp.ac.be
[2] University of Ottawa, Canada
damyot@site.uOttawa.ca

Abstract. BPMN 2.0 is an OMG standard and one of the leading process modelling notations. Although the current language specification recognises the importance of defining a visual notation carefully, it does so by relying on common sense, intuition and emulation of common practices, rather than by adopting a rigorous scientific approach. This results in a number of suboptimal language design decisions that may impede effective model-mediated communication between stakeholders. We demonstrate and illustrate this by looking at BPMN 2.0 through the lens of the Physics of Notations, a collection of evidence-based principles that together form a theory of notation design. This work can be considered a first step towards making BPMN 2.0's visual notation more cognitively effective.

1 Introduction

The Business Process Modeling Notation (BPMN) has recently emerged as the industry standard notation for modelling business processes. Originally developed by the Business Process Management Initiative (BPMI), it is now maintained by the Object Management Group (OMG). It aims to provide a common language for modelling business processes, to replace the multiple competing standards that currently exist. As stated in its latest release (BPMN 2.0 [1][1]), BPMN ambitions to "provide a notation that is readily understandable by all business users, from the business analysts that create the initial drafts of the processes, to the technical developers responsible for implementing the technology that will perform those processes, and finally, to the business people who will manage and monitor those processes" [1, p. 28]. Considering the enormous influence that the OMG has in the IT industry, this mission statement holds the promise of delivering a standardised *lingua franca* for all those who, in one way or another, have to deal with business processes.

The standard goes on to say that "[a] key element of BPMN is the choice of shapes and icons used for the graphical elements [...]" and that "[the] intent is to create a standard visual language that all process modelers will recognize and understand" [1, p. 29–30]. From these statements, it is clear that a chief

[1] This analysis was performed on version 0.9.15.

concern of BPMN 2.0 is *cognitive effectiveness* – although the standard does not use that term. Cognitive effectiveness is defined as the *speed*, *ease* and *accuracy* with which a representation can be processed by the human mind [2,3]. A major incentive for using visual notations is the widely-held belief that they convey information more effectively than text, especially to novices [4]. However, cognitive effectiveness is not an intrinsic property of visual notations but something that must be designed into them [3,5].

The proponents of BPMN 2.0 somehow acknowledged this when they state that "[they have] brought forth expertise and experience with many existing notations and [have] sought to consolidate the best ideas from these divergent notations into a single standard notation" [1, p. 28]. However, a striking thing when reading through the voluminous document (504 pages) is the lack of *design rationale*, that is, explicit reference to theories and empirical evidence for designing effective visual notations.

To be perfectly fair to BPMN 2.0, we should recognise that it actually does *better* than many other language definitions, in at least two respects. Firstly, its notation has been defined after reviewing a large number of other notations, including UML Activity Diagrams, UML EDOC Business Processes, IDEF, ebXML BPSS, ADF, RosettaNet, LOVeM, and EPC [1, p. 28]. Secondly, it gives a thorough description of its graphical notation, including an extensive list of the available symbols, how they can be combined and how they can (and cannot) be customised for domain- or application-specific usages.

Still, BPMN 2.0 does not justify its notation design choices by referring to explicit principles. This is actually not surprising since this is a common practice in visual notation design [2]. In this respect, BPMN 2.0 does neither better nor worse than the vast majority of notations: it relies on common sense, intuition and emulation of common practice, rather than adopting a rigorous *scientific* approach.

The objective of this paper is to conduct an analysis of the current BPMN 2.0 visual notation that is based on theory and empirical evidence rather than common sense and opinion. Of course, it is always easy to criticise, but our aim in conducting this analysis is *constructive*: to provide an independent analysis of the strengths and weaknesses of the BPMN visual notation that can be used to improve its usability and effectiveness in practice, especially for communicating with business users. We believe that a unified business process modelling notation is an important innovation and our aim is to help remove potential barriers to its adoption and usage in practice.

A broader goal of this paper is to increase awareness about the importance of visual representation in business process modelling, and the need to refer to theory and empirical evidence in defining notations (evidence-based design practice). Accordingly, our approach enriches emerging research in the area (see Section 2.3). Visual syntax has a profound effect on the effectiveness of modelling notations, equal to (if not greater than) decisions about semantics [6]. But, in contrast to process modelling semantics [7,8], the analysis and definition of process modelling visual notations is a much less mature discipline. Our intention is to remedy this situation.

This paper is structured as follows: Section 2 presents the research background we build upon, and in particular the Physics of Notations, i.e., the theory against which we evaluate BPMN. Section 3 reports on the analysis itself. Section 4 puts the results in a broader perspective and provides a summary of the paper.

2 Previous Research

2.1 Language Evaluation Frameworks

The Cognitive Dimensions of Notations (CDs) framework defines a set of 13 dimensions that provide a vocabulary for describing the structure of cognitive artefacts [9]. This has become the predominant paradigm for analysing visual languages in the IT field. CDs have a level of genericity/generality that makes them applicable to various domains, but it also precludes specific predictions for visual notations. In [9], Green *et al.* noted that the dimensions are vaguely defined and that "the lack of well-defined procedure disturbs some would-be users". These limitations as well as other disadvantages discussed by Moody [10] support our choice not to base our analysis on CDs.

Also popular is the semiotic quality (SEQUAL) framework [11]. It proposes a list of general *qualities* for models and modelling languages, that it organises along the *semiotic ladder* (i.e., the scale 'physical', 'empirical', 'syntactic', 'semantic', 'pragmatic' and 'social'). SEQUAL also distinguishes quality *goals* from the *means* to achieve them, and sees modelling activities as socially situated (*constructivistic worldview*). Essentially, SEQUAL offers a comprehensive ontology of model and modelling language quality concepts. It provides a precise vocabulary and checklist when engaging in a comprehensive analysis. The part of SEQUAL that is most closely related to notation quality is termed 'comprehensibility appropriateness' [12]. However, for our purpose, SEQUAL shares a number of important limitations with CDs (although the two frameworks are very different in intent and content). The two main limitations are the level of generality and the lack of theoretical and empirical foundations related to visual aspects of notations.

2.2 The Physics of Notations

The Physics of Notations theory [2] provides a framework that is *specifically* developed for *visual notations*. It defines a set of 9 evidence-based principles to evaluate and improve the visual notation of modelling languages. The principles are clearly defined and operationalised using evaluation procedures and/or metrics. They are synthesised from theory and empirical evidence stemming from various scientific disciplines such as cognitive and perceptual psychology, cartography, graphic design, human computer interface, linguistics, and communication. This theory is falsifiable [13], i.e., the principles can be used to generate predictions, which are empirically testable. So far, the Physics of Notations has been used to evaluate the visual notations of Archimate [14], UML [15], *i** [16] and UCM [17].

The 9 principles are:

1. *Semiotic Clarity*: there should be a one-to-one correspondence between semantic constructs and graphical symbols.
2. *Perceptual Discriminability*: symbols should be clearly distinguishable.
3. *Visual Expressiveness*: use the full range and capacities of visual variables.
4. *Semantic Transparency*: use symbols whose appearance is evocative.
5. *Complexity Management*: include mechanisms for handling complexity.
6. *Cognitive Integration*: include explicit mechanisms to support integration of information from different diagrams.
7. *Dual Coding*: enrich diagrams with textual descriptions.
8. *Graphic Economy*: keep the number of different graphical symbols cognitively manageable.
9. *Cognitive Fit*: use different visual dialects when required.

Operationalisations of the principles often rely on values of *visual variables*, i.e., the elementary characteristics forming the visual alphabet of diagrammatic notations. The seminal work of Bertin [18] identified 8 visual variables divided into two categories: planar and retinal variables (see Figure 1). Essentially, symbols are obtained by combining visual variable values. Henceforth, we take the convention of underlining visual variable names.

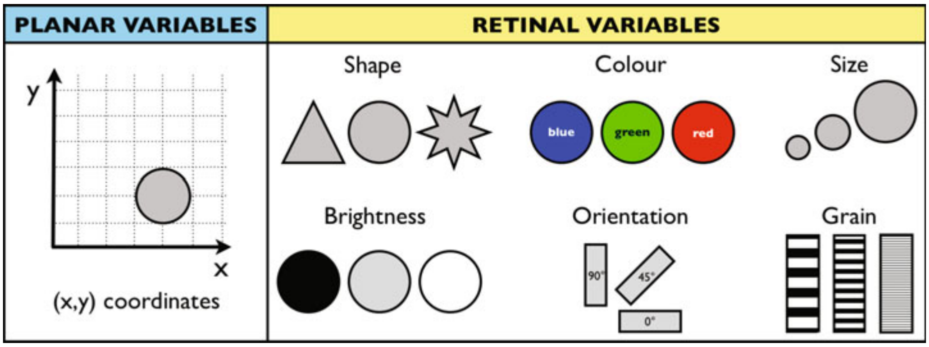


Fig. 1. The 8 visual variables from Bertin [18]

2.3 Visual Aspects of Process Modelling Notations

Studies of the visual syntax of process modelling languages are emerging. Some are concerned with making improvements at the *diagram level* [19,20], whereas our work makes observations and suggestions at the *language level* based on the Physics of Notations. Our work thus focuses on defining notations that are cognitively effective *by construction*. This, of course, does not preclude diagram level improvements (which we actually support) by using so-called *secondary notation* (see Section 3.7). Empirical research has also started to study the impacts of language and context characteristics, such as diagram layout and user expertise [21], routing symbol [22], dual use of icons and labels for process activities [23], ease of use of the language and user experience [24] as well as modularity [25]. Such empirical research is important as it can falsify or corroborate predictions from theories such as the Physics of Notations. However, in the case of BPMN 2.0, we think that application of the theory must come *first* (see Section 4). Nevertheless, we took those studies into account for our analysis of BPMN 2.0, as well work by zur Muehlen and Recker [26] who studied which BPMN concepts are the most frequently used by modellers.

3 Analysis of BPMN 2.0 Process Diagrams

BPMN 2.0 consists of four[2] types of diagrams: PROCESS, CHOREOGRAPHY, COLLABORATION and CONVERSATION diagrams. Process diagrams are by far the most important. The scope of our analysis is limited to them. The 9 principles of the Physics of Notations were thus used to conduct a systematic, symbol-by-symbol analysis of process diagrams. The main findings are reported in the following subsections, organised by principle. For each principle, we provide a definition, summarise the results of the evaluation (how well BPMN satisfies the principle), and give recommendations for improvement. However, we do not go as far as defining a complete new notation which would be overly ambitious and premature. The full analysis can be found in [27].

3.1 Cognitive Fit

As stated by Vessey in the theory of Cognitive Fit [28], there should be a 3-way fit between the *audience* (sender and receiver), the *task characteristics* (how and for what purpose(s) the notation is used) and the *medium* on which the information is represented. BPMN's aim is to "provide a notation that is readily understandable by all business users, from the business analysts that create the initial drafts of the processes, to the technical developers responsible for implementing the technology that will perform those processes, and finally, to the business people who will manage and monitor those processes" [1, p. 28]. However, Cognitive Fit theory suggests that trying to design a language that is "all things to all men" is likely to be an impossible mission. Instead, different representations should be used for different tasks, audiences and/or media. BPMN process diagrams can be used in various contexts, e.g., to communicate with non-technical stakeholders, as visual representations of information to be processed by software, etc. The relative importance of the principles differs from one context to another. Different versions (dialects) of a same notation can even be defined for specific contexts. In practice, Cognitive Fit is usually analysed according the two following points of view.

Expert-Novice Differences. There are well-known differences in the way experts and novices create and use diagrams. According to [2], the most important differences are:

[2] The standard does not clearly state if CONVERSATION is a type of diagram or a view on COLLABORATION diagrams.

- Novices have more difficulties discriminating between symbols [4,29].
- Novices have more difficulties remembering what symbols mean [30].
- Novices are more affected by complexity, as they lack mental "chunking" strategies [31].

Concretely, when the purpose of a diagram is to reach a shared high-level understanding of a process among an audience of *novices* (non-technical stakeholders as end users, business experts), then it is particularly important to have a language with few symbols (Graphic Economy) that are mnemonic (Semantic Transparency) and quickly distinguishable (Perceptual Discriminability). Moreover, such diagrams should be kept simple through Complexity Management and Cognitive Integration. On the contrary, if a language is to support detailed discussion between *experts*, or if it is to be fed into a workflow engine, it is more important to have a comprehensive set of symbols with clear semantics (Semiotic Clarity), to be able to represent all required extra details through text (Dual Coding) and to be able to structure large models through Complexity Management and Cognitive Integration. Of course, many other usage contexts could be envisioned, especially when diagrams are devoted to a particular and well-defined task.

In this paper, we chose to illustrate the principles with examples that target an audience of novices. We did so because we think that the BPMN 2.0 notation is more challenging for novices than for experts.

The Differences in Representational Media also support the utilisation of several visual dialects. Rendering diagrams in a computer-based editing tool or drawing them by hand on a whiteboard call for distinct skills. Sketching on a whiteboard requires good drawing abilities, which cannot be assumed from everybody. Indeed, IT practitioners cannot be assumed to be skilled in graphic design. Therefore, sophisticated icons and complex geometrical shapes must be avoided in this kind of notation. Computer-based tools typically do not require such skills (symbols are 'dragged-and-dropped' from a menu onto the diagram) and can easily produce sophisticated visual objects. However, the whiteboard remains a better support for collaborative and interactive modelling [32].

The most common BPMN symbols (see Figure 2A) are basic geometrical shapes. They are easy to draw by hand which is convenient for use on whiteboards, flip charts as well as for paper sketches.

3.2 Semiotic Clarity

According to Goodman's theory of symbols [33], for a notation to satisfy the requirements of a notational system, there should be a one-to-one correspondence between symbols and their referent concepts. Hence, our first task was to inventorise all *semantic constructs* and all visual symbols of process diagrams. At first sight, the list of semantic constructs can be approximated by the list of concrete metaclasses in the BPMN metamodel. However, this is not 100% reliable due to variations in metamodelling styles. A common example is when a construct subtype (e.g., ExclusiveGateway, a subtype of Gateway) can be encoded either as a subclass, or as a particular attribute value of the base metaclass (i.e., GATEWAY). Unless explicitly stated in the standard, this leads to subjective decisions (based on document context and common sense) as to which metamodel elements ought to be considered semantic constructs. We had to make 160 such decisions, adopting consistent policies for similar cases. We eventually counted 242 semantic constructs in process diagrams. Symbols, on the other hand, correspond to legend entries and are thus more straightforward to identify. Process diagrams have 171 symbols, a small sample of which appears in Figure 2. The exhaustive list of all semantic constructs and symbols we took into account is available in [27]. An immediate observation is that these numbers are huge: one or two orders of magnitude beyond languages such as ER (5 symbols), DFD (4 symbols) or YAWL (14 symbols).

Once these numbers are established, Semiotic Clarity can be assessed. Goodman's theory of symbols pinpoints four types of anomalies (analogous to ontological anomalies [7]) that can occur:

- *Symbol deficit*: a construct is not represented by any symbol.
- *Symbol redundancy*: a single construct is represented by multiple symbols.
- *Symbol overload*: a single symbol is used to represent multiple constructs.
- *Symbol excess*: a symbol does not represent any construct.

Semiotic Clarity maximises *expressiveness* (by eliminating symbol deficit), *precision* (by eliminating symbol overload) and *parsimony* (by eliminating symbol redundancy and excess) of visual notations.

We obtained the following results: 23.6% symbol deficit, 5.4% symbol overload, 0.5% symbol excess and 0.5% symbol redundancy. The latter two are negligible. Symbol deficit is the biggest anomaly. It has diverse causes, including domain-specific metaclasses that do not yet have a notation (e.g., AUDITING) or apparent omissions. Symbol overload appears for some constructs, like GATEWAY, which can be represented equally by a diamond or a crossed diamond, with no associated semantic distinction. We see no particular difficulties in removing those anomalies at the notational level, provided that semantic constructs are established. Determining whether modifications to the semantics are required or not is beyond the scope of the Physics of Notations.

3.3 Perceptual Discriminability

Perceptual Discriminability is defined as the ease and accuracy with which different symbols can be differentiated from each other. Discriminability is determined by the visual distance between symbols, which is measured by the number of visual variables on which they differ and the size of these differences (number of perceptible steps between the values of a visual variable). Perceptual Discriminability is a prerequisite for accurate interpretation of diagrams [34]. The greater the visual distance between symbols, the faster and the more accurately they will be recognised [30]. When differences are too subtle, symbol interpretation is much inaccurate. This is especially crucial for novices who have higher requirements for discriminability than experts [4].

Discriminability is a two-step mental process: firstly, symbols are distinguished from the background. This step is called *figure-ground segregation*. Then symbols of different types are discriminated from each other. This second step, called *symbol differentiation*, relies on pair-wise value variations of visual variables between symbols : Shape plays a privileged role in this process, as it represents the primary basis on which we classify objects in the real world [35]. In BPMN process diagrams, 4 shapes are used to derive the majority of symbols. Variations are introduced by changing border style and thickness, and by incorporating additional markers. A selection of these symbols appears in Figure 2A. All shapes are 2-D and from one of two shape families: ellipses (incl. circle) and quadrilaterals (incl. "roundtangle", rectangle and diamond). Ideally, shapes used to represent different semantic concepts should be taken from different shape families to minimise the possibility of confusion [35].

Size is another factor that influences discriminability. Large symbols take more space on the reader's field of vision and focus attention. The size of process diagram symbols is not specified in BPMN 2.0. In practice, the size depends on the amount of text inside the symbol and is thus not related to its semantics. Since Activity appears to be the main symbol type in process diagrams, we would suggest making it explicitly bigger than the 3 other types. Further variations in Size can be used as a secondary notation [21].

Grain (aka. Texture) determines the style and thickness of shape borders. This visual variable can also facilitate symbol differentiation. All BPMN border styles for Events and Activities are shown in Figure 2A. Grain is used to discriminate between 5 types of Events and 4 types of Activities. All 5 visual variable values are distinct, which is a good point, but they quickly become hard to discern when zooming out on the diagram: double lines are merged into a single thick line and dotted/dashed lines become solid lines. Even if this issue is not specific to BPMN, it remains an obstacle to effective discriminability.

Colour is one of the most cognitively effective visual variables: the human visual system is highly sensitive to variations in colours and can quickly and accurately distinguish between them [30]. However, if not used carefully, Colour can undermine communication. In BPMN 2.0, the use of colours is mainly up to tool developers [1, p. 29] (with the exception of the cases discussed in Section 3.7). Nevertheless, the Physics of Notations argues that the choice of colours should be justified by the existence of some sort of association (e.g., logical, metaphorical, rhetorical, cultural) between a symbol and the concept(s) it represents. Additionally, Colour can be used to achieve *redundant coding*. Redundancy is a well-known technique to thwart noise and preserve the signal from errors [36]. Applied to our context, it consists in making symbols distinguishable through concomitant use of several visual variables. Redundant coding is achieved when the value of each visual variable (taken individually) is *unique* among all symbols of the notation.

Based on the above considerations, we illustrate how Perceptual Discriminability can be enhanced in practice, using symbols from BPMN 2.0 (see Figure 3). These new versions of the symbols are motivated by several

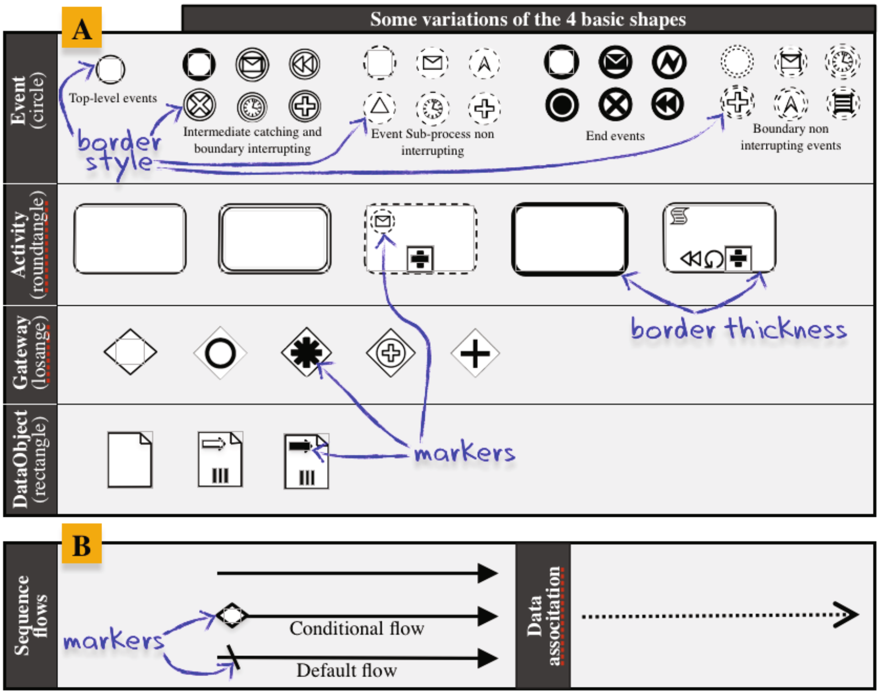


Fig. 2. Basic shapes and connection objects of BPMN 2.0 process diagrams

principles of the Physics of Notations, so further justifications will be provided in the appropriate sections. Here we focus on the rationale related to Perceptual Discriminability:

– We increased discriminability by using 3 shape categories: ellipses, quadrilaterals and cylinders. Further justification of the shapes appears under Semantic Transparency (Section 3.4).
– Border <u>Grain</u> is kept *as is*. It is hard to find another style of border that is easy to draw by hand. However, we propose to improve symbol discriminability by using <u>Colour</u> (see Semantic Transparency).

So far, we have focused on BPMN flow objects, i.e., EVENT, ACTIVITY and GATEWAY. Relationships between FLOW OBJECTS are called CONNECTING OBJECTS in the BPMN jargon. In BPMN process diagrams, there are two types of relationships: SEQUENCE FLOW and DATA ASSOCIATION[3]. As shown in Figure 2B, they are represented by monochrome arrows. SEQUENCE FLOWS can have additional markers at the source, if they are constrained by a condition or if they are the default output of a GATEWAY. The two representations differ on their <u>Grain</u>: SEQUENCE FLOWS are solid lines whereas DATA ASSOCIATIONS are

[3] The BPMN standard defines MESSAGE FLOW as an element of COLLABORATION diagrams. Hence they are not considered in this work.

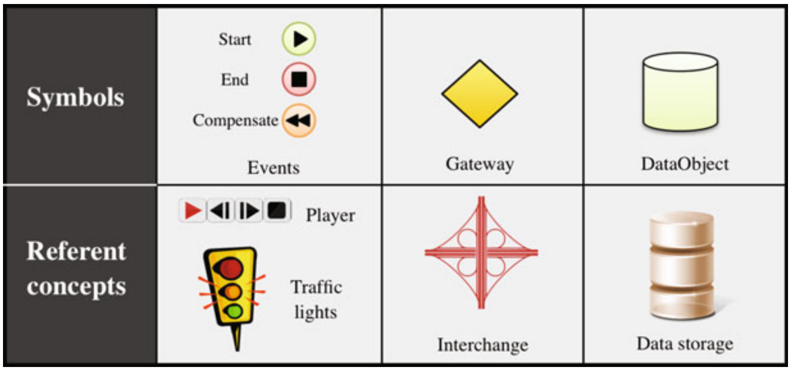


Fig. 3. More semantically transparent symbols proposed for BPMN

dotted. This way, they are distinguishable because they have unique values on at least one visual variable. We should also note that the arrowheads of these two types of flows are distinct (Shape), which reinforces their discriminability. Colour and Grain could also be used to improve their differentiation by introducing redundant coding.

3.4 Semantic Transparency

Symbols should provide cues to their meaning. Semantically direct representations reduce cognitive load through built-in mnemonics: their meaning can be either perceived *directly* or *easily learned* [37]. Such representations speed up recognition and improve intelligibility, especially for novices [4,38]. A symbol's semantic transparency is not a binary state: the transparency level is in a range that goes from semantically *immediate* – the referent concept is understood immediately – to semantically "*perverse*" – the symbol suggests a different meaning.

In BPMN 2.0 process diagrams, symbols are *conventional shapes* on which iconic markers are added (Figure 2A). Symbol shapes seem not to convey any particular semantics: there is no explicit rationale to represent an Event as a circle, an Activity as a roundtangle and a Gateway as a diamond. The situation is even worse for DataObject: its symbol suggests a "sticky note" (a rectangle with a folded corner). This icon is typically used for comments and textual annotations (e.g., in UML), not for first-class constructs. DataObject is thus a case of semantic perversity. The differentiation of Event and Activity subtypes is also purely conventional: it depends on styles of border that are not perceptually immediate.

This lack of semantic immediacy is particularly puzzling, as one of the *leitmotivs* of BPMN is its simplicity of use for novices. Figure 3 proposes more semantically immediate shapes. These shapes are not meant to be the best alternative to the current BPMN 2.0 notation. They are only demonstrations of potential improvements. The suggested Event symbols are inspired from YAWL [8], where they represent types of Conditions using icons inspired from a video player metaphor. Our proposal reuses the BPMN EVENT circle filled with traffic light colours and adorned with player icons. The GATEWAY symbol keeps its diamond shape which suggests the idea of an interchange road sign (yellow diamond). A cylinder has been chosen to represent DATAOBJECT. It stands out from other symbols by being 3-D, thereby also improving discriminability. The cylinder reuses the symbol usually depicting databases or data storage devices, which are in the end closely related.

Regarding Colour, we already justified the choices made for EVENTS. The choice of Colour values should always be based on the nature of the elements represented by the symbols. In practice, this often results from a trade-off that takes into account Symbol and Background Discriminability as well as Semantic Transparency.

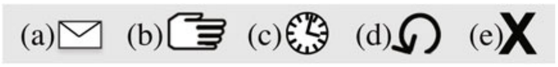


Fig. 4. Semantically transparent BPMN icons

The second element that makes BPMN symbols vary is the use of markers. These markers are *icons* (also called mimetic symbols or pictographs) because they perceptually resemble the concepts they represent [39]. Empirical studies show that replacing abstract shapes with icons improves understanding of models by novices [38]. They also improve likeability and accessibility: a visual representation appears more daunting to novices if it is comprised only of abstract symbols [37,40].

BPMN 2.0 includes a large repertoire of icons (more than 25). While the theory recommends *replacing* conventional shapes by icons, BPMN makes a different use of them. They are added *inside* symbols instead of *being the symbols.* Part of the icon repertoire allows distinguishing between subtypes of the 4 basic semantic constructs, while the other part represents attribute values of these constructs. The Semantic Transparency of these icons varies: Figure 4 shows semantically immediate icons that respectively mean: (a) *message*, (b) *manual*, (c) *timer*, (d) *loop* and (e) *exclusive.* If symbols do not appear evocative to novices, once they have been learned, these icons are easily remembered.

On the contrary, Figure 5A illustrates semantically opaque, and in some cases "perverse", icons from BPMN 2.0. The pentagon does not suggest any obvious meaning. In relation to EVENT triggers, it actually means *multiple.* The second icon represents a kind of lightning and could refer to something happening suddenly like an event. In fact, it signifies *error.* The third icon is particularly opaque and even misleading, i.e., it does not mean list but *condition.* The 2 gears, that usually suggest the idea of process or task, are also a case of perversity: this icon refers to the concept of *service* (e.g., web service). The last icon resembles a data sheet, but stands for *business rule.*

In Figure 5B, we suggest new icons that, following the Physics of Notations, are more semantically immediate. As discussed previously, the level of semantic transparency depends on several factors and what seems immediate to someone can remain opaque to somebody else. This suggestion is therefore by no means definitive. The Service icon is a waiter carrying a tray and the Business Rule icon is now composed of a judge's hammer (meaning "rule") filled with a dollar symbol (meaning "business"). Consequently, and even if they may appear semantically perverse at first sight to some people, these icons would probably become immediate for novices as soon as their rationale is *explained*.

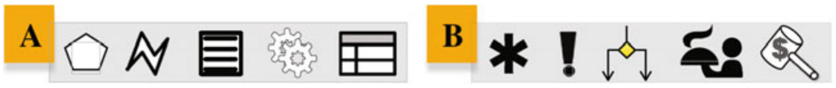


Fig. 5. (A) Semantically opaque icons and (B) More semantically transparent icons

Binary directed relationships are classically represented by arrows that are essentially unidimensional (1-D) visual objects. Therefore, only a small portion of the design space is available to achieve Semantic Transparency. Arrowheads are a slight incursion in the 2-D world to show direction. Additionally, the source anchor of SEQUENCE FLOWS can be adorned with a GATEWAY or DEFAULT marker (Figure 2B, left). The GATEWAY marker is rather well chosen whereas the DEFAULT marker is purely conventional. The latter could be removed and the default flow could be indicated by a larger Grain. This technique is used for priority road signs, for example. The representation of DATA ASSOCIATION (see Figure 2B, right) is not semantically immediate but this quality seems hard to achieve for this symbol.

3.5 Complexity Management

One of the major flaws of visual notations is their diagrammatic complexity, which is mainly due to their poor scaling capability [41]. This complexity is measured by the number of elements displayed on a diagram. The degree of complexity management varies according to the ability of a notation to represent information without overloading the human mind. The two main solutions to decrease diagrammatic complexity are *modularisation* and *hierarchic structuring*.

Modularisation consists in dividing complex diagrams into manageable chunks. The decomposition can be *horizontal* or *vertical*. While horizontal decomposition takes place at the same level of abstraction, vertical decomposition produces finer grained sub-diagrams.

BPMN 2.0 provides several mechanisms to manage diagrammatic complexity. First, it supports modelling along 4 different viewpoints that correspond to the 4 types of diagrams: PROCESS, CHOREOGRAPHY, COLLABORATION and CONVERSATION. In a diagram, only the information relevant to the chosen viewpoint has to be represented. BPMN process diagrams achieve modularity thanks to 2 constructs (see Figure 6): (a) LINK EVENTS are used as *intra* or *inter* diagram connectors. They support horizontal decomposition. A Link Event comes as a pair of symbols: a black arrow indicates the source while a white arrow represents the target. Naming the pairs facilitates their association when there are several instances on the same diagram. (b) SubProcesses are self-contained parts of a process. They allow to vertically decompose a diagram in two levelled views: a high-level view – collapsed subprocess – and a fine-grained view – expanded subprocess.

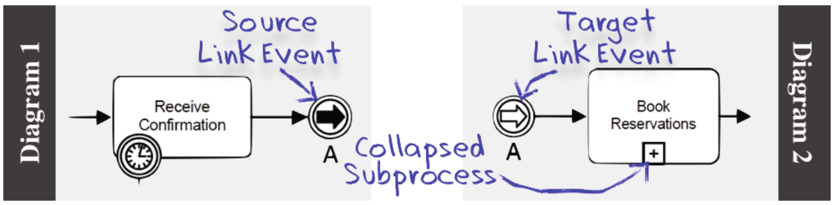


Fig. 6. Modularisation with Link Events (example from the BPMN 2.0 spec)

Hierarchic Structuring is one of the most effective ways of organising complexity for human comprehension. It allows systems to be represented at different levels of detail, with manageable complexity at each level [42]. SubProcesses provide a means for hierarchic structuring. Yet, to be effective, different levels of information should be displayed in independent diagrams instead of expanding into their parent diagram (see hierarchical *vs.* inline expansion in [43]).

3.6 Cognitive Integration

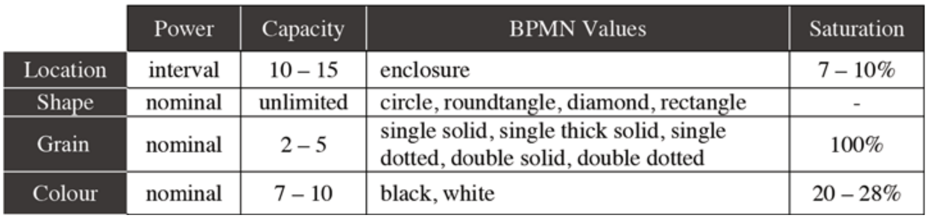
Large system representations cannot fit into a single diagram. The information is spread across several diagrams and the reader needs to mentally integrate all these pieces of knowledge. Cognitive Integration helps making this integration easier. It takes place at two levels: perceptual integration and conceptual integration. *Perceptual integration* refers to cues that simplify navigation and transitions between diagrams. *Conceptual integration* addresses the assembly of information from separate diagrams into a coherent mental representation.

While Complexity Management leads to the multiplication of diagrams, no technique is available in BPMN to reinforce perceptual or conceptual integration. Mechanisms such as diagram level numbering, signposting and navigation maps [44] could improve perceptual integration. Contextualisation information [45,46] and summary diagrams [47] enhance conceptual integration. Concretely, the notation should ensure that modellers could name their diagrams and number them according to their level in the hierarchic structure. A navigation map could be created based on Link Events and SubProcesses. Contextualisation is partially achieved as expanded SubProcesses are integrated into their parent Activity.

3.7 Visual Expressiveness

Visual Expressiveness measures the extent to which the graphic design space is used, i.e., the number of visual variables used by a notation and the range of values for each variable. While Perceptual Discriminability focuses on pair-wise visual variation between symbols, Visual Expressiveness measures visual variations across the entire visual vocabulary. Variables that encode information are called information-carrying variables and compose the *primary notation*. The other variables, called the free variables, form the *secondary notation* [21]. They allow modellers to reinforce or clarify the meaning of diagram elements.

Primary Notation. The BPMN process diagram notation uses half of the visual variables: Location (x,y), Shape, Grain and Colour carry semantic information, while Size, Orientation and Brightness are relegated to the secondary notation. Visual variables also have to be chosen according to the type of information to encode. Figure 7 summarises the *power* (highest level of measurement that can be encoded), the *capacity* (number of perceptible steps), the BPMN *values* and the *saturation* (range of values / capacity) of each information-carrying variable.



	Power	Capacity	BPMN Values	Saturation
Location	interval	10 – 15	enclosure	7 – 10%
Shape	nominal	unlimited	circle, roundtangle, diamond, rectangle	-
Grain	nominal	2 – 5	single solid, single thick solid, single dotted, double solid, double dotted	100%
Colour	nominal	7 – 10	black, white	20 – 28%

Fig. 7. Design space covered by the BPMN process diagram notation

We observe that visual variables in BPMN were chosen appropriately according to the nature of information, which here is purely nominal (i.e., there is no ordering between values). Location can actually be used to encode *intervals* but it is used in BPMN only for enclosure (a symbol is contained in another symbol), which is only a small portion of its capacity. Visual variable capacities are rather well exploited and Grain is even completely saturated. However, as we discussed in Section 3.3, this causes discriminability problems. The perceptible steps between Shape values are a major problem of the current notation. Current shapes belong to only two categories (circles and quadrilaterals), whereas there is no semantic relationship between the referent concepts within a shape category. We have already illustrated possible solutions to this problem (see Figure 3).

Colour is one of the most cognitively effective of all visual variables. BPMN 2.0 specification states that “Graphical elements may be colored, and the coloring may have specified semantics that extend the information conveyed by the element as specified in this standard” [1, p. 30]. In fact, BPMN uses only two colours

– black and white – that allow distinguishing between "throwing" (filled) and "catching" (hollow) markers. Hence, the Colour capacity is underused. Improvement proposals have been made for this, e.g., the EVENT symbols in Figure 3.

Secondary Notation consists of the visual variables that do not carry semantic information, called free variables. At first sight, secondary notation could be thought of as a low priority matter. However, it proves to be of utmost importance when notation engineers face one of the two following situations. The first is when there is a need for introducing a new type of information in the language (e.g., if Semiotic Clarity requires a new symbol to be added in the notation). The language engineer should then first consider free variables before overloading a visual variable that already belongs to the primary notation. This is also the case if the language engineer wants to achieve redundant coding as described in Section 3.3. The second situation occurs when modellers need to place visual cues or hints on the diagram to improve its understandability. Most such cues are *cognitive helpers* that should be implemented with the secondary notation. The main reason is that they do not carry language information, so they should not compete with existing information-carrying variables.

In practice, cognitive helpers are often defined when designing CASE tools. But CASE tool developers usually implement these helpers differently for each tool, resulting in *non standard* solutions even for the same modelling language. To be effective, CASE tools should propose helpers defined by the notation and based on theory and evidence.

3.8 Dual Coding

So far, text has not been considered as an option for encoding information (see Perceptual Discriminability and Visual Expressiveness). However, this does not mean that text has no place in visual notations. According to Dual Coding theory [48], using text and graphics together to convey information is more effective than using either on their own. BPMN makes limited use of Dual Coding. It does so for CONDITIONAL and COMPLEX GATEWAYS only. Labels accompany the ALTERNATIVE or CONDITIONAL FLOWS as appearing in Figure 8. Although we did not observe any major issue with current uses of Dual Coding in BPMN, we suggest to further explore the usage of text in order to improve Graphic Economy, which is BPMN's major problem as discussed in the next section. This contrasts with a recent proposal [49] where Dual Coding is achieved at the expense of Graphic Economy by adding 25 new iconic markers.

3.9 Graphic Economy

Graphic complexity refers to the size of the visual vocabulary, i.e., the number of symbols in a notation [50]. It is measured by the number of legend entries required. This differs from diagrammatic complexity as graphic complexity focuses on the language (type level) rather than the diagram (token level). Graphic Economy seeks to reduce graphic complexity. It is a key factor for cognitive effectiveness since humans' span of absolute judgement when discriminating visual alternatives is around 6 [51]. It can be higher for experts though.

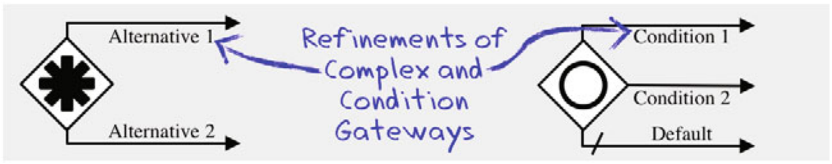


Fig. 8. Dual Coding in BPMN process diagrams

BPMN 2.0 process models have a graphic complexity of 171. This is at least an order of magnitude beyond novice capabilities (28 times the aforementioned limit). zur Muehlen and Recker observe that, in practice, the graphic complexity of BPMN is significantly lower than its nominal complexity [26]. Their study shows that most process diagrams designed for novices use only basic symbols: Event, Activity, Gateway, Sequence Flow, DataObject and Association, plus a few refinements. The *practical* complexity is thus around 10. This is certainly much more manageable than the full language, but it is still high compared to popular languages [52] such as ER diagrams (complexity of 5) and DFDs (complexity of 4). YAWL, which is more closely related to BPMN, has a complexity of 14. Moreover, the question remains open for BPMN experts: do we really need 171 symbols, even when the goal is to produce detailed models for other experts or for execution in workflow engines? A study similar to that of zur Muehlen and Recker is necessary for such usage contexts too. It could make the case for introducing *symbol deficit* [2], i.e., choosing *not* to provide symbols for some seldom used constructs. Those can still be represented separately using text as suggested in Dual Coding, similar to integrity constraints in ER. It might also be useful to check BPMN for semantic redundancies that could be factored out. Such semantic analyses are beyond the scope of this paper, but at the notation level it is still possible to improve Perceptual Discriminability, Semantic Transparency and Visual Expressiveness as discussed in the previous sections.

4 Discussion and Conclusions

Defining a *cognitively effective* notation is a time-consuming activity governed by conflicting goals. As shown in the analysis presented in this paper, BPMN tries to strike a balance between such goals. But we have argued that it does so in a suboptimal way due to its lack of consideration for existing concepts and scientific principles of notation design. This first complete analysis[4] of BPMN 2.0 against the Physics of Notations theory reveals various problems in the BPMN notation, suggests some improvements, but most importantly recommends a change of methodology.

Given the effort that this would require, we did not go as far as defining a new notation. Our various suggestions are thus not meant to be definitive or consistent with each other. Yet, we deem that they have a value in illustrating

[4] The full analysis is available as a technical report [27].

"outside-of-the-box" thinking about process modelling notations. We hope they will be regarded as sources of *inspiration* and *debate* by the BPM(N) community.

Our analysis (in particular that of Cognitive Fit) also suggests that aiming at a notation that is perfect for all audiences and tasks is utopian. There is no silver bullet. An important question for the future is thus whether there should be one or multiple dialects of BPMN. Silver [43] seems to support this idea with his 3-level methodology for BPM process modelling, using 3 dialects of the BPMN notation. Empirical studies [26] also suggest to restrict the BPMN symbol set when used by novices. But, the *quantity* of symbols (Graphic Complexity) is only one of the dimensions to act upon. For example, in a recent similar analysis of the *i** goal modelling notation [16], Moody *et al.* suggested two dialects, one (for hand sketching) being a *qualitatively* simplified version of the other (for computer-based editing).

An obvious limitation of our research (and the Physics of Notations) is that it focuses only on syntactic issues, whereas solving some of the identified problems (especially the huge number of constructs and symbols) partly requires re-examining the semantics. Another limitation of our work is the lack of empirical validation of *our* suggestions with real BPMN users. This is mitigated by the fact that they are based on theory and empirical evidence synthesised in the Physics of Notations. Moreover, we argue that it would be premature to empirically test these ideas at this stage as they are only *our* suggestions and are not yet fully developed. More work is needed to explore alternative solutions, preferably with participation from BPMN users and researchers. Finally, we acknowledge that there is much legacy related to BPMN as version 1.2 is already used in practice, with support from dozens of commercial tools, some of which cover additional elements from BPMN 2.0. Our contributions may have a limited impact on several legacy symbols but they certainly apply to the numerous new concepts and symbols found in version 2.0, to future versions of BPMN, and to other related languages.

Acknowledgement. The authors are most thankful to Daniel L. Moody for fruitful discussions. This work was supported in part by the Interuniversity Attraction Poles Programme - Belgian State - Belgian Science Policy (MoVES). NSERC provided funding to D. Amyot in support of this collaboration.

References

1. OMG: Business Process Model and Notation (BPMN) Specification 2.0 V0.9.15. Object Management Group Inc. (2009)
2. Moody, D.L.: The "Physics" of Notations: Towards a Scientific Basis for Constructing Visual Notations in Software Engineering. IEEE Transactions on Software Engineering 35, 756–779 (2009)
3. Larkin, J., Simon, H.: Why a Diagram Is (Sometimes) Worth Ten Thousand Words. Cognitive Science 11, 65–99 (1987)
4. Britton, C., Jones, S.: The Untrained Eye: How Languages for Software Specification Support Understanding by Untrained Users. Human Computer Interaction 14, 191–244 (1999)

5. Kosslyn, S.M.: Graphics and Human Information Processing: A Review of Five Books. Journal of the American Statistical Association 80(391), 499–512 (1985)
6. Hitchman, S.: The Details of Conceptual Modelling Notations are Important - A Comparison of Relationship Normative Language. Communications of the AIS 9(10) (2002)
7. Recker, J., Rosemann, M., Indulska, M., Green, P.: Business Process Modeling - a Comparative Analysis. Journal of the Association for Information Systems (JAIS) 10(4), 333–363 (2009)
8. van der Aalst, W., ter Hofstede, A.: YAWL: Yet Another Workflow Language. Information Systems Journal 30(4), 245–275 (2005)
9. Green, T., Blandford, A., Church, L., Roast, C., Clarke, S.: Cognitive Dimensions: Achievements, New Directions, and Open Questions. Journal of Visual Languages and Computing 17, 328–365 (2006)
10. Moody, D.L.: Theory Development in Visual Language Research: Beyond the Cognitive Dimensions of Notations. In: Proc. of the IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC 2009), pp. 151–154 (2009)
11. Krogstie, J., Sindre, G., Jorgensen, H.: Process Models Representing Knowledge for Action: a Revised Quality Framework. European Journal of Information Systems 15, 91–102 (2006)
12. Krogstie, J., Solvberg, A.: Information systems engineering - conceptual modeling in a quality perspective. In: Kompendiumforlaget, Trondheim, Norway (2003)
13. Popper, K.R.: Science as Falsification. In: Routledge, Keagan, P. (eds.) Conjectures and Refutations, London, pp. 30–39 (1963)
14. Moody, D.L.: Review of archimate: The road to international standardisation. Technical report, Report commissioned by the ArchiMate Foundation and BiZZDesign B.V, Enschede, The Netherlands (2007)
15. Moody, D., van Hillegersberg, J.: Evaluating the visual syntax of UML: An analysis of the cognitive effectiveness of the UML family of diagrams. In: Gašević, D., Lämmel, R., Van Wyk, E. (eds.) SLE 2008. LNCS, vol. 5452, pp. 16–34. Springer, Heidelberg (2009)
16. Moody, D.L., Heymans, P., Matulevičius, R.: Improving the Effectiveness of Visual Representations in Requirements Engineering: An Evaluation of i* Visual Syntax (Best Paper Award). In: Proc. of the 17th IEEE International Requirements Engineering Conference (RE 2009), Washington, DC, USA, pp. 171–180. IEEE Computer Society, Los Alamitos (2009)
17. Genon, N., Amyot, D., Heymans, P.: Analysing the Cognitive Effectiveness of the UCM Visual Notation (to appear, 2010)
18. Bertin, J.: Sémiologie graphique: Les diagrammes - Les réseaux - Les cartes. Gauthier-VillarsMouton & Cie (1983)
19. Mendling, J., Reijers, H., van der Aalst, W.: Seven Process Modelling Guidelines (7PMG). Information and Software Technology 52(2), 127–136 (2010)
20. Mendling, J., Reijers, H., Cardoso, J.: What Makes Process Models Understandable? In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 48–63. Springer, Heidelberg (2007)
21. Schrepfer, M., Wolf, J., Mendling, J., Reijers, H.: The Impact of Secondary Notation on Process Model Understanding. In: Persson, A., Stirna, J. (eds.) PoEM 2009. Lecture Notes in Business Information Processing, vol. 39, pp. 161–175. Springer, Heidelberg (2009)

22. Figl, K., Mendling, J., Strembeck, M., Recker, J.: On the Cognitive Effectiveness of Routing Symbols in Process Modeling Languages. In: Abramowicz, W., Tolksdorf, R. (eds.) BIS 2010. Lecture Notes in Business Information Processing, vol. 47, pp. 230–241. Springer, Heidelberg (2010)
23. Mendling, J., Recker, J., Reijers, H.A.: On the Usage of Labels and Icons in Business Process Modeling. Computer, 40–58 (2010)
24. Figl, K., Mendling, J., Strembeck, M.: Towards a Usability Assessment of Process Modeling Languages. In: Proc. of the 8th Workshop Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten (EPK 2009), Berlin, Germany. CEUR Workshop Proceedings, vol. 554, pp. 118–136 (2009)
25. Reijers, H.A., Mendling, J.: Modularity in Process Models: Review and Effects. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, pp. 20–35. Springer, Heidelberg (2008)
26. Muehlen, M.z., Recker, J.: How Much Language Is Enough? Theoretical and Practical Use of the Business Process Modeling Notation. In: Bellahsène, Z., Léonard, M. (eds.) CAiSE 2008. LNCS, vol. 5074, pp. 465–479. Springer, Heidelberg (2008)
27. Genon, N., Heymans, P., Moody, D.L.: BPMN 2.0 Process Models: Analysis according to the "Physics" of Notations Principles. Technical report, PReCISE - University of Namur (2010), `http://www.info.fundp.ac.be/~nge/BPMN/BPMN2_PoN_Analysis.pdf`
28. Vessey, I.: Cognitive Fit: A Theory-based Analysis of the Graphs versus Tables Literature. Decision Sciences 22, 219–240 (1991)
29. Koedinger, K., Anderson, J.: Abstract planning and conceptual chunks: Elements of expertise in geometry. Cognitive Science 14, 511–550 (1990)
30. Winn, W.: An Account of How Readers Search for Information in Diagrams. Contemporary Educational Psychology 18, 162–185 (1993)
31. Blankenship, J., Danseraeau, D.F.: The effect of animated node-link displays on information recall. The Journal of Experimental Education 68(4), 293–308 (2000)
32. Persson, A.: Enterprise Modelling in Practice: Situational Factors and their Influence on Adopting a Participative Approach. PhD thesis, Department of Computer and Systems Sciences, Stockholm University (2001)
33. Goodman, N.: Languages of Art: An Approach to a Theory of Symbols. Bobbs-Merrill Co., Indianapolis (1968)
34. Winn, W.: Encoding and Retrieval of Information in Maps and Diagrams. IEEE Transactions on Professional Communication 33(3), 103–107 (1990)
35. Biederman, I.: Recognition-by-Components: A Theory of Human Image Understanding. Psychological Review 94(2), 115–147 (1987)
36. Green, D.M., Swets, J.A.: Signal Detection Theory and Psychophysics. Wiley, Chichester (1966)
37. Petre, M.: Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. Communications of ACM 38(6), 33–44 (1995)
38. Masri, K., Parker, D., Gemino, A.: Using Iconic Graphics in Entity Relationship Diagrams: The Impact on Understanding. Journal of Database Management 19(3), 22–41 (2008)
39. Pierce, C.: The Essential Writings. Prometheus Books (1998)
40. Bar, M., Neta, M.: Humans Prefer Curved Visual Object. Psychological Science 17(8), 645–648 (2006)
41. Citrin, W.: Strategic Directions in Visual Languages Research. ACM Computing Surveys 24(4) (1996)
42. Flood, R., Carson, E.: Dealing with Complexity: an Introduction to the Theory and Application of Systems Science. Plenum Press, New York (1993)

43. Silver, B.: BPMN Method and Style. Cody-Cassidy Press (June 2009)
44. Lynch, K.: The Image of the City. MIT Press, Cambridge (1960)
45. Lamping, J., Rao, R.: The Hyperbolic Browser: a Focus + Context Technique for Visualizing Large Hierarchies. Journal of Visual Languages and Computing 7, 33–55 (1999)
46. Turetken, O., Schuff, D., Sharda, R., Ow, T.: Supporting Systems Analysis and Design Through Fisheye Views. Communications of ACM 47(9), 72–77 (2004)
47. Kim, J., Hahn, J., Hahn, H.: How Do We Understand a Systeme with (So) Many Diagrams? Cognitive Integration Processes in Diagrammatic Reasoning. Information Systems Research 11(3), 284–303 (2000)
48. Paivio, A.: Mental Representations: A Dual Coding Approach. Oxford University Press, Oxford (1986)
49. Mendling, J., Recker, J., Reijers, H.: On the Usage of Labels and Icons in Business Process Modeling. International Journal of Information System Modeling and Design, IJISMD (2009)
50. Nordbotten, J., Crosby, M.: The Effect of Graphic Style on Data Model Interpretation. Information Systems Journal 9(2), 139–156 (1999)
51. Miller, G.A.: The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. Psycological Review, 81–97 (1956)
52. Davies, I., Green, P., Rosemann, M., Indulska, M., Gallo, S.: How Do Practitioners Use Conceptual Modelling in Practice? Data and Knowledge Engineering 58, 358–380 (2006)

Featherweight TEX and Parser Correctness

Sebastian Thore Erdweg and Klaus Ostermann

University of Marburg, Germany

Abstract. TEX (and its LATEX incarnation) is a widely used document preparation system for technical and scientific documents. At the same time, TEX is also an unusual programming language with a quite powerful macro system. Despite the wide range of TEX users (especially in the scientific community), and despite a widely perceived considerable level of "pain" in using TEX, there is almost no research on TEX. This paper is an attempt to change that.

To this end, we present Featherweight TEX, a formal model of TEX which we hope can play a similar role for TEX as Featherweight Java did for Java. The main technical problem which we study in terms of Featherweight TEX is the parsing problem. As for other dynamic languages performing syntactic analysis at runtime, the concept of "static" parsing and its correctness is unclear in TEX and shall be clarified in this paper. Moreover, it is the case that parsing TEX is impossible in general, but we present evidence that parsers for practical subsets exists.

We furthermore outline three immediate applications of our formalization of TEX and its parsing: a macro debugger, an analysis that detects syntactic inconsistencies, and a test framework for TEX parsers.

1 Introduction

Almost every user of TEX [7,8] or LATEX [10] is familiar with the technique of binary error search: Since TEX error messages often give no hints about the cause of an error ("File ended while scanning use of ...", "Something's wrong - perhaps a missing ..."), TEX users comment out one half of the code were the cause is suspected and continue by binary search. The situation gets even worse when macros are involved, since all errors in macro definitions – including simple syntactic errors – only show up when the macro is invoked. Even when the error message contains some context information it will be in terms of expanded macros with no obvious relation to the cause of the error in the original document. There is no formal grammar or parser for TEX, and consequently no syntactic analysis which could reveal such errors in a modular way – let alone more sophisticated analyses such as type checkers.

Errors also often arise since it is not clear to the user how the evaluation model of TEX works: When and in which context is a piece of code evaluated? There is no formal specification, but only rather lengthy informal descriptions in TEX books [4,7]. The TEX reference implementation [8] is much too long and complicated to serve as a substitute for a crisp specification.

We believe it is a shame that the programming community designs beautiful programming languages, analyses, and tools in thousands of papers every year, yet the language which is primarily used to produce these documents is neither very well understood, nor amenable to modern analyses and tools.

We have developed a formal model (syntax and operational semantics) of TEX, which we call Featherweight TEX, or FTEX for short. FTEX omits all type-setting related properties of TEX (which are constituted mostly by a set of a few hundred primitive commands) and concentrates on the TEX macro system. We hope that it can have the same fertilizing effect on TEX research that Featherweight Java [6] had for Java.

We use FTEX to study the parsing problem. A parser for TEX would have immediate applications such as "static" syntactic error checking, code highlighting or code folding and many other conveniences of modern programming editors, and it would enable the application of other more sophisticated analyses which typically require an abstract syntax tree as input. It would also open the door to migrating thousands of existing TEX libraries to other text preparation systems or future improved versions of TEX.

To appreciate the parsing problem it is important to understand that in many dynamic languages – general-purpose languages as well as domain-specific languages – parsing and evaluation are deeply intertwined. In this context, it is not clear how a static parser could operate or what a correct parser even is. Due to the many advantages of static syntactic analyses, the parsing problem is not only relevant to TEX but to dynamic language engineering in general.

In particular, programs of dynamic languages do not necessarily have a syntax tree at all. For example, consider the following TEX program:

$$\begin{aligned}
&\backslash def \cdot \backslash app \cdot \#1 \cdot \#2 \cdot \{\#1 \cdot \#2\}\\
&\backslash def \cdot \backslash id \cdot \#1 \cdot \{\#1\}\\
&\backslash app \cdot a \cdot b\\
&\backslash app \cdot \backslash id \cdot c
\end{aligned}$$

It defines a macro $\backslash app$ which consumes two arguments, the "identity macro" $\backslash id$, and two applications of $\backslash app$. This program will evaluate to the text $a \cdot b \cdot c$. Now consider the question whether the body of $\backslash app$, $\#1 \cdot \#2$, is a macro application or a text sequence. The example shows that it can be both, depending on the arguments. If the first argument to $\backslash app$ is a macro consuming at least one argument, then it will be a macro application, otherwise a sequence. Since TEX is a Turing-complete language, the property whether a program has a parse tree is even undecidable.

Our work is based on the hypothesis that most TEX documents do have a parse tree – for example, TEX users will typically not define macros where an argument is sometimes a macro, and sometimes a character. Hence our ultimate goal is to solve the parsing problem for a class of documents that is large enough for most practical usages of TEX. To this end, we identify a set of TEX features that are particularly problematic from the perspective of parsing, and present evidence that many TEX documents do not use these features. As additional evidence we define a parser for a subset of TEX in which the absence of the problematic features is syntactically guaranteed.

We will *not* present a working parser for unrestricted TEX documents, though. A careful design of a parser will require a more in-depth analysis of typical TEX libraries and documents, to come up with reasonable heuristics. But even more importantly, we believe that the first step in developing a syntactic analysis must be to understand exactly what it means for such an analysis to be correct. For TEX, this is not clear at all. Most languages are designed on top of a context-free grammar, hence the question of correctness does not arise, but since the semantics of TEX is defined on a flat, unstructured syntax it is not clear what it means for a parse tree to be correct. We will formulate correctness of a parse tree as as a correct prediction of the application of reduction rules during program evaluation.

The contributions of this work are as follows:

- We present FTEX, the first formal model of TEX. It is, compared to existing descriptions, rather simple and concise. It can help TEX users and researchers to better understand TEX evaluation, and it can be the basis for more research on TEX.
- We describe in detail the problem of parsing TEX and formalize a correctness criterion for syntactic analyses. This correctness criterion can not only be used for formal purposes, but also as a technique to test parser implementations.
- We identify those features of TEX that are particularly worrisome from the perspective of parsing. We have also evaluated how the TEX macro system is used in typical LATEX documents by instrumenting a LATEX compiler to trace macro usages. This analysis shows that worst-case scenarios for parsing (such as dynamically changing the arity of a macro) rarely occur in practice.
- We present a working parser for a subset of TEX and verify its correctness.
- We show how our formalism can be adapted to form tools relevant in practice. Amongst others, we outline how a macro debugger can be constructed using our TEX semantics.
- Since LATEX is just a library on top of TEX, all our results apply to LATEX as well.

We also believe that our formal model and parsing approaches can be applied and adopted to other languages that have similar parsing problems, such as C with the C preprocessor, or Perl. CPP is agnostic to the grammar of C, hence the C grammar cannot be used to parse such files. Obviously, it would be desirable to have a parser that identifies macro calls and their corresponding arguments. In Perl, like in TEX, parsing and evaluation are intertwined. For example, the syntactic structure of a Perl function call `(foo $arg1, $arg2)` may be parsed as either `(foo($arg1), $arg2)` or `foo($arg1, $arg2)`, depending on whether `foo` currently accepts one or two arguments. In this paper, we will concentrate on TEX, though, and leave the application of our techniques to these languages for future work.

With regard to related work, to the best of our knowledge this is the first work investigating the macro system and parsing problem of TEX (or *any* aspect of TEX for that matter). There are some tools that try to parse LATEX code, such as pandoc[1] or syntax highlighters in LATEX IDEs such as TEXnicCenter[2], but these tools only work on a fixed quite small subset of LATEX and cannot deal with user-defined macro definitions appropriately. This is also, to our knowledge, the first work to define a correctness criterion for syntactic analyses of languages that mix parsing and evaluation. The main existing related work is in the domain of parsers for subsets of C with CPP [2,5,11,12,13,14], but none of these works is concerned with formally (or informally) defining correctness.

The rest of this paper is structured as follows. The next section describes why parsing TEX is hard. Section 3 presents our TEX formalization, FTEX. Section 4 defines parser correctness by means of a conformance relation between syntax trees and parsing constraints generated during program evaluation. In Section 5 we explain the difficulties of parsing TEX in terms of our formalization and demonstrate that provably correct parsers exist for reasonably large subsets of TEX. Our empirical study of TEX and LATEX macro usage is presented in Section 6. Section 7 discusses the applicability of our techniques to everyday TEX programming. Section 8 concludes.

2 Problem Statement

TEX has a number of properties that make parsing particularly challenging. First, TEX macros are dynamically scoped: A macro call is resolved to the last macro definition of that name which was encountered, which is not necessarily the one in the lexical scope of the macro call. This means that the target of a macro call cannot be statically determined, which is a problem for parsing since the actual macro definition determines how the call is parsed. For example, whether a is an argument to $\backslash foo$ in the macro call $\backslash foo \cdot a$ depends on the current definition of $\backslash foo$.

Second, macros in TEX can be passed as arguments to other macros. This induces the same problem as dynamic scoping: Targets of macro calls can in general not be determined statically.

Third, TEX has a lexical [3] macro system. This means that macro bodies or arguments to macros are not necessarily syntactically correct[3] pieces of TEX code. For example, a macro body may expand to an incomplete call of another macro, and the code following the original macro invocation may then complete this macro call. Similarly, macros may expand to new (potentially partial) macro definitions.

Fourth, TEX allows a custom macro call syntax through delimited parameters. Macro invocations are then "pattern-matched" against these delimiters. For example, the TEX program

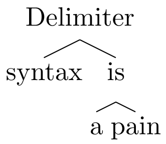
[1] `http://johnmacfarlane.net/pandoc/`
[2] `http://www.texniccenter.org`
[3] We do not know yet what syntactic correctness for TEX means anyway.

$$\backslash \mathit{def} \cdot \backslash \mathit{foo} \cdot \#1 \cdot d \cdot \#2 \cdot \{\#1 \cdot x \cdot y \cdot \#2\}$$
$$\backslash \mathit{foo} \cdot a \cdot b \cdot c \cdot d \cdot e$$

will evaluate to $a \cdot b \cdot c \cdot x \cdot y \cdot e$. Such delimiters can be used by TEX libraries to effectively define new domain-specific syntax. For example, using the `qtree`[4] library for drawing trees, the command

```
\Tree [.Delimiter syntax [.is [.a ] pain ] ]
```

results in the following tree:



There are two other parsing-related properties of TEX which, however, we will not be concerned with in this paper, namely category codes and TEX primitives. During the evaluation of a TEX program, a mapping of characters to category codes is maintained. For example, the category code `0` describes escape characters, which is usually only the backslash `\`. In principle, it is possible to change the category code mapping while evaluating a TEX program, but to our knowledge this happens very rarely outside the "bootstrapping" libraries, so we do not expect this to be a big problem in practice. It is in any case a problem that can be dealt with separately from the problem we are dealing with here.

TEX primitives are also relevant for parsing, because they can change the argument evaluation in a way that cannot be expressed using macros. Most notably, the $\backslash \mathit{expandafter}$ and $\backslash \mathit{futurelet}$ commands affect the evaluation order of programs. The former temporarily skips ahead of the expression following it, while the latter constitutes a lookahead mechanism. Furthermore, with TEX's various kinds of variables come a multitude of special-syntax assignment primitives, and alignments need their own special treatment anyway [4,7]. Still, there seems to be no conceptual hurdle for our main goal, that is the development of a correct syntactic analysis for a feature-rich subset of TEX, and we believe that the formal model presented next captures the most interesting and challenging aspects of TEX for parsing.

3 Featherweight TEX

We have formalized the core of the TEX macro system – in particular the aspects described in the previous section – in the form of a small-step operational semantics. We call this language FTEX and present its syntax in Fig. 1.

In FTEX, a program s consists of five primitive forms, namely characters c, macro identifiers m, macro parameters x, groups $\{s\}$, and the macro definition command $\backslash \mathit{def}$. We call these forms *expressions*. In addition, let $\diamondsuit$ represent the empty expression, which is not allowed to occur in user programs. Expressions are composed by sequentialization only, i.e. there is no syntactic distinction between

[4] `http://www.ling.upenn.edu/advice/latex/qtree/`

c	$\in$ *character*	characters
m	$\in \mathcal{M}$	macro variables
x	$::= \#1 \mid \ldots \mid \#9 \mid \#x$	macro parameters
e	$::= c \mid m \mid x \mid \{s\} \mid \backslash def \mid \diamondsuit$	expressions
s	$::= \overline{e}$	FTEX programs
σ	$::= \{\overline{x \mapsto s}\}$	substitutions
r	$::= c \mid x \mid m \mid \diamondsuit$	parameter tokens
d_M	$::= \backslash def \cdot M \cdot \overline{r} \cdot \{s\}$	macro values
v	$::= \varepsilon \mid c \cdot v \mid d_{\mathcal{M}} \cdot v \mid \diamondsuit \cdot v$	values
R_M	$::= [\,] \mid c \cdot R_M \mid d_M \cdot R_M$	reduction contexts
	$\mid \diamondsuit \cdot R_M \mid \{R_M\} \mid R_M \cdot e$	

Fig. 1. FTEX syntax

macro calls and the concatenation of text, see definition of s. Since this syntax does not identify the structure of macro applications, we call s the *flat syntax.* We do not restrict the use of the macro definition command to syntactically valid macro definitions (such as $md ::= \backslash def \cdot m \cdot \overline{x} \cdot \{s\}$) since in TEX macro definitions may be computed dynamically by the expansion of other macros. For example, one could define a macro $\backslash def'$ which behaves exactly like $\backslash def$ but, say, adds an additional dummy argument to the macro.

The operational semantics of FTEX, Fig. 2 is defined in a variant of the evaluation context style from Wright and Felleisen [16]. This means that, instead of introducing environments and similar entities, every relevant runtime entity is encoded within the language's syntax. The necessary additional (runtime) syntax and evaluation contexts are defined below the FTEX syntax in Fig. 1. It is important to note that the forms d and R are parametrized over the set of macro variables $M \subseteq \mathcal{M}$ that may occur in definitions.

$$\begin{aligned}
&\mathit{dropDefs} : v \rightarrow v\\
&\mathit{dropDefs}(\varepsilon) = \varepsilon\\
&\mathit{dropDefs}(c \cdot v) = c \cdot \mathit{dropDefs}(v)\\
&\mathit{dropDefs}(d \cdot v) = \diamondsuit \cdot \mathit{dropDefs}(v)\\
&\mathit{dropDefs}(\diamondsuit \cdot v) = \diamondsuit \cdot \mathit{dropDefs}(v)
\end{aligned}$$

$$\begin{aligned}
&\hat{\sigma} : s \rightarrow s\\
&\hat{\sigma}(x) = \sigma(x)\\
&\hat{\sigma}(\{s\}) = \{\hat{\sigma}(s)\}\\
&\hat{\sigma}(e) = e\\
&\hat{\sigma}(\varepsilon) = \varepsilon\\
&\hat{\sigma}(e \cdot \overline{e}) = \hat{\sigma}(e) \cdot \hat{\sigma}(\overline{e})
\end{aligned}$$

$$\frac{s \rightarrow s'}{R_{\mathcal{M}}[s] \rightarrow R_{\mathcal{M}}[s']}\ \text{(R-RSTEP)} \qquad \frac{}{\{v\} \rightarrow \mathit{dropDefs}(v)}\ \text{(R-GVAL)}$$

$$\frac{\mathit{match}(\overline{r},\, s') = \sigma}{\begin{array}{l}\backslash def \cdot m \cdot \overline{r} \cdot \{s\} \cdot R_{\mathcal{M}\backslash\{m\}}[m \cdot s']\\ \rightarrow \backslash def \cdot m \cdot \overline{r} \cdot \{s\} \cdot R_{\mathcal{M}\backslash\{m\}}[\hat{\sigma}(s)]\end{array}}\ \text{(R-MACRO)}$$

Fig. 2. FTEX reduction semantics

The reduction system has only three rules: A congruence rule (R-RStep) which allows the reduction inside a context, a reduction rule (R-GVal) to eliminate groups that are already fully evaluated, and a rule (R-Macro) for evaluating macro applications.

(R-RStep) is standard for every evaluation-context based semantics. For (R-GVal), it is important to understand that macro definitions inside a group are not visible outside the group, hence there is actually a mix of static and dynamic scoping. Within a fixed nesting level, scoping is dynamic, i.e., the rightmost definition of the macro wins, as long as it is on the same or a lower nesting level, but definitions in deeper nesting levels are ignored. For this reason it is safe to discard all macro definitions in fully evaluated groups and just retain the text contained in it - which is exactly what *dropDefs* does.

Not surprisingly, the (R-Macro) rule for evaluating macro calls is the most sophisticated one. The evaluation context $R_{\mathcal{M}\setminus\{m\}}$ is used to make sure that the macro definition of m is indeed the right-most one on the same or lower nesting level by prohibiting further definitions of m. (R-Macro) uses the parameter text $\overline{r}$ and the macro arguments s' to compute a substitution σ, which is then applied to the macro body s.[5] Substitution application $\hat{\sigma}$ is not hygienic [9], capture-avoiding or the like; rather, it just replaces every occurrence of a variable by its substitute.

The *match* function is used for matching actual macro arguments with the parameter text, i.e., the part between the macro name and the macro body, which may potentially contain delimiters of the form c or m in addition to the macro parameters; see the syntax of r. *match* expects the parameter text as its first argument and the argument text as its second one. It then generates a substitution σ, mapping macro parameters x to FTEX terms s.

Due to space limitations we cannot display the full definition of *match* here, but instead present a few examples in Fig. 3. The first two calls of *match* accord to usual parameter instantiation, where in the first one the argument's group is unpacked. Delimiter matching is illustrated by the subsequent three calls. In particular, the fifth example shows that delimited variables may instantiate to the empty expression. The last example fails because only delimited parameters can consume sequences of arguments; the second character is not matched by the parameter.

$$\begin{array}{lll}
\mathit{match}(x, & \{c_1 \cdot c_2\} &) = \{x \mapsto c_1 \cdot c_2\} \\
\mathit{match}(x_1 \cdot x_2, & c_1 \cdot c_2 &) = \{x_1 \mapsto c_1, x_2 \mapsto c_2\} \\
\mathit{match}(c_1 \cdot c_2, & c_1 \cdot c_2 &) = \{\} \\
\mathit{match}(x \cdot c_4, & c_1 \cdot c_2 \cdot c_3 \cdot c_4 &) = \{x \mapsto c_1 \cdot c_2 \cdot c_3\} \\
\mathit{match}(x \cdot c, & c &) = \{x \mapsto \Diamond\} \\
\mathit{match}(x, & c_1 \cdot c_2 &) = \mathit{undefined}
\end{array}$$

Fig. 3. Examples of Matching

[5] Here and in the remainder of this paper, we use $\overline{a} = a_1 \cdots a_n$ to denote a sequence of a's and ε to denote the empty sequence.

In summary, macros are expanded by first determining their active definition using parametrized reduction contexts, then matching the parameter text against the supplied argument text, and finally applying the resulting substitution to the macro body, which replaces the macro call.

4 Correctness of Syntactic Analyses

Given an unstructured representation of a program, syntactic analyses try to infer its structured representation, that is a syntax tree. A syntax tree is a program representation in which program fragments are composed according to the syntactic forms they inhabit. In FTEX, there are three compositional syntactic forms, namely macro application, macro definition and sequentialization, see Fig. 4. Macro applications $t\,@\,\overline{t}$ consist of a macro representation and the possibly empty list of arguments. The definition of a macro is represented by the sequence of program fragments forming the definition $\langle\overline{t}\rangle$. The sequentialization of two trees $t\,;t$ is used to denote sequential execution and result concatenation.

$$\begin{array}{lll} f & ::= c \mid m \mid x \mid \{t\} \mid \backslash def & \text{tree expressions} \\ t & ::= \emptyset \mid f \mid t\,;t \mid t\,@\,\overline{t} \mid \langle\,\overline{t}\,\rangle & \text{syntax trees} \\ t_\perp & ::= \perp \mid t & \text{parse result} \end{array}$$

Fig. 4. Tree syntax

An FTEX parser thus is a total function $p : s \rightarrow t_\perp$ assigning either a syntax tree or $\perp$ to each program in s (see Fig. 1 for the definition of s). However, not each such function is a valid parser; the resulting syntax trees must represent the original code and its structure correctly. But what characterizes a correct structural representation? Looking at syntax trees from another angle helps answer this question.

Syntax trees can also be understood to predict a program's run in that the evaluation needs to follow the structure specified by the tree. More precisely, if we assume the language's semantics to be syntax-directed, the syntax tree's inner nodes restrict the set of applicable rules to those matching the respective syntactic form. By inversion, the reduction rules used to evaluate a program constrain the set of valid syntax trees. For instance, in a program which is reduced by macro expansion, the macro and its arguments must be related by a macro application node in the syntax tree. A value, on the other hand, must correspond to a sequence of macro definitions and characters.

Accordingly, a correct FTEX parser is a total function $p : s \rightarrow t_\perp$ such that (i) each syntax tree represents the original code and (ii) each tree is compatible with the reduction rules chosen in the program's run.

In the present section, we formalize these requirements, i.e. we give a formal specification of correct syntax trees and parsers. Often, however, one wants a syntax tree to be compatible not only with the specific program run generated by evaluating the represented program; rather, syntax trees should be modular, i.e. reusable in all contexts the represented programs may be used in. We refer the specification of modular syntax trees and analyses to future work, though.

4.1 Structure Constraints

Essentially, FTEX programs are syntactically underspecified. The operator $\cdot$ is used to compose expressions, but its syntactic meaning is ambiguous; it may correspond to any of the following structural forms of the stronger tree syntax:

– Expression sequences: for example, $c_1 \cdot c_1$ corresponds to the character sequence $c_1 \,;c_2$.
– Macro application: for example, $\backslash\mathit{foo}{\cdot}c$ corresponds to the application $\backslash\mathit{foo} \mathbin{@} c$ if $\backslash\mathit{foo}$ represents a unary macro.
– Macro argument sequence: for example, in $\backslash\mathit{bar} \cdot c_1 \cdot c_2$ the second use of $\cdot$ forms a sequence of arguments to $\backslash\mathit{bar}$ if the macro takes two arguments. The call then corresponds to the syntax tree $\backslash\mathit{bar} \mathbin{@} c_1 \cdot c_2$.
– Macro definition constituent sequence: for example, all occurrences of $\cdot$ in $\backslash\mathit{def} \cdot \backslash\mathit{id} \cdot \#1 \cdot \{\#1\}$ contribute to composing the constituents of the macro definition $\langle\backslash\mathit{def} \cdot \backslash\mathit{id} \cdot \#1 \cdot \{\#1\}\rangle$.

While evaluating an FTEX program, the syntactic meaning of uses of $\cdot$ become apparent successively, as illustrated in the above examples. In order to reason about a program's syntactic runtime behavior, we introduce a constraint system which relates the syntactic meaning of occurrences of $\cdot$ to their use during reduction. A syntax tree then has to satisfy all generated constraints, i.e. it has to predict the syntactic meaning of each occurrence of $\cdot$ correctly.

In order to distinguish different uses of $\cdot$ and relate them to their incarnations in syntax trees, we introduce labels $\ell \in \mathcal{L}$ for expression concatenation and tree composition as shown in Fig. 5.[6] Similar to sequences $\overline{a}$, we write $\widetilde{a} = a_1 \cdot^{\ell_1} \cdots \cdot^{\ell_{n-1}} a_n$ to denote sequences of a's where the composing operator $\cdot$ is labeled by labels $\ell_i \in \mathcal{L}$. All labels in non-reduced FTEX programs and syntax trees are required to be unique. The reduction semantics from the previous section is refined such that labels are preserved through reduction. This, however, violates label-uniqueness as the following example shows.

$$\begin{array}{lll}
\ell & \in & \mathcal{L} \\
s & ::= & \widetilde{e} \\
t & ::= & \emptyset \mid f \mid t\,;^{\ell}\,t \mid t \mathbin{@^{\ell}} \widetilde{t} \mid \langle\widetilde{t}\rangle \\
 & & \\
k & ::= & \mathit{SEQ}(\ell) \mid \mathit{APP}(\overline{\ell}) \mid \mathit{DEF}(\overline{\ell})
\end{array}$$

Fig. 5. Label-extended Syntax and Structure Constraints

[6] Though the set of labels $\mathcal{L}$ is left abstract, we will use natural numbers as labels in examples.

Example 1

$$\begin{aligned} & \backslash \mathit{def} \cdot^1 \backslash \mathit{seq} \cdot^2 \#1 \cdot^3 \#2 \cdot^4 \{\#1 \cdot^5 \#2\} \cdot^6 \backslash \mathit{seq} \cdot^7 c_1 \cdot^8 c_2 \\ \rightarrow & \backslash \mathit{def} \cdot^1 \backslash \mathit{seq} \cdot^2 \#1 \cdot^3 \#2 \cdot^4 \{\#1 \cdot^5 \#2\} \cdot^6 c_1 \cdot^5 c_2 \end{aligned}$$

In this example, a macro $\backslash \mathit{seq}$ taking two arguments is defined. In the macro body the two arguments are composed by the operator labeled 5. Since labels are preserved through reduction, label 5 occurs outside the macro body after expanding $\backslash \mathit{seq}$, namely in the body's instantiation $c_1 \cdot^5 c_2$.

When only regarding the uninstantiated macro body of $\backslash \mathit{seq}$ there is no way of telling how the macro arguments are combined, i.e. which syntactic meaning the operator labeled 5 inhabits. Depending on the actual first argument the operator could, for instance, denote a macro application or the construction of a macro definition. In the above example, however, the macro expands into a composition of characters, hence the operator has to represent the sequentialization of expressions.

More generally, labels fulfill two purposes. First, they enable the identification of conflicts: if, for instance, a macro is called several times and expands into conflicting syntactical forms, such as sequentialization and application, there will be conflicting requirements on the involved labels. The respective operators thus have to have several syntactic meanings during runtime, which rules out a static syntactic model. Second, by using the same labels in FTEX programs and syntax trees, the structural prediction made by a parser can be easily related to the program's structure during runtime.

Our formulation of syntax tree correctness relies on the notion of structure constraints k, Fig. 5, which restrict the syntactic meaning of labeled composition operators. $\mathit{SEQ}(\ell)$ requires that $\cdot^\ell$ represents an expression sequentialization, $\mathit{APP}(\ell_1 \cdots \ell_n)$ denotes that $\cdot^{\ell_1}$ is a macro application composition and $\cdot^{\ell_2} \cdots \cdot^{\ell_n}$ forms a sequence of macro argument. Since, macro argument sequentialization always is accompanied by a macro application, only one form of constraint is needed. Lastly, constraints of the form $\mathit{DEF}(\ell_1 \cdots \ell_n)$ demand that the labeled operators $\cdot^{\ell_1} \cdots \cdot^{\ell_n}$ represent the composition of macro definition constituents.

To see these constraints in action, reconsider Ex. 1. As we will show in the subsequent subsection, all of the following constraints can be derived by evaluation:

$$\mathit{DEF}(1 \cdot 2 \cdot 3 \cdot 4),\ \mathit{SEQ}(5),\ \mathit{SEQ}(6)\ \mathit{APP}(7 \cdot 8)$$

4.2 Constraint Generation

The syntactic relations between different parts of a program emerge during evaluation. To this end, we instrument the previously presented reduction semantics to generate structure constraints as the syntactic meanings of compositions $\cdot$ become apparent.

The adapted reduction semantics is shown in Fig. 6. For a reduction step $s \xrightarrow{K} s'$, K denotes the set of generated constraints. Accordingly, the reduction rule (R-RStep) simply forwards the constraints generated for the reduction of the subexpression s. When applying (R-GVal), it is exploited that the group contains a value only, i.e. that it contains a sequence of macro definitions and characters. The corresponding structure constraints are generated by applying *valcons* to that value.

$$\begin{array}{ll}
\mathit{valcons} : v \rightarrow \{k\} & \\
\mathit{valcons}(\varepsilon) & = \emptyset \\
\mathit{valcons}(c) & = \emptyset \\
\mathit{valcons}(d) & = \{\mathit{DEF}(\ell_1 \cdots \ell_n)\}, \\
 & \quad \text{where } d = e_1 \cdot^{\ell_1} \cdots \cdot^{\ell_n} e_{n+1} \\
\mathit{valcons}(c \cdot^{\ell} v) & = \{\mathit{SEQ}(\ell)\} \cup \mathit{valcons}(v) \\
\mathit{valcons}(d \cdot^{\ell} v) & = \mathit{valcons}(d) \cup \{\mathit{SEQ}(\ell)\} \cup \mathit{valcons}(v) \\
\mathit{valcons}(\diamondsuit \cdot^{\ell} v) & = \{\mathit{SEQ}(\ell)\} \cup \mathit{valcons}(v)
\end{array}$$

$$\frac{s \xrightarrow{K} s'}{R_{\mathcal{M}}[s] \xrightarrow{K} R_{\mathcal{M}}[s']} \text{(R-RStep)} \qquad \frac{}{\{v\} \xrightarrow{\mathit{valcons}(v)} \mathit{dropDefs}(v)} \text{(R-GVal)}$$

$$\frac{}{\begin{array}{l}\backslash \mathit{def} \cdot^{\ell_1} m \cdot^{\ell_2} \{s\} \cdot^{\ell_3} R_{\mathcal{M}\backslash\{m\}}[m] \\ \xrightarrow{\emptyset} \backslash \mathit{def} \cdot^{\ell_1} m \cdot^{\ell_2} \{s\} \cdot^{\ell_3} R_{\mathcal{M}\backslash\{m\}}[\hat{\sigma}(s)]\end{array}} \text{(R-MacroEps)}$$

$$\frac{\begin{array}{c}\mathit{match}(\widetilde{r}, \widetilde{e}) = \sigma \\ \widetilde{e} = e_1 \cdot^{\ell'_1} \cdots \cdot^{\ell'_n} e_{n+1} \\ k = \mathit{APP}(\ell' \cdot \ell'_1 \cdots \ell'_n)\end{array}}{\begin{array}{l}\backslash \mathit{def} \cdot^{\ell_1} m \cdot^{\ell_2} \widetilde{r} \cdot^{\ell_3} \{s\} \cdot^{\ell_4} R_{\mathcal{M}\backslash\{m\}}[m \cdot^{\ell'} \widetilde{e}] \\ \xrightarrow{\{k\}} \backslash \mathit{def} \cdot^{\ell_1} m \cdot^{\ell_2} \widetilde{r} \cdot^{\ell_3} \{s\} \cdot^{\ell_4} R_{\mathcal{M}\backslash\{m\}}[\hat{\sigma}(s)]\end{array}} \text{(R-Macro)}$$

Fig. 6. Constraint Generation

Macro expansion is split into two rules. The first one, (R-MacroEps), covers the expansion of macros that have no parameters (note the lack of $\widetilde{r}$ in the macro definition). In this case no syntactical structure is exploited and thus no constraints are generated. The second macro expansion rule, (R-Macro), generates a single constraint which denotes the applicative structure of the expanded macro call.

Constraint generation now can be summarized as follows.

Definition 1. *The set of constraints $\Omega(s \rightarrow^* s')$ associated to the reduction sequence $s \rightarrow^* s'$ of the program s is defined by:*

$$\begin{array}{ll}
\Omega(v) & = \mathit{valcons}(v) \\
\Omega(s) & = \emptyset, \text{ if } s \neq v \text{ for all } v \\
\Omega(s \xrightarrow{K} s' \rightarrow^* s'') & = K \cup \Omega(s' \rightarrow^* s'')
\end{array}$$

In the definition of Ω, the first two cases cover reduction sequences of length zero, i.e. the constraints for the final state of the particular run of the program is defined. The third case collects all constraint sets generated during reduction and is recursively defined on the given reduction sequence.

Let us once again consider Ex. 1, where s is the initial program and s' its reduct. Then $\Omega(s \rightarrow^* s') = \{DEF(1 \cdot 2 \cdot 3 \cdot 4), SEQ(5), SEQ(6) APP(7 \cdot 8)\}$. The constraint $APP(7, 8)$ is generated because of the expansion of the macro *\seq* with arguments c_1 and c_2. The remaining constraints follow from applying *valcons* to s', which is a value and therefore is handled by the first case of Ω.

4.3 Parser Correctness

We are finally back to formulating a correctness criterion for FTEX parsers. First of all, a correct parser needs to produce a syntax tree which represents the analyzed program. In order to be able to relate syntax trees to flat program representations and structure constraints, we introduced labeled trees in Fig. 5. Labeled trees can be easily flattened and compared to weakly structured programs since composition operators are uniquely identified by their label. Accordingly, dropping all of a syntax tree's structural information while retaining the labels reveals the underlying FTEX program:

$$\begin{array}{ll}
\pi : t \rightarrow s & \\
\pi(\emptyset) & = \varepsilon \\
\pi(t_1 ;^{\ell} t_2) & = \pi(t_1) \cdot^{\ell} \pi(t_2) \\
\pi(t \,@^{\ell}\, t_1 \cdot^{\ell_1} \ldots \cdot^{\ell_{n-1}} t_n) & = \pi(t) \cdot^{\ell} \pi(t_1) \cdot^{\ell_1} \ldots \cdot^{\ell_{n-1}} \pi(t_n) \\
\pi(\langle t_1 \cdot^{\ell_1} \ldots \cdot^{\ell_{n-1}} t_n \rangle) & = \pi(t_1) \cdot^{\ell_1} \ldots \cdot^{\ell_{n-1}} \pi(t_n) \\
\pi(\{t\}) & = \{\pi(t)\} \\
\pi(f) & = f, \qquad \text{if } f \neq \{t\} \text{ for all } t
\end{array}$$

Nevertheless, not only need syntax trees to represent the code correctly, but also its structure. In the previous subsection, we were able to derive the set of constraints representing all syntactic structure a program exhibits during and after evaluation. According to our viewpoint of syntax trees as structural predictions, correct parsers must foresee a program's dynamic structures, i.e. syntax trees have to satisfy all constraints associated with the evaluation of the program.

A constraint is satisfied by a syntax tree if the constrained composition operators are represented within the tree as required. To this end, we first define what it means for a syntax tree to match a constraint, in symbols $\vdash t : k$, Fig. 7. Note that we require the labels in the constraints to equal the ones in the syntax tree, thus assuring that appropriate composition operators are matched only. Constraint satisfaction then is defined as follows.

Definition 2. *A syntax tree t satisfies a constraint k, in symbols $t \models k$, if t contains a subtree t' such that $\vdash t' : k$. A parse tree t satisfies a set of constraints K, in symbols $t \models K$, iff t satisfies all constraints $k \in K$.*

We are now able to specify correctness of FTEX parsers.

$$\frac{}{\vdash t_1 \,;^{\ell}\, t_2 \;:\; \mathit{SEQ}(\ell)}\ (\text{K-Seq}) \qquad \frac{}{\vdash t \,@^{\ell}\, t_1 \cdot^{\ell_1} \cdots \cdot^{\ell_n} t_{n+1} \;:\; \mathit{APP}(\ell \cdot \ell_1 \cdots \ell_n)}\ (\text{K-App})$$

$$\frac{}{\vdash \langle t_1 \cdot^{\ell_1} \cdots \cdot^{\ell_n} t_{n+1} \rangle \;:\; \mathit{DEF}(\ell_1 \cdots \ell_n)}\ (\text{K-Def})$$

Fig. 7. Constraint matching

Definition 3. *A total function $p : s \to t_{\bot}$ is a correct FTEX parser if and only if for all FTEX programs s with $p(s) \neq \bot$*

1. *$\pi(p(s)) = s$, and*
2. *for all programs s' with $s \to^* s'$, $p(s) \models \Omega(s \to^* s')$.*

A syntactic analysis for FTEX thus is correct if the resulting syntax trees are proper representations and structural predictions of original programs.

5 Towards Parsing TEX

In the previous section, we presented a formal specification of correct FTEX parsers. Accordingly, syntax trees computed by correct parsers must satisfy all structure constraints emerging during the evaluation of the analyzed program. For some programs, however, the set of generated structure constraints is inherently unsatisfiable, that is, inconsistent: the constraints place contradicting requirements on syntax trees.

In the present section we show that these inconsistencies arise from the problematic language features of TEX we discussed in Section 2. We furthermore demonstrate that by restricting the language such that the problematic features are excluded, we are able to define a provably correct parser for that subset.

5.1 Parsing-Contrary Language Features

When language features allow syntactic ambiguities they actually hinder syntactic analyses. For TEX, these ambiguities translate into the generation of inconsistent structure constraints, because ambiguous expression can be used inconsistently. Here we present an example for each such language feature and show that it entails inconsistent constraints. For brevity, we only denote those labels in the examples that are used for discussion.

Dynamic scoping. In TEX, parsing a macro application depends on the applied macro's actual definition. It matters whether the macro expects one or two arguments, for example, because then either the following one or two characters, say, are matched. With dynamic scoping the meaning of a macro variable depends on the dynamic scope it is expanded in. Therefore, one and the same macro variable can exhibit different syntactic properties:

$\backslash def \cdot \backslash foo \cdot \{c\}$
$\backslash def \cdot \backslash bar \cdot \{\backslash foo \cdot^1 c\}$
$\backslash bar$
$\backslash def \cdot \backslash foo \cdot \#1 \cdot \{c\}$
$\backslash bar$

Here, each of the calls to $\backslash bar$ reduces to $\backslash foo \cdot^1 c$. In the former call, however, $\backslash foo$ is defined as a constant, i.e. it does not expect any argument. Therefore, the former expansion of $\backslash bar$ implies the structure constraint $SEQ(1)$. Contrarily, when expanding the second call to $\backslash bar$, $\backslash foo$ is bound to expect one argument. Therefore, the expansion of $\backslash bar$ corresponds to a macro call of $\backslash foo$ with argument c, and hence the constraint $APP(1)$ is generated.

In a lexically scoped language, the expansion of $\backslash bar$ would consist of a closure that binds all free variables in the macro body. The macro identifier $\backslash foo$ thus would refer to the definition $\backslash def \cdot \backslash foo \cdot \{c\}$ in both expansions of $\backslash bar$.

Higher-order arguments. Similarly to macro identifiers in dynamic scoping, higher-order arguments lead to syntactically ambiguous interpretations of macro parameters. Depending on the actual argument, a parameter may represent a constant expression or in turn a macro.

In the following we define two macros, both of which take two arguments. The former one builds the sequence of its parameters while the second applies the first parameter to the second parameter.

$\backslash def \cdot \backslash seq \cdot \#1 \cdot \#2 \cdot \{\#1 \cdot^1 \#2\}$
$\backslash def \cdot \backslash app \cdot \#1 \cdot \#2 \cdot \{\#1 \cdot \#2\}$

Evidently, the only difference between the definitions of $\backslash seq$ and $\backslash app$ are their names. The macro body's syntactic structure thus only depends on whether or not the first argument is a constant or expects further input. When calling $\backslash seq$ with two characters, say, the call expands to a sequence of these characters and the constraint $SEQ(1)$ is generated. In contrast, when calling $\backslash seq$ with a unary macro and a character, it will expand into yet another macro call. In this case the conflicting constraint $APP(1)$ is generated.

Lexical macro system. In contrast to syntactical macro systems [15], macros in TEX are lexical, that is, macro arguments and bodies do not necessarily correspond to complete syntax trees.

$\backslash def \cdot \backslash foo \cdot \{\backslash def \cdot^1 \backslash baz\}$
$\backslash def \cdot \backslash bar \cdot \#1 \cdot \{\backslash foo \cdot^2 \#1 \cdot^3 \{c\}\}$

In this example neither dynamic scoping nor higher-order macros is relevant. Still, the structure of $\backslash bar$'s body is ambiguous and depends on the argument #1. The call $\backslash bar \cdot c'$, for example, expands to $\backslash def \cdot^1 \backslash baz \cdot^2 c' \cdot^3 \{c\}$ in two steps. Correspondingly, the structure constraint $DEF(1, 2, 3)$ is generated. On the other hand, if $\backslash bar$ is called as in $\backslash bar \cdot \{\{c'\}\}$, it expands to $\backslash def \cdot^1 \backslash baz \cdot^2 \{c'\} \cdot^3 \{c\}$, thus the constraints $DEF(1, 2)$ and $SEQ(3)$ are generated.

Furthermore, the body of $\backslash foo$ cannot be correctly represented by any syntax tree, because it does not inhibit a valid syntactical form. This is the intrinsic difficulty in performing syntax analyses on top of a lexical transformation system such as TEX.

Custom macro call syntax. TEX users are allowed to define their own macro call syntax as desired. For instance, the following macro needs to be called with parentheses.

$\backslash def \cdot \backslash foo \cdot (\cdot \#1 \cdot) \cdot \{c\}$

This, however, easily leads to ambiguities when the call syntax depends on macro parameters, i.e. when a macro argument is matched against the call pattern.

$\backslash def \cdot \backslash bar \cdot \#1 \cdot \{\backslash foo \cdot^1 (\cdot^2 \#1 \cdot^3)\}$
$\backslash bar \cdot c$
$\backslash bar \cdot)$

Here, the call to $\backslash foo$ in the body of $\backslash bar$ depends on the macro parameter #1. While the first call to $\backslash bar$ entails the constraint $APP(1, 2, 3)$ as expected, the second call expands to $\backslash foo \cdot^1 (\cdot^2) \cdot^3)$. Consequently, the constraints $APP(1 \cdot 2)$ and $SEQ(3)$ are generated, and establish an inconsistency with the first expansion of $\backslash bar$.

5.2 Parsing FTEX Correctly

We just identified some sources of syntactic ambiguities in FTEX, and can now focus on finding an actually parsable subset of the language. To this end, we present a non-trivial FTEX parser and prove it correct with respect to Def. 3.

First, let us fix the set of programs our parser p will be able to parse, i.e. for which p results in an actual syntax tree. These programs are subject to the following restrictions:

1. All macro definitions are complete, unary, top-level and prohibit custom call syntax, that is, they strictly follow the syntactic description $\backslash def \cdot^\ell m \cdot^\ell \#1 \cdot^\ell \{s\}$ and occur non-nestedly.
2. All uses of macro variables (except in macro definitions) are directly followed by a grouped expression, as in $m \cdot^\ell \{s\}$. Since all macros are unary, the grouped expression will correspond to the macro's argument which thus can be statically identified.
3. All macros are first-order, i.e. their argument is not a macro itself. To this end, we require all occurrences of macro parameters in macro bodies to be wrapped in groups. A higher-order argument would thus always be captured by a group, as in $\{\backslash foo\}$. This would lead to a runtime error since the content of a group must normalize to a value.

The complete parser p_i is defined in Fig. 8. p_0 accounts for the top-level programs which may contain definitions. Nested program fragments are parsed by p_1 where definitions are prohibited but wrapped macro parameters are allowed.

$$\begin{array}{ll}
p_i(\varepsilon) & = \emptyset \\
p_i(c \cdot^{\ell} s) & = c \,;^{\ell} p_i(s) \\
p_i(m \cdot^{\ell_1} \{s\} \cdot^{\ell_2} s') & = (m \,@^{\ell_1} \{p_1(s)\}) \,;^{\ell_2} p_i(s') \\
p_i(\{s\} \cdot^{\ell} s') & = \{p_1(s)\} \,;^{\ell} p_i(s') \\
p_i(\diamondsuit \cdot^{\ell} s) & = \emptyset \,;^{\ell} p_i(s) \\
p_0(\backslash def \cdot^{\ell_1} m \cdot^{\ell_2} \#1 \cdot^{\ell_3} \{s\} \cdot^{\ell_4} s') & = \langle \backslash def \cdot^{\ell_1} m \cdot^{\ell_2} x \cdot^{\ell_3} \{p_1(s)\} \rangle \,;^{\ell_4} p_0(s') \\
p_1(\{\#1\} \cdot^{\ell} s) & = \{x\} \,;^{\ell} p_1(s) \\
p_i(s) & = \bot
\end{array}$$

Fig. 8. A provably correct FTEX parser

The definition of p_i is to be read such that in each case the parser also returns $\bot$ if any nested call returns $\bot$.

In order to verify the correctness of p_0 with respect to Def. 3, we need to show that the resulting syntax trees represent the input programs and satisfy all structure constraints associated to runs of the programs.

Theorem 1. *For all programs s with $p_i(s) \neq \bot$, $\pi(p_i(s)) = s$ for $i \in \{0, 1\}$.*

Lemma 1. *For all values v, $p_i(v) \models valcons(v)$ for $i \in \{0, 1\}$.*

Lemma 2. *For $i \in \{0, 1\}$ and all programs s and s' with $s \xrightarrow{K} s'$ and $p_i(s) \neq \bot$*

(i) $p_i(s) \models K$,
(ii) $p_i(s') \neq \bot$, *and*
(iii) $\{k \mid p_i(s) \models k\} \supseteq \{k \mid p_i(s') \models k\}$.

Proof. By case distinction on the applied reduction rule.

(i) By Lem. 1 and inversion on p_0 and p_1.
(ii) For (R-MACRO) this holds since macro bodies are p_1-parsable and p_1-parsability is closed under substitution. The other cases are simple.
(iii) For (R-MACRO) let $R = \backslash def \cdot^{\ell_1} m \cdot^{\ell_2} \#1 \cdot^{\ell_3} \{s_b\} \cdot^{\ell_4} R_{\mathcal{M}\setminus\{m\}}[]$ with $R[m \cdot^{\ell'} \{s_a\}] = s$, $R[\sigma(s_b)] = s'$ and $\sigma = \{\#1 \mapsto s_a\}$. Then $\{k \mid p(R[m \cdot^{\ell'} \{s_a\}]) \models k\} \supseteq \{k \mid p(R[]) \models k \ \vee \ p(s_a) \models k\} \supseteq \{k \mid p(R[\sigma(s_b)]) \models k\}$. The other cases are simple.

Theorem 2. *For $i \in \{0, 1\}$ and all FTEX programs s and s' with $s \to^* s'$, if $p_i(s) \neq \bot$ then $p_i(s) \models \Omega(s \to^* s')$.*

Proof. By induction on the reduction sequence $s \to^* s'$. For $s = v$ by Lem. 1. For s in normal form and $s \neq v$, $\Omega(s) = \emptyset$. For $s \xrightarrow{K} s' \to^* s''$ by Lem. 2 and the induction hypothesis.

Corollary 1. *p_0 is a correct FTEX parser.*

6 Macro Usage in TEX and LATEX

In the previous section, we demonstrated how our formal specification of FTEX parsers can be instantiated to give a provably correct parser. The subset of FTEX the parser supports was designed to avoid all of the pitfalls described in Section 5.1. Parsers which support larger subsets of FTEX do exist, though, and in the present section we set out to justify their relevance in practice.

Identifying a sublanguage of TEX which is broad enough to include a wide range of programs used in practice and, at the same time, is precise enough to include a considerably large parsable subset, is a task involving a detailed study of existing TEX and LATEX libraries as well as conflict-resolving heuristics, which exclude some programs in favor of others. Nonetheless, we believe that significantly large portions of the existing TEX and LATEX libraries in fact are parsable, i.e. not exposed to the issues associated with dynamic scoping, delimiter syntax and macro arguments consuming further input. This claim is supported by a study of macro usage in existing TEX and LATEX libraries, Fig. 9, which we have been carrying out.

Macro definitions		
total	345823	∼ 100.0%
constant	320357	∼ 92.6%
recursive	56553	∼ 16.4%
delimiter syntax	12427	∼ 3.6%
redefinitions	160697	∼ 46.5%
redef. arity changes	16289	∼ 4.7%
– ignoring macros “*temp*”	7175	∼ 2.1%
redef. delimiter syntax changes	16927	∼ 4.9%
– ignoring macros “*temp*”	5827	∼ 1.7%

Macro expansions		
total	2649553	∼ 100.0%
constant	746889	∼ 28.2%
recursive	524320	∼ 19.8%
delimiter syntax	95512	∼ 3.6%
higher-order arguments	70579	∼ 2.7%

Fig. 9. Macro usage in TEX and LATEX libraries

To collect various statistics about macro definitions and applications, we adapted a Java implementation of TEX, called the New Typesetting System [1].[7] Our adaption is available at `http://www.informatik.uni-marburg.de/~seba/texstats`. We analyzed 15 LATEX documents originating from our research group and ranging from research papers to master's theses. These documents amount

[7] This implementation is incomplete with respect to the typesetting of documents, but complete regarding TEX's macro facilities.

to 301 pages of scientific writing and contain more than 300,000 macro definitions and 2,500,000 macro expansions. The high number of macro definitions, which might be surprising, is mainly caused by the use of libraries, e.g. amsmath.

Our analysis observes facts about the sample documents' uses of macro definitions and macro applications. To this end, a macro is considered constant if its definition has no declared parameter. A macro is called recursive if the transitive hull of macro identifiers used in its body contains the macro's own identifier.[8] Since the assumed recursive call not necessarily is executed, a macro may be considered recursive without actually being so. A macro has a delimited syntax if the parameter text in its definition contains characters or macro identifiers. A macro is redefined if a definition for the same macro identifier has previously been executed. A redefinition changes the arity of a macro, if the number of parameters in the previously executed definition differs. Similarly, the delimiter syntax of a macro is changed by its redefinition if the previously executed definition's parameter text contains different characters or macro identifiers, or a different order thereof. Additionally, arity and delimiter syntax changes have been recorded for macro identifiers not containing the string "temp". Lastly, a macro is said to be applied with higher-order arguments if at least on argument is a non-constant macro.

In the tables of Fig. 9, we state the absolute number of occurrences of the observed effects and their percentage relative to the respective category's total number of effects.

Our study shows that only 3.6% of all macro definitions and expansions use delimiter syntax. Furthermore, only 4.7% of all macro definitions correspond to redefinitions changing the redefined macro's arity, and 4.9% correspond to redefinitions changing its delimiter syntax. Nonetheless, most of these definitions redefine a macro with the string "temp" in its name. This gives reason to believe that the different versions of these macros are used in unrelated parts of the program, and do not entail conflicting constraints. Moreover, higher-order arguments are used only in 2.7% of all analyzed expansions.

The high number of redefinitions and constant macro definitions, 46.5% and 92.6%, is also interesting. This is in contrast to the number of expansions of parameterless macros, which is only 28.2%. We believe that this effect can be explained by the frequent usage of constant macros as variables, as opposed to behavioral abstractions. Accordingly, a redefinition of a constant macro occurs whenever the stored value has to change. This technique is often used for configuring libraries. For example, the LLNCS document class for this conference contains the macro redefinition

```
\renewcommand\labelitemi{\normalfont\bfseries --}
```

which is used in the environment *itemize* and controls the label of items in top-level lists. Any change of this parameter is a macro redefinition. Such macro redefinitions cause no problems for parsing, since the protocol and parsing-related behavior of the macro does not change.

[8] For parsing, it is unimportant whether a macro is recursive. We included this property anyway because we found it an interesting figure nonetheless.

The presented data suggests that despite dynamic scoping, delimiter syntax and higher-order arguments, the syntactical behavior of macros often can be statically determined, because these language features, which are particularly troublesome for parsing, are rarely used. Therefore, we believe that the development of a practical syntactic analysis for TEX and LATEX is possible.

7 Applications

Besides giving a precise formulation of parser correctness, our formal model has valuable, immediate applications in practice. In the following we propose three tools easing the everyday development with TEX and LATEX.

Macro debugging. Our formal FTEX semantics is defined in form of a small-step operational reduction semantics. Consequently, it allows single stepping of macro expansions and FTEX processing. This support is of high potential benefit for TEX users since it enables comprehending and debugging even complicated macro libraries.

In order to apply macro stepping meaningfully to only part of a TEX document, the context in which the code is evaluated has to be fixed. One promising possibility is to scan the document for top-level library imports and macro definitions, and reduce the code in the context of those definitions. Similar techniques are already applied in some TEX editors, however not for the purpose of debugging.

Syntactic inconsistency detection. In Section 4.2, we introduced constraint generation which denotes the dynamic syntactical structure of FTEX code. Constraint generation cannot be used in static analyses, though, because it implies evaluating the program. In a programming tool, however, gathering structural information by running the program is a valid approach. Therefore, the instrumented FTEX semantics can be used to generate and identify conflicting structure constraints, which indicate accidental syntactic inconsistencies in the analyzed program and should be reported to the programmer.

Parser testing. Almost all TEX editors contain a rudimentary parser for TEX documents, so that syntax highlighting, for example, is possible. Often enough, however, these simplistic parsers produce erroneous syntax trees, essentially disabling all tool support. By taking advantage of constraint generation again, the parser can actually be tested. As stated in the definition of parser correctness, all generated constraints need to be satisfied by resulting syntax trees. The TEX editor can thus check whether its parser is correctly predicting the document's structure, and deliberately handle cases where it is not.

More generally, all generated constraints can be understood as test cases for user defined FTEX parsers. To this end, the constraint generator and the constraint satisfaction relation comprise a framework for testing FTEX parsers.

8 Conclusion

We have defined a formal model of TEX and have clarified formally what it means for a static parser to be correct. We have identified language features that are hindering syntactic analyses and have shown that provably correct parsers exist for subsets of TEX that exclude such features. Furthermore, we have given empirical evidence that this class of practical interest. We have also demonstrated how TEX programmers may benefit from our formal model in their everyday work, for instance by using a macro debugger.

We hope that this work will trigger a new line of research that will result in a broad range of tools for TEX users and, eventually, in a new design of TEX and LATEX according to modern programming language design principles which will remove the idiosyncrasies that today's TEX users have to suffer.

Acknowledgments. This work was supported in part by the European Research Council, grant #203099.

References

1. NTS: A New Typesetting System, `http://nts.tug.org/` (visited on 20.03.2010)
2. Badros, G.J., Notkin, D.: A Framework for Preprocessor-Aware C Source Code Analyses. Software: Practice and Experience 30(8), 907–924 (2000)
3. Brabrand, C., Schwartzbach, M.I.: Growing Languages with Metamorphic Syntax Macros. In: Partial Evaluation and Semantics-Based Program Manipulation, pp. 31–40. ACM, New York (2002)
4. Eijkhout, V.: TEX by Topic, A TEXnicans Reference. Addison-Wesley, Reading (1992)
5. Garrido, A., Johnson, R.: Refactoring C with Conditional Compilation. In: Automated Software Engineering, pp. 323–326. IEEE Computer Society, Los Alamitos (2003)
6. Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java - A Minimal Core Calculus for Java and GJ. ACM Transactions on Programming Languages and Systems, 132–146 (1999)
7. Knuth, D.E.: The TEXbook. Addison-Wesley, Reading (1984)
8. Knuth, D.E.: TEX: The Program. Addison-Wesley, Reading (1986)
9. Kohlbecker, E.E., Friedman, D.P., Felleisen, M., Duba, B.F.: Hygienic Macro Expansion. In: LISP and Functional Programming, pp. 151–161. ACM, New York (1986)
10. Lamport, L.: LATEX: A Document Preparation System. Addison-Wesley, Reading (1986)
11. Latendresse, M.: Rewrite Systems for Symbolic Evaluation of C-like Preprocessing. In: European Conference on Software Maintenance and Reengineering, pp. 165–173. IEEE Computer Society, Los Alamitos (2004)
12. Livadas, P.E., Small, D.T.: Understanding Code Containing Preprocessor Constructs. In: Program Comprehension, pp. 89–97. IEEE Comp. Society, Los Alamitos (1994)
13. Padioleau, Y.: Parsing C/C++ Code without Pre-processing. In: de Moor, O., Schwartzbach, M.I. (eds.) CC 2009. LNCS, vol. 5501, pp. 109–125. Springer, Heidelberg (2009)
14. Saebjoernsen, A., Jiang, L., Quinlan, D.J., Su, Z.: Static Validation of C Preprocessor Macros. In: Automated Software Engineering, pp. 149–160. IEEE Computer Society, Los Alamitos (2009)
15. Weise, D., Crew, R.: Programmable Syntax Macros. In: Programming Language Design and Implementation, pp. 156–165. ACM, New York (1993)
16. Wright, A.K., Felleisen, M.: A Syntactic Approach to Type Soundness. Information and Computation 115(1), 38–94 (1994)

Author Index