Krzysztof Czarnecki
Görel Hedin (Eds.)

LNCS 7745

Software Language Engineering

5th International Conference, SLE 2012
Dresden, Germany, September 2012
Revised Selected Papers

Springer

Krzysztof Czarnecki Görel Hedin (Eds.)

Software Language Engineering

5th International Conference, SLE 2012
Dresden, Germany, September 26-28, 2012
Revised Selected Papers

Springer

Volume Editors

Krzysztof Czarnecki
University of Waterloo
Department of Electrical and Computer Engineering
200 University Ave. West, Waterloo, ON N2L 3G1, Canada
E-mail: kczarnec@gsd.uwaterloo.ca

Görel Hedin
Lund University
Department of Computer Science
Box 118, 221 00 Lund, Sweden
E-mail: gorel.hedin@cs.lth.se

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-36088-6 e-ISBN 978-3-642-36089-3
DOI 10.1007/978-3-642-36089-3
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012955417

CR Subject Classification (1998): D.3.2-4, D.2.1-2, D.2.11-13, F.4.2, I.2.4, I.6.5, 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 5th International Conference of Software Language Engineering (SLE 2012). The conference was held in Dresden, Germany, during September 26–28, 2012.

SLE 2012 was co-located with the 11th International Conference on Generative Programming and Component Engineering (GPCE 2012), the 4th International Workshop on Feature-Oriented Software Development (FOSD 2012), and two SLE workshops: the Industry Track of Software Language Engineering and the SLE Doctoral Symposium.

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.

The conference program included two keynote presentations, three mini-tutorials, and 19 technical paper presentations. The invited keynote speakers were Oege de Moor (Semmle Ltd. and Oxford University, UK) and Margaret-Anne Storey (University of Victoria, Canada). The mini-tutorials covered three major technical spaces relevant to SLE: grammarware, presented by Eelco Visser (Delft University of Technology, The Netherlands); modelware, presented by Richard Paige (University of York, UK); and ontologyware, presented by Giancarlo Guizzardi (Federal University of Espírito Santo, Brazil).

The response to the call for papers for SLE 2012 was higher than in 2011. We received 62 full submissions from 86 abstract submissions. From these submissions, the Program Committee (PC) selected 19 papers: 17 full papers and 2 tool demonstration papers, resulting in an acceptance rate of 31%. Each submitted paper was reviewed by at least three PC members and each paper was discussed in detail during the electronic PC meeting.

SLE 2012 would not have been possible without the significant contributions of many individuals and organizations. We are grateful to Fakultät Informatik, Technische Universität Dresden, Germany, for hosting the conference. The SLE 2012 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 submissions. We thank the authors for their efforts in writing and then revising their papers. Our final thanks go to the sponsoring and cooperating institutions for their generous support.

November 2012

Krzysztof Czarnecki
Görel Hedin

Organization

SLE 2012 was hosted by the Faculty of Computer Science, Technische Universität Dresden, Germany, in cooperation with ACM/SIGPLAN.

General Chair

Uwe Assmann	Technische Universität Dresden, Germany

Program Co-chairs

Krzysztof Czarnecki	Waterloo University, Canada
Görel Hedin	Lund University, Sweden

Organizing Committee

Jurgen Vinju	CWI, The Netherlands (Workshop Chair)
Anya Helene Bagge	University of Bergen, Norway (Poster Track Co-chair)
Dimitris Kolovos	University of York, UK (Poster Track Co-chair)
Ulrich Eisenecker	University of Leipzig, Germany (Doctoral Symposium Co-chair)
Christian Bucholdt	Credit Suisse AG, Switzerland (Doctoral Symposium Co-chair)
Jean-Marie Favre	University of Grenoble, France (Research 2.0 Chair)
Birgit Demuth	Technische Universität Dresden, Germany (Local Organization)
Sven Karol	Technische Universität Dresden, Germany (Local Organization)
Christiane Wagner	Technische Universität Dresden, Germany (Local Organization)

Steering Committee

Mark van den Brand	Eindhoven University of Technology, The Netherlands
James Cordy	Queen's University, Canada
Jean-Marie Favre	University of Grenoble, France
Dragan Gašević	Athabasca University, Canada
Görel Hedin	Lund University, Sweden
Eric Van Wyk	University of Minnesota, USA
Jurgen Vinju	CWI, The Netherlands
Kim Mens	Catholic University of Louvain, Belgium

Program Committee

Emilie Balland	INRIA Bordeaux – Sud-Ouest, France
Paulo Borba	Federal University of Pernambuco, Brazil
Claus Brabrand	IT University of Copenhagen, Denmark
Jordi Cabot	INRIA-École des Mines de Nantes, France
Dragan Gašević	Athabasca University, Canada
Jeremy Gibbons	University of Oxford, UK
Jeff Gray	University of Alabama, USA
Markus Herrmannsdörfer	Technische Universität München, Germany
Zhenjiang Hu	National Institute of Informatics, Japan
Paul Klint	Centrum Wiskunde & Informatica, The Netherlands
Julia Lawall	INRIA-Regal, France
Kim Mens	Université catholique de Louvain, Belgium
Mira Mezini	Darmstadt University of Technology, Germany
Daniel Moody	Ozemantics Pty Ltd, Australia
Pierre-Etienne Moreau	INRIA-LORIA Nancy, France
Peter Mosses	Swansea University, UK
Ralf Möller	Hamburg University of Technology, Germany
Klaus Ostermann	Marburg University, Germany
Arnd Poetzsch-Heffter	University of Kaiserslautern, Germany
Lukas Renggli	Google, Switzerland
Bernhard Rumpe	Aachen University, Germany
João Saraiva	Universidade do Minho, Portugal
Friedrich Steimann	Fernuniversität in Hagen, Germany
Mark van den Brand	Eindhoven University of Technology, The Netherlands
Eric Van Wyk	University of Minnesota, USA
Steffen Zschaler	King's College London, UK

Additional reviewers

Accioly, Paola
Al-Humaimeedy, Abeer
Andova, Suzana
Andrade, Rodrigo
Asadi, Mohsen
Bach, Jean-Christophe
Brauner, Paul
Brunelière, Hugo
Büttner, Fabian
Canovas Izquierdo, Javier Luis
Cardozo, Nicolas
Crichton, Charles
Cunha, Jácome
de Roquemaurel, Marie
Enard, Quentin
Feller, Christoph
Fernandes, Joao
Gabriels, Joost
Geilmann, Kathrin
Gonzalez, Sebastian
Grundmann, Thomas
Haber, Arne
Harper, Thomas
Izmaylova, Anastasia

Sponsoring Institutions

Technische Universität Dresden, Germany
Software Technology Group
DevBoost GmbH
i.s.x. Software GmbH & Co. KG
ANECON Software Design und Beratung GmbH

Table of Contents

Addressing Cognitive and Social Challenges in Designing and Using Ontologies in the Biomedical Domain

Margaret-Anne Storey

University of Victoria
Victoria, BC, Canada
mstorey@uvic.ca
http://www.cs.uvic.ca/~mstorey

In the life sciences and biomedical domains, ontologies are frequently used to provide a theoretical conceptualization of a domain leading to both a common vocabulary for communities of researchers and important standards to facilitate computation, software interoperability and data reuse [3]. As a theory, an ontology defines the meaning of important concepts in a domain, describes the properties of those concepts, as well as indicates the relations between concepts [7]. Most ontologies will have one or more underlying hierarchical structures, providing ways to classify or categorize important data sets. When scientific or clinical data is annotated with ontological terms, that data can be searched for and reasoned about more effectively.

The application of ontologies has become the cornerstone of semantic technologies in eScience facilitating many important discoveries and translational research in biomedicine [5]. For example, terms from the Gene Ontology are used by several research groups to manually curate experimental results from genetic experiments so that these results can be compared, queried or reasoned about. With such data annotations, findings about a gene from one model organism may be transferable to findings from other organisms bringing fresh insights on the role of certain genes in disease. Moreover, computational techniques can automate the annotations of both structured and unstructured data sets such as research papers, or clinical notes. These powerful computational techniques introduce many new opportunities for learning and hypothesizing about genes, diseases, drug interactions and biological processes.

Despite the obvious benefits of using ontologies for annotating important data, there are many challenges with designing commonly used ontologies. Each proposed application of an ontology will have diverse needs in terms of the formality requirements (i.e. how much reasoning power the ontology language may offer), the scope of the ontology (which concepts and relations are included/excluded), and the granularity of the concepts modeled in the ontology. Moreover, even if decisions based on these choices can be agreed upon, often the underlying meanings of the defined concepts will differ across or even within a community. Since an ontology is a theory and theories, particularly in the life sciences, are constantly evolving and open to debate, there will inevitably be a need for ontologies to evolve but also a need for multiple ontologies to coexist. Ontologies may further need to include concepts from upper ontologies that contain very general concepts, such as concepts about time and space or they may need to include concepts from specialized domain ontologies. Mappings can be automatically or manually created to link synonymous or related concepts in different ontologies. These mappings are important because they will enhance the querying and reasoning engines that process annotated data. Research on tools to support manual mapping activities as well as computational approaches to generate automatic mappings between ontologies is ongoing.

To handle the inevitable emerging complexity in ontology use and design, effective tool support is essential in helping humans understand existing ontologies, to understand how established ontologies evolve over time, to understand and to create mappings between ontologies, and to know which terms to use from a selected ontology for data annotation. As mentioned above, there are various technical enhancements that can automate or partially automate the mapping and annotation activities, but for most applications the human must interact with and understand the underlying ontologies and data. One form of cognitive support that is useful for ontology navigation and comprehension is information visualization [1]. There has been extensive research on developing visualization approaches suitable for the interactive display of ontologies over the past ten years [4]. However, much more research is needed to evaluate and improve these tools as most of them are difficult to use and do not scale well to more complex ontologies and sets of mappings.

Supporting collaborative activities is also essential, as the core of ontology reuse, use and design is centered on community consensus. When different communities collaboratively develop a common ontology, tensions will inevitably arise thus tool support that provides adequate transparency and communication support is crucial. Ontology authors should be able to recommend changes, add notes, compare versions and record or review ontology design decisions. Tool support for collaborative ontology authoring is also an active area of research.

In this talk, I will discuss how the relatively recent adoption and widespread use of ontologies and their associated computational tools is having an impact on scientific and medical discovery within the biomedical domain. Specifically, I will refer to examples that are supported by the BioPortal ontology library and tools. The BioPortal library, developed by the US National Center for Biomedical Ontology [6], hosts over 320 ontologies and thousands of mappings between them. BioPortal provides a wide array of web based tools and web services for browsing and visualizing ontologies (for example, BioMixer [2]). BioPortal also hosts web services for recommending ontology terms and for supporting data annotation. Currently, there are over 16,000 unique visitors to the BioPortal each month while the use of the underlying BioPortal technology has been spawned for ontology library development in domains outside the biomedical space e.g. earth science.

I will also present a recent project led by the World Health Organization that leverages the use of social media in broadening participation in the development of the next version of the International Classification of Diseases [8]. The International Classification of Diseases (ICD) is undergoing the 11th revision, and for this revision will go beyond a simple classification to include definitions of diseases with defined properties and relations added to concepts from other key health terminologies, such as SNOMED. What really sets this revision apart from earlier iterations is that for this revision many more stakeholders will be able to have a voice in how the ICD is structured and how particular diseases are defined. Previously, the ICD was defined by a very small number of experts. This project highlights the upcoming role of social media and collaborative tool support in ontology design and reuse.

To conclude this talk, I will discuss both the ongoing challenges as well as exciting opportunities that arise from using ontologies to bridge communities and integrate information systems.

Keywords: Ontology, biomedical, visualization, social media.

References

1. Ernst, N.A., Storey, M.A., Allen, P.: Cognitive support for ontology modeling. International Journal of Human-Computer Studies 62(5), 553–577 (2005)
2. Fu, B., Grammel, L., Storey, M.A.: BioMixer: A Web-based Collaborative Ontology Visualization Tool. In: 3rd International Conference on Biomedical Ontology, ICBO 2012 (2012)
3. Gruber, T.R.: Towards Principles for the Design of Ontologies Used for Knowledge Sharing. In: Guarino, N., Poli, R. (eds.) Formal Ontology in Conceptual Analysis and Knowledge Representation, vol. 43, pp. 907–928. Kluwer Academic Publishers (1993)
4. Katifori, A., Halatsis, C., Lepouras, G., Vassilakis, C., Giannopoulou, E.: Ontology visualization methods—a survey. ACM Computing Surveys 39(4) (2007)
5. Musen, M.A., Noy, N.F., Shah, N.H., Chute, C.G., Storey, M.A., Smith, B., Team, the NCBO: The National Center for Biomedical Ontology. Journal of the American Medical Informatics Association (in press, 2012), http://bmir.stanford.edu/file_asset/index.php/1729/BMIR-2011-1468.pdf
6. Noy, N.F., Shah, N.H., Whetzel, P.L., Dai, B., Dorf, M., Griffith, N., Jonquet, C., Rubin, D.L., Storey, M.A., Chute, C.G., Musen, M.A.: BioPortal: ontologies and integrated data resources at the click of a mouse. Nucleic Acids Research 37(Web Server issue) (July 2009)
7. Smith, B.: Ontology (Science). Nature Precedings (i) (July 2008)
8. Tudorache, T., Falconer, S., Noy, N.F., Nyulas, C., Üstün, T.B., Storey, M.-A., Musen, M.A.: Ontology Development for the Masses: Creating ICD-11 in WebProtégé. In: Cimiano, P., Pinto, H.S. (eds.) EKAW 2010. LNCS, vol. 6317, pp. 74–89. Springer, Heidelberg (2010)

Object Grammars
Compositional and Bidirectional Mapping between Text and Graphs

Tijs van der Storm[1,2], William R. Cook[3], and Alex Loh[3]

[1] Centrum Wiskunde & Informatica (CWI)
[2] INRIA Lille Nord Europe
[3] University of Texas at Austin

Abstract. *Object Grammars* define mappings between text and object graphs. *Parsing* recognizes syntactic features and creates the corresponding object structure. In the reverse direction, *formatting* recognizes object graph features and generates an appropriate textual presentation. The key to Object Grammars is the expressive power of the mapping, which decouples the syntactic structure from the graph structure. To handle graphs, Object Grammars support declarative annotations for resolving textual names that refer to arbitrary objects in the graph structure. Predicates on the semantic structure provide additional control over the mapping. Furthermore, Object Grammars are compositional so that languages may be defined in a modular fashion. We have implemented our approach to Object Grammars as one of the foundations of the Ensō system and illustrate the utility of our approach by showing how it enables definition and composition of domain-specific languages (DSLs).

1 Introduction

A grammar is traditionally understood as specifying a language, defined as a set of strings. Given such a grammar, it is possible to recognize whether a given string is in the language of the grammar. In practice it is more useful to actually parse a string to derive its meaning. Traditionally parsing has been defined as an extension of the more basic recognizer: when parts of the grammar are recognized, an action is invoked to create the (abstract) syntax tree. The actions are traditionally implemented in a general-purpose programming language.

In this paper we introduce *Object Grammars*: grammars that specify mappings between syntactic presentations and graph-based object structures. *Parsing* recognizes syntactic features and creates object structures. Object grammars include declarative directives indicating how to create cross-links between objects, so that the result of parsing can be a graph. *Formatting* recognizes object graph features and creates a textual presentation. Since formatting is not uniquely specified, an Object Grammar can include formatting hints to guide the rendering to text.

The second problem addressed in this paper is modularity and composition of Object Grammars. Our goal is to facilitate construction of domain-specific languages (DSLs). It is frequently desirable to reuse language fragments when creating new languages. For example, a state machine language may require an expression sub-language to represent constraints, conditions, or actions. In many cases the sublanguages may also be extended during reuse. We present a generic merge operator on that covers both reuse and extension of languages.

The contributions of this paper can be summarized as follows:

- We introduce *Object Grammars* to parse textual syntax into object graphs.
- Cross references in the object structure are resolved using *declarative paths* in the Object Grammar.
- Complex mappings can be further controlled using *predicates*.
- We show that Object Grammars are both *compositional* and *bidirectional*.
- The entire system is self-describing.

The form of Object Grammars presented in this paper is one of the foundations of Ensō, a new programming system for the definition, composition and interpretation of external DSLs[1].

2 Object Grammars

In domain-specific modeling, a software system is modeled using a variety of dedicated languages, each of which captures the essence of a single aspect. In textual modeling [27], models are represented as text, which is easy to create, edit, compare and share. To unlock their semantics, textual models must be parsed into a structure suitable for further processing, such as analysis, (abstract) interpretation or code generation.

Many domain-specific models are naturally graph structured. Well-known examples include state machines, workflow models, petri nets, network topologies and grammars. Nevertheless, traditional approaches to parsing text have focused on tree structures. Context-free grammars, for instance, are conceptually related to algebraic data types. As such, existing work on parsing is naturally predisposed towards expression languages, not modeling languages. To recover a semantic graph structure, textual references have to be resolved in a separate name-analysis phase.

Object Grammars invert this convention, taking the semantic graph structure (the model) as the primary artifact rather than the parse tree. Hence, when a textual model is parsed using an Object Grammar, the result is a graph. Where the traditional tree structure of a context-free grammar can be described by an algebraic data type, the graphs produced by object grammars are described by a *schema*. In Ensō, a schema is a class-based information model [26], similar to UML Class Diagrams [29], Entity Relationship Diagrams [8] or other meta-modeling formalisms (e.g., [5]).

There is, however, an *impedance mismatch* between grammars (as used for parsing), and object-oriented schemas (to describe structure). Previous work has suggested the use of one-to-one mappings between context-free grammar productions and schema classes [1, 40]. However, this is leads to tight coupling and synchronization of the two formats. A change to the grammar requires a change to the schema and vice versa. Object Grammars are designed to bridge grammars and schemas without sacrificing flexibility on either side. This bridge works both ways: when parsing text into object model and when formatting a model back to text. An Object Grammar specifies a mapping

[1] `http://www.enso-lang.org`

between syntax and object graphs. The syntactic structure is specified using a form of Extended Backus-Naur Form (EBNF) [41], which integrated regular iteration and optional symbols into BNF. Object Grammar extend BNF with constructs to declaratively construct objects, bind values to fields, create cross links and evaluate predicates.

2.1 Construction and Field Binding

The most fundamental feature of Object Grammars is the ability to declaratively construct objects and assign their fields with values taken from the input stream. The following example defines a production rule named `P` that parses the standard notation `(x, y)` for cartesian points and creates a corresponding `Point` object.

```
P ::= [Point] "(" x:int "," y:int ")"
```

The production rule begins with a *constructor* `[Point]` which indicates that the rule creates a `Point` object. The literals `"("`, `","` and `")"` match the literal text in the input. The *field binding* expressions `x:int` and `y:int` assign the fields `x` and `y` of the new point to integers extracted from the input steam. The classes and fields used in a grammar must be defined in a *schema* [26]. For example, the schema for points is:

```
class Point   x: int  y: int
```

Any pattern in a grammar can be refactored to introduce new non-terminals without any effect on the result of parsing For example, the above grammar can be rewritten equivalently as

```
P  ::= [Point] "(" XY ")"
XY ::= x:int "," y:int
```

The `XY` production can be reused to set the `x` and `y` fields of any kind of object, not just points.

The Object Grammars given above can also be used to format points into textual form. The constructor acts as a guard that specifies that points should be rendered. The literal symbols are copied directly to the output. The field assignments are treated as selections that format the `x` and `y` fields of the point as integers.

2.2 Alternatives and Object-Valued Fields

Each alternative in a production can construct an appropriate object. The following example constructs either a constant, or one of two different kinds of `Binary` objects. The last alternative does not construct an object, but instead returns the value created by the nested `Exp`.

```
Exp ::= [Binary] lhs:Exp op:"+" rhs:Exp
      |  [Binary] lhs:Exp op:"*" rhs:Exp
      |  [Const] value:int
      |  "(" Exp ")"
```

This grammar is not very useful, because it is ambiguous. To resolve this ambiguity, we use the standard technique for encoding precedence and associativity using additional non-terminals.

```
Term ::= [Binary] lhs:Term op:"+" rhs:Fact   |  Fact
Fact ::= [Binary] lhs:Fact op:"*" rhs:Prim   |  Prim
Prim ::= [Const] value:int                   |  "(" Term ")"
```

This grammar refactoring is independent of the schema for expressions; the additional non-terminals (`Term`, `Fact`, `Prim`) do not have corresponding classes. Object grammars allow ambiguous grammars: as long as individual input strings are not ambiguous there will be no error. Thus the original version can only parse fully parenthesized expressions, while the second version handles standard expression notation.

During formatting, the alternatives are searched in order until a matching case is found. For example, to format `Binary(Binary(3,"+",5),"*",7)` as a `Term`, the top-level structure is a binary object with a $*$ operator. The `Term` case does not apply, because the operator does not match, so it formats the second alternative, `Fact`. The first alternative of `Fact` matches, and the left hand side `Binary(3,"+",5)` must be formatted as a `Fact`. The first case for `Fact` does not match, so it is formatted as a `Prim`. The first case for `Prim` also does not match, so parentheses are added and the expression is formatted as a `Term`. The net effect is that the necessary parentheses are added automatically, to format as `(3+5)*7`.

2.3 Collections

Object Grammars support regular symbols to automatically map collections of values. For example, consider this grammar for function calls:

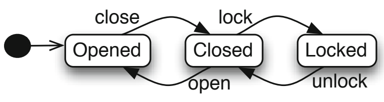
```
C ::= [Call] fun:id "(" args:Exp* @"," ")"
```

The regular repetition grammar operator $*$ may be optionally followed by a separator using `@`, which in this case is a comma. The `args` field of the `Call` class is assigned objects created by zero-or-more occurrences of `Exp`. A collection field can also be explicitly bound multiple times, rather than using the $*$ operator. For example, `args:Exp*` could be replaced by `Args?` where `Args ::= args:Exp (","Args)?`.

For formatting, the regular operators $*$ and $+$ provide additional semantics, allowing the formatter to perform intelligent grouping and indentation. A repeated group is either formatted on one line, or else it is indented and broken into multiple lines if it is too long.

2.4 Reference Resolution

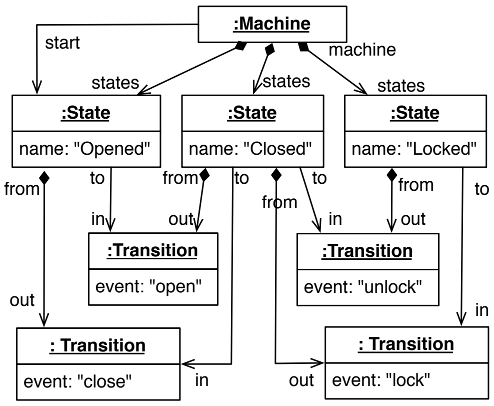
In order to explain path-based reference resolution in Object Grammars, it is instructive to introduce a slightly more elaborate example. Consider a small DSL for modeling state machines. Figure 1 displays three representations of a simple state machine representing a door that can be opened, closed, and locked. Figure 1(a) shows the state machine in graphical notation. The same state machine is rendered textually in Fig. 1(b). Internally,



(a)

```
start Opened
state Opened
  on close go Closed
state Closed
  on open go Opened
  on lock go Locked
state Locked
  on unlock go Closed
```

(b)



(c)

Fig. 1. (a) Example state machine in graphical notation, (b) the state machine in textual notation, and (c) the internal representation of the state machine in object diagram notation

```
class Machine
  start  : State
  states ! State*
```

```
class State
  machine: Machine / states
  name   # str
  out    ! Transition*
  in     : Transition*
```

```
class Transition
  event # str
  from  : State / out
  to    : State / in
```

Fig. 2. Ensō schema defining the structure of state machine object graphs

the machine itself, its states and the transitions are all represented explicitly as objects. This is illustrated in the object diagram given in Fig. 1(c).

The object diagram conforms to the State Machine schema given in Fig. 2. The schema consists of a list of named classes, each having a list of fields defined by a name, a type, and some optional modifiers. For example, the `Machine` class has a field named `states` which is a set of `State` objects. The ∗ after the type name is a modifier that marks the field as many-valued. The # annotation marks a field as a primary key, as is the case for the `name` field of the State class. As a result, state names must be unique and the `states` field of `Machine` can be indexed by name. The `/` annotation after the `machine` field indicates that the `machine` and `states` are *inverses*, as are `from`/`out` and `to`/`in`. The `!` modifier indicates that the field is part of the *spine* (a minimal spanning tree) of the object graph. If a spanning tree is defined, then all nodes in a model must be uniquely reachable by following just the spine fields. The spine gives object models a stable traversal order. Note that in the example schema, the `start` field is not part of the spine, since the start state must be included in the set of all `states`.

The textual representation in Fig. 1(b) uses *names* to represent links between states, while the graphical presentation in Fig. 1(a) uses graphical edges so names are not needed. When humans read the textual presentation in Fig. 1(b), they immediately resolve the names in each transition to create a mental picture similar Fig. 1(a).

```
start M
M ::= [Machine] "start" \start:</states[it]> states:S*
S ::= [State] "state" name:sym out:T*
T ::= [Transition] "on" event:sym "go" to:</states[it]>
```

Fig. 3. Object Grammar to parse state machines

Figure 3 shows an Object Grammar for state machines[2]. It uses the reference `</states[it]>` to look up the start state of a machine and to find the the target state of a transition. The path `/states[it]` starts at the root of the resulting object model, as indicated by the forward slash `/`. In this case the root is a `Machine` object, since `M` is the start symbol of the grammar, and the `M` production creates a `Machine`. The path then navigates into the field `states` of the machine (see Fig. 2), and uses the identifier from the input stream to index into the keyed collection of all states. The same path is used to resolve the `to` field of a transition to the target state.

References and Paths In general, a reference <*p*> represents a lookup of an object using the path *p*. Parsing a reference always consumes a single identifier, which can be used as a key for indexing into keyed collections. Binding a field to a reference thus results in a cross-link from the current object to the referenced object.

```
Path ::= [Anchor] type:"."
      |  [Anchor] type:".."
      |
[Sub] parent:Path? "/" name:sym Subscript?
Subscript
::= "[" key:Key "]"
Key  ::= Path | [It] "it"
```

Fig. 4. Syntax of paths

The syntax of paths is given in Fig. 4. A path is anchored at the current object (`.`), at its parent (`..`), or at the root. In the context of an object a path can descend into a field by post-fixing a path with `/` and the name of the field. If the field is a collection, a specific element can be referenced by indexing in square brackets. The keyword **`it`** represents the string-typed value of the identifier in the input stream that represents the reference name.

The grammar of schemas, given in Fig. 5, illustrates a more complex use of references. To lookup inverse fields, it is necessary to look for the field within the class that is the type of the field. For example, in the state machine schema in Fig. 1(b), the field `from` in `Transition` has type `State` and its inverse is the `out` field of `State`. The path for the type is `type:</types[it]>`, while the path for the inverse is `inverse:<./type/fields[it]>`, which refers to the type object. To resolve these paths, the parser must iteratively evaluate paths until all paths have been resolved.

To format a path, for example `/states[it]` in Fig. 3, the system solves the equation `/states[it]=`o to compute **`it`** given the known value o for the field. The resulting name is then output, creating a symbolic reference to a specific object.

[2] The field label `start` is escaped using \ because `start` is a keyword in the grammar of grammars; cf. Section 2.7.

```
start Schema
Schema ::= [Schema] types:TypeDef*
TypeDef ::= Primitive | Class
Primitive ::= [Primitive] "primitive" name:sym
Class ::= [Class] "class" name:sym Parent? defined_fields:Field*
Parent ::= "<" supers:</classes[it]>+ @","
Field ::= [Field] name:sym Kind type:</types[it]> Multiplicity? Annot?
Kind ::= "#" { key }   |   "!" { spine }   |   ":"
Multiplicity ::= "*" { many && optional }
               | "?" { optional }
               | "+" { many }
Annot ::= "/" inverse:<./type/fields[it]> |   "=" computed:Expr
```

Fig. 5. Schema Grammar

2.5 Predicates

The mapping between text and object graph can further be controlled using predicates. Predicates are constraint expressions on fields of objects in the object graph. During parsing, the values of these fields are updated to ensure these constraints evaluate to `true`. Conversely, during formatting, the constraints are interpreted as conditions to guide the search for the right rule alternative to format an object.

Predicates are useful for performing field assignments that are difficult to express using basic bindings. For instance, Ensō grammars have no built-in token type for boolean values to bind to. To write a grammar for booleans, one can use predicates as follows:

```
Bool ::= [Bool] "true"  { value }
       |  [Bool] "false" { !value }
```

Predicates are enclosed in curly braces. When the parser encounters the literal "`true`" it creates a `Bool` object and set its `value` field to true. Alternatively, when encountering the literal "`false`" the value field is assigned false, to satisfy the constraint that `!value` is true.

When formatting a `Bool` object, the predicates act as guards. The grammar is searched for a constructor with a fulfilled predicate or no predicate at all. Thus, a `Bool` object with field `value` set to true prints "`true`" and one with field `value` set to false prints "`false`".

A more complex example is shown in the Schema Grammar of Fig. 5. The classes and fields used in the grammar are defined in the Ensō Schema Schema [26]. The production rule for `Multiplicity` assigns the boolean fields `many` and `optional` in different ways. For instance, when a field is suffixed with the modifier "∗", both the `many` and `optional` fields are assigned to values that make the predicate true; in this case both `optional` and `many` are set to true. Conversely, during formatting, *both* `many` and `optional` must be true in the model in order to select this branch and output "∗".

2.6 Formatting Hints

Object Grammars are bidirectional: they are used for reading text into an object structure and for formatting such structure back to text. Since object structures do not maintain the layout information of the source text, formatting to text is in fact pretty-printing, and not unparsing: the formatter has to invent layout. As mentioned above, default lines breaks and indenting are generated based on the repeated expressions (marked with ∗ or +).

The layout can be further controlled by including formatting hints directly in the grammar. There are two such hints: suppress space (`.`) and force line-break (`/`). They are ordinary grammar symbols so may occur anywhere in a production.

The following example illustrates the use of `.` and `/`.

```
Exp ::= name:sym | Exp "+" Exp | "(".Exp.")"
Stat ::= Exp.";" | "{" / Stat* @/ / "}"
```

Spaces are added between all tokens by default, so the dot (`.`) is used to suppress the spaces after open parentheses and before close parentheses around expressions. Similarly, the space is suppressed before the semicolon of an expression-statement. The block statement uses explicit line breaks to put the open and close curly braces, and each statement, onto its own line. Note that the `Stat` repetition is *separated by* line-breaks (`@/`) during formatting, but this has no effect on parsing.

2.7 Lexical Syntax

Ensō's Object Grammars have a fixed lexical syntax. This is not essential: Object Grammars can easily be adapted to scannerless or tokenization-based parser frameworks. For Ensō's goal, a fixed lexical syntax is sufficient. Furthermore, it absolves the language designer of having to deal with tokenization and lexical disambiguation.

First of all, whitespace and comments are completely fixed: spaces, tabs and newlines are ignored. There is one comment convention, `//` to end of line. Second, the way primitive values are parsed is also fixed. In the examples we have seen the **`int`** and **`sym`** symbols to capture integers and identifiers respectively. Additional types are **`real`** and **`str`** for standard floating point syntax and strings enclosed in double quotes.

The symbol to capture alpha-numeric identifiers, **`sym`**, is treated in a special way, since it may cause ambiguities with the keyword literals of a language. The parser avoids such ambiguities in two ways. First, any alpha-numeric literal used in a grammar is automatically treated as a keyword and prohibited from being a **`sym`** token. Second, for both keyword literals and identifiers a longest match strategy is applied. To use reserved keywords as identifiers they can be escaped using `\`. An example of this can be seen in the state machine grammar of Fig. 3, where the `start` field name is escaped because start is a keyword in grammars.

3 Self-description

The Ensō framework is fully self-describing and Object Grammars are one of the foundations that make this possible. Grammars and schemas are both first-class Ensō models [24], just like other DSLs in the system. In Ensō, all models are an *instance* of a schema, and grammar and schema models are no exception. Schemas are instances of a "schema of schemas", which is in turn an instance of itself. For grammars the relation is *formatting*. For example, the state machine grammar of Fig. 3 *formats* state machine

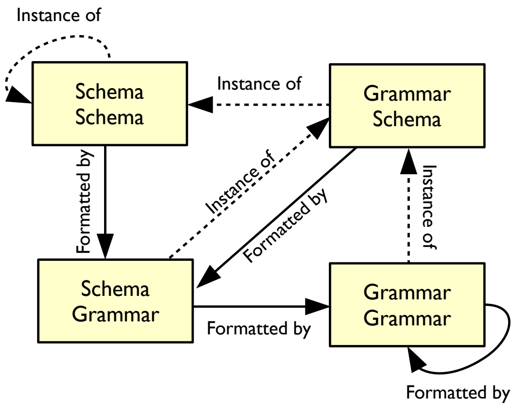


Fig. 6. The four core schema and grammar models

models. Similarly, the grammar of grammars (Fig. 7) formats itself. The grammar of schemas (Fig. 5) parses and formats schemas. The schema of grammars (Fig. 8) instantiates grammars, and is formatted using the grammar of schemas. The schema of schemas and its relationship to other schemas are explained further in a companion paper [26]. These four core models and their relations are graphically depicted in Fig. 6.

Self-description provides two important benefits. First, the interpreters that provide the parsing and formatting behavior for Object Grammars can be reused to parse and format the grammars themselves. The same holds for Schema factories that are used to construct object graphs typed by a schema: the schema of schemas is just a schema that allows the creation of schemas, including its own schema. Second, by representing core languages grammar and schema as the first-class models of Fig. 6, they become amenable to extension in the same way just like ordinary models. For instance, both the Schema Schema and the Grammar Grammar reuse a generic expression language (cf. Section 5). The self-described nature of Ensō poses interesting bootstrapping challenges. However, we consider this to be outside the scope of this paper.

3.1 Grammar Grammar

The formal syntax of Object Grammars is specified by the Grammar Grammar defined in Fig. 7. A grammar consists of the declaration of the start symbol and a collection of production rules. There are two types of rules: concrete rules and abstract rules. Both types are identified by their `name`, which identifies the non-terminal that is introduced. A concrete rule has a body that consists of one or more alternatives separated by (`|`) as defined in the `Alt` rule. For an abstract rule, the body is bound through composition with another grammar. `Path` is an example of an abstract rule, which was defined in Fig. 4. See Section 5 for more discussion of grammar composition.

The grammar rules use the standard technique for expressing precedence of grammar patterns, by adding extra non-terminals. An alternative is a `Sequence` of `Patterns` possibly prefixed by a constructor (`Create`), which creates a new object that becomes the current object for the following sequence of patterns. If there is no constructor, the

```
start Grammar
Grammar  ::= [Grammar] "start" \start:</rules[it]> rules:Rule*
Rule     ::= [Rule] name:sym "::=" arg:Alt
           |  [Rule] "abstract" name:sym
Alt      ::= [Alt] alts:Create+ @"|"
Create   ::= [Create] "[" name:sym "]" arg:Sequence  |  Sequence
Sequence ::= [Sequence] elements:Field*
Field    ::= [Field] name:sym ":" arg:Pattern        |  Pattern
Pattern  ::= [Lit] value:str
           |  [Value] kind:("int" | "str" | "real" | "sym" | "atom")
           |  [Ref] "<" path:Path ">"
           |  [Call] rule:</rules[it]>
           |  [Code] "{" code:Expr "}"
           |  [Regular] arg:Pattern "*" Sep?  { optional && many }
           |  [Regular] arg:Pattern "+" Sep?  { many }
           |  [Regular] arg:Pattern "?"       { optional }
           |  [NoSpace] "."
           |  [Break] "/"
           |  "(" Alt ")"
Sep      ::= "@" sep:Pattern
abstract Path
abstract Expr
```

Fig. 7. The Grammar Grammar: an Object Grammar that describes Object Grammars

```
class Grammar             start: Rule         rules: Rule*
class Rule                name: str           arg: Alt?
                          grammar: Grammar / rules
class Pattern
class Alt < Pattern       alts: Pattern+
class Sequence < Pattern  elements: Pattern*
class Create < Pattern    name: str           arg: Pattern
class Field < Pattern     name: str           arg: Pattern
class Lit < Pattern       value: str
class Value < Pattern     kind: str
class Ref < Pattern       path: Path
class Call < Pattern      rule: Rule
class Code < Pattern      expr: Expr
class NoSpace < Pattern
class Break < Pattern
class Regular < Pattern   arg: Pattern         sep: Pattern?
                          optional: bool       many: bool
```

Fig. 8. The Grammar Schema

current object is inherited from the calling rule. The `Patterns` in a sequence can be `Field` bindings or syntactical symbols commonly found in grammar notations, such as literals, lexical tokens, non-terminals, regular symbols, and formatting hints.

Since the grammar of grammars is itself an Ensō model, it is accompanied by a schema of grammars. This is shown in Fig. 8. Note that the different forms of regular operators in the grammar are represented by a single class `Regular` with boolean properties to define the number of repetitions.

There is something very elegant and appealing about the concise self-description in the Grammar Grammar. For example the `Create` and `Field` rules both explain and use the creation/binding syntax at the same time. The `Ref` and `Call` rules seem to be inverses of each other, as the body of a `Call` is defined by a reference, and the body of a `Ref` is a call to `Path`. The normal level and meta-level are also clearly separated, as illustated by the various uses of unquoted and quoted operators (`|` vs. `"|"`, $*$ vs. `"*"`, etc).

4 Implementation

The implementation of Ensō is a collection of interpreters for the DSLs that are used in the system. Currently, these interpreters are implemented in the Ruby programming language [14]. In contrast to most systems, which are based on generating code for a parser by compiling a grammar, Ensō uses dynamic interpretation of grammars. The same applies to schemas: a "factory" object interprets a schema to dynamically create objects and assign fields [26].

These are the two interpreters relevant for the purpose of this paper:

$$parse : (S : Schema) \to Grammar_S \to String \to S \tag{1}$$

$$format : Grammar_S \to S \to String \tag{2}$$

The *parse* function takes a value S of type Schema, a grammar (compatible with S), and a string, and returns a value of type S. Note that *parse* is dependently typed: the value of the first argument determines the type of the result, namely S. The *format* function realizes the opposite direction: given a grammar compatible with S and an value of type S it produces a textual representation.

4.1 Parsing

The parser is implemented as an interpretive variant of the GLL algorithm [33]. GLL is a general parsing algorithm. As a result it supports infinite lookahead and supports the general class of context-free grammars. Tokenization of the input stream happens on the fly, during parsing. When a certain token type is expected on the basis of the state of the parser, the scanner is asked to provide this token at the current position of the input stream. If it delivers, parsing continues, otherwise, an alternative branch in the grammar will be taken. If there are no remaining branches, a parse error is issued. The result of a successful, non-ambiguous parse is a concrete syntax tree where the nodes are annotated with grammar patterns (e.g., `Sequence`, `Create` etc.—see Fig. 8).

If parsing is successful, the object-graph is constructed from the concrete syntax tree in two steps. First, the spine of the object graph is created. This is shown in the left-hand

```
def build(t, ob=nil, f=false, vs=[], ps=[])
 l = t.label
 case l.schema_class.name
 when :Sequence then t.kids.each { |k|
     build(k, ob, false, vs, ps)
   }
 when :Create then t.kids.each { |k|
     build(k, ob=@factory.make(l.name))
   }
 when :Field then t.kids.each { |k|
     build(k, ob, true, vs=[], ps=[])
   }
   vs.each { |v| update(ob, l.name, v) }
   ps.each { |p| @fixes << [p, ob, l.name] }
 when :Lit then vs << t.value if f
 when :Value
   vs << convert(t.value, l.kind)
 when :Ref
   ps << subst_it(t.value, l.path)
 when :Code
   l.code.assert(ob)
 else then t.kids.each { |k|
     ob = build(k, ob, f, vs, ps)
   }
 end
 return ob
end
```

```
def fixup(root)
 begin
   later = []; change = false
   @fixes.each do |path, obj, fld|
     x = path.deref(root, obj)
     if x then
       # the path can be resolved
       update(obj, fld, x)
       change = true
     else
       # if not, try it later
       later << [path, obj, fld]
     end
   end
   @fixes = later
 end while change
 unless later.empty?
   raise "Fix-up error"
 end
end
```

Fig. 9. Pseudo Ruby code for building the spine and fix-up of cross-links

side of Fig. 9. The `build` algorithm recursively traverses the syntax tree and depending on the label of a node, creates objects and assigns fields. The first argument to `build` is the syntax tree, `ob` represents the "current" object; `f` indicates if field assignment can be performed. Finally, `vs` and `ps` collect values and paths respectively.

For constructor directives, a factory is called to create an object of the right class. The created object becomes the new current object when recursing down the tree. In the case for `Field` nodes, the values collected in `vs` are directly assigned to the current object. The paths `ps` are recorded as "fixes" to the current object for the current field in the global variable `@fixes`; these fixes are applied later to create cross-links.

Both `Literals` and `Values` (tokens) are simply added to the collection of values `vs`. `Values` are first converted to the expected type; the value of a literal is recorded literally, but only when the node is directly below a field binder. When a reference is encountered (`Ref`) the special keyword **`it`** is substituted for the name that has been parsed, and the result is added to `ps`. Finally, predicates (`Code`) are asserted in the context of the current object so that the referenced fields are appropriately set.

In the second step, the path-based references are resolved in an iterative fix-point process. This is shown in the right-hand side of Fig. 9. The fix-point process ensures that dependencies between references are dynamically discovered. If in the end some of the paths could not be resolved—for instance because of a cyclic dependency—an error is produced.

4.2 Formatting

Formatting works by matching constructor and field binding specifications in an Object Grammar against objects. In essence, the formatter searches for a minimal rendering that is compatible with the object graph. When the class in a constructor directive matches the class of the object being formatted, the object is formatted using the body of the production alternative. If formatting fails when recursing through the grammar, the formatter backtracks to select a different production alternative. If no suitable alternatives can be found, an error is raised.

Literals are formatted directly to the output, and fields are selected from the object. The formatter creates an intermediate formatting structure that includes the pretty printing hints of the grammar. This structure is then formatted to text using Wadler's prettier printer algorithm [39].

5 Language Composition

Modular language development presupposes a composition operator to combine two language modules into one. For two grammars, this usually involves taking the union of their production rules, where the alternatives of rules with the same name are combined. To union Object Grammars in such a way, it is also necessary to merge their target schemas so that references to classes and fields in both languages can be resolved.

In Ensō, composition of grammars and schemas are both accomplished using the same generic merge operator $\cdot \lhd \cdot$. This operator can be characterized as an overriding union where conflicts are resolved in favor of the second argument. Since a language is defined by its schema and grammar, the composition of a base language B with an extension E is given by $B_{grammar} \lhd E_{grammar}$ and $B_{schema} \lhd E_{schema}$.

5.1 Merge

The algorithm implementing $\lhd$ is shown in pseudo Ruby code in Fig. 10. There are two passes in the merge algorithm. In the first pass, `build` traverses the spine of the object graph `o1` to create any new object required. If `build` encounters an object in `o2` but none at the same location on the spine in `o1`, it creates a new copy of that object and attaches it to the graph of `o1`. Primitive fields from `o1` are always overridden by the same fields of `o2`, allowing the extension to modify the original language. Pairs of objects are merged by merging the values of each field. Collections are merged pair-wise according to their keys; `outer_join` is a relational join of two collections, matching up all pairs of items with equivalent keys and pairing up the remaining items with `nil`. At the same time, the first pass also establishes a mapping `memo`, between each object in `o2` and the corresponding object in the same spine location in `o1`.

```
def merge_into(type, o1, o2)
  build(type, o1, o2, memo = {})
  link(type, true, o1, o2, memo)
end

def build(type, a, b, memo)
  return if b.nil?
  memo[b] = new = a || type.new
  type.fields.each do |fld|
    ax = a && a[fld.name]
    bx = b[fld.name]
    if fld.type.Primitive? then
      new[fld.name] = bx
    elsif fld.spine
      if !fld.many
        build(fld.type, ax, bx, memo)
      else
        ax.outer_join(bx) do |ai, bi|
          build(fld.type, ai, bi, memo)
        end
      end
    end
  end
end
```

```
def link(type, spine, a, b, memo)
  return a if b.nil?
  new = memo[b]
  return new if !spine
  type.fields.each do |fld|
    ax = a && a[fld.name]
    bx = b[fld.name]
    next if fld.type.Primitive?
    if !fld.many? then
      val = link(fld.type, fld.spine, ax, bx, memo)
      new[fld.name] = val
    else
      ax.outer_join(bx) do |ai, bi|
        x = link(fld.type, fld.spine, ai, bi, memo)
        unless new[fld.name].include?(x)
          new[fld.name] << x
        end
      end
    end
  end
  return new
end
```

Fig. 10. Pseudo Ruby code for the generic ◁ operator

In the second phase, non-spine fields—those without the `!` modifier—are made to point to their new locations. The object graph is once again traversed along the spine, but this time `link` looks up `memo` for each non-spine field in order to find the updated target object.

5.2 Composition in Ensō

Many of the current set of languages in Ensō are defined by composing two or more language modules. Fig. 11 shows how Ensō languages are related with respect to language composition. Each edge in the diagram represents an invocation of ◁. The arrow points in the direction of the result. For instance, the Stencil and Web languages are, independently, merged into the Command language. As a result both Stencil and Web include, and possibly override and/or extend the Command language. If a language reuses or extends multiple other languages, the merge operator is applied in sequence. For instance, Grammar is first merged into Path, and then merged into Expr.

The core languages in Ensō include both the Schema and Grammar languages, as well as Stencil, a language to define graphical model editors. Additionally, Ensō features a small set of library languages that are not deployed independently but reused in other languages. An example of a library language is Expr, an expression language

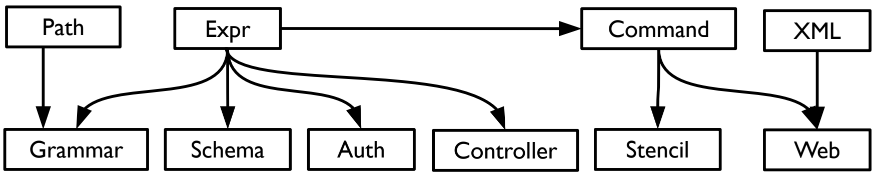


Fig. 11. Language composition in Ensō. Each arrow $A \rightarrow B$ indicates an invocation of $B \lhd A$.

Table 1. SLOC count (a) and reuse percentages (b) for schemas, grammars and interpreters of the languages currently in Ensō

Language	Schema	Grammar	Interpreter
Grammar	53	31	1243
Schema	30	20	667
Stencil	51	26	1387
Web	79	43	885
Auth	28	16	276
Piping	80	22	306
Controller	26	14	155
Path	14	6	222
Command	39	26	265
Expr	47	30	91
XML	10	6	47

(a)

Reuse Percentages			
Language	Schema	Grammar	Interpreter
Grammar	54%	54%	20%
Schema	61%	60%	12%
Stencil	63%	68%	20%
Web	55%	59%	31%
Auth	63%	65%	25%
Piping	0%	0%	0%
Controller	64%	68%	37%

(b)

with operators, variables and primitive values. It is, for instance, reused in Grammar for predicates and in Schema for computed fields. Command is a control-flow language that captures loops, conditional statements and functions. The Command language reuses the Expr language for the guards in loops and conditional statements. Another example is the language of paths (Path), shown in Fig. 4, which provides a model to address nodes in object graphs.

The reuse of Expr and Path are examples of a simple embedding. The languages are reused as black boxes, without modification. The composition of Command with Stencil and Web, however is different. Stencil is created by adding language constructs for user-interface widgets, lines, and shapes to the Command language as valid primitives. The Command language can now be used to create diagrams. A similar extension is realized in the Web language: here a language for XML element structure is mixed with the statement language of Web. The extension works in both directions: XML elements are valid statements, statements are valid XML content. The Piping and Controller languages are from a domain-specific modeling case-study in the domain of piping and instrumentation for the Language Workbench Challenge 2012 [25]. Fig. 11 only shows the Controller part which reuses Expr.

An overview of the number source lines of code (SLOC) is shown in Table 1(a). We show the number for the full languages in Ensō as well as the reused language modules (Path, Command, Expr and XML). A language consists of a schema, a grammar and an interpreter. The interpreters are all implemented in Ruby. Table 1(b) shows the reuse percentage for each language [17]. This percentage is computed as $100 \times \#SLOC_{\text{reused}}/\#SLOC_{\text{total}}$. Which languages are reused in each case can be seen from Fig. 11. As can be seen from this table, the amount of reuse in schemas and grammars is consistently high, with the exception of the Piping language, which does not reuse any language. It shows that the merge operator is powerful enough to combine real languages in a variety of ways, with actual payoff in terms of reuse.

6 Related Work

The subject of bridging modelware and grammarware is not new [1,40]. In the recent past, numerous approaches to mapping text to models and vice versa have been proposed [13,16,19,21,23,27,28]. Common to many of these languages is that references are resolved using globally unique, or hierarchically scoped names. Such names can be opaque Unique Universal Identifiers (UUIDs) to uniquely identify model elements or key attributes of the elements themselves [18]. The main difference between these approaches and Object Grammars is that Object Grammars replace the name-based strategy by allowing arbitrary paths through the model to find a referenced object. This facilitates mappings that require non-global or non-hierarchical scoping rules. Below we discuss representative systems in more detail.

The Textual Concrete Syntax (TCS) language supports deserialization and serialization of graph-structured models [21]. Field binders can be annotated with `{refersTo =` $\langle name \rangle$`}`, which binds the field to the object of the field's class with the field $\langle name \rangle$ having the value of the parsed token. Rules can furthermore be annotated with `addToContext` to add it to the, possibly nested, symbol table. The symbol table is built after the complete source has been parsed to allow forward references. Only simple references to in-scope entities are allowed, however. Path-based references of Object Grammars allow more complex reference resolving, possibly across nested scopes. TCS aims to have preliminary support for pretty printing directives to control nesting and indentation, spacing and custom separators. However, these features seem to be unimplemented.

Xtext is an advanced language workbench for textual DSL development [13]. The grammar formalism is restricted form of ANTLR so that both deserialization and serialization is supported. Xtext supports name-based referencing. To customize the name lookup semantics Xtext provides a Scoping API in Java. Apart from the use of simple names, Xtext differs from Object Grammars in that, by default, linking to target objects is performed lazily. Again, this can be customized by implementing the appropriate interfaces. Xtext is said to support a limited form of modularity through grammar mixins. For lexical syntax Xtext provides a standard set of terminal definitions such as INT and STRING, which are available for reuse.

EMFText is an Ecore based formalism similar to Xtext grammars [9,19]. EMFText, however, supports accurate unparsing of models that have been parsed. For models that have been created in memory or have been modified after parsing, formatting can be controlled by pretty printing hints similar to the `.` and `/` symbols presented in this paper. The grammar symbol `#`n forces printing of the n spaces. Similarly, `!`n is used for printing a line-break, followed by n indentation steps.

In the MontiCore system both metamodel (schema) and grammar are described in a single formalism [23]. This means that the non-terminals of the grammar introduces classes and syntactic categories at the same time. Grammar alternatives are declared by non-terminal "inheritance". As a result, the defined schema is directly tied to the syntactic structure of the grammar. The formalism supports the specification of associations and how they are established in separate sections. The default resolution strategy assumes file-wide unique identifiers, or syntactically hierarchical namespaces. This can be customized by programming if needed.

The Textual Concrete Syntax Specification Language (TCSSL) is another formalism to make grammars metamodel-aware [15]. It features three kinds of syntax rules: CreationRules which function like our `[Create]` annotations,—SeekRules, which look for existing objects satisfying an identifying criterion,—and SingletonRules, which are like CreationRules, but only create a new object if there is no existing object satisfying a specified criterion. The queries used in SeekRules seem more powerful than simple, name-based resolution; it is however unclear from the paper how they are applied for complex scenarios. TCSSL furthermore allows code fragments enclosed in double angular brackets (<<>>) but it is unclear how this affects model-to-text formatting.

Discussion. The requirements for mapping grammars to metamodels were first formulated in [20]: the mapping should be customizable, bidirectional and model-based. The Object Grammars presented in this paper satisfy these requirements. First, the mapping is customizable because of asynchronous binding: the resulting structures are to a large extent independent of the structure of the grammar. Path-based referencing and predicates are powerful tools to control the mapping, but admit a bidirectional semantics so that formatting of models back to text is possible. Formatting can be further guided using formatting hints. Finally, Object Grammars are clearly model-based: both grammars and schemas are themselves models, self-formatted and self-described respectively. A comparative overview of systems to parse text into graph structures that conform to class-based metamodels can be found in [18].

To our knowledge, Object Grammars represent the first approach to mapping between grammars and metamodels that supports modular combination of languages. Xtext, EMFText, TCS, MontiCore, and TCSSL are implemented using ANTLR. ANTLR's LL(*) algorithm, however, makes true grammar composition impossible. Object Grammars, on the other hand, *are* compositional due to the use of the general parsing algorithm GLL [33]. Moreover, the use of a general parsing algorithm has the advantage that there is no restriction on context-free structure. For instance, the designer of a modeling language does not have to worry about whether she can use left-recursion or whether her grammar is within some restricted class of context-free grammars, such as LL(k) or LR(k). As a result, Object Grammars can be structured in a way that is beneficial for resulting structure, without being subservient to a specific parsing algorithm. Object Grammars share this freedom with other grammar formalisms based on general parsing, such as SDF [36] and Rascal [22].

The way references are resolved in Object Grammars bears resemblance to the way attributes are evaluated in attribute grammars (AGs) [30]. AGs represent a convenient formalism to specify semantic analyses, such as name analysis and type checking, by declaring equations between inherited attributes and synthesized attributes. The AG system schedules the evaluation of the attributes automatically. Modern AG systems, such as JastAdd [11] and Silver [35], support reference attributes: instead of simple values, such attributes may evaluate to pointers to AST nodes. They can be used, for instance, to super-impose a control-flow graph on the AST. Reference resolving in Object Grammars is similar to attributes: they are declarative statements of fact, and the system—in our case the *parse* function— decides how to operationally make these statements true. Object-grammars are different, however, in the sense that the object graph is first-class, and not a decoration of an AST. Moreover, path-based references only allow navigating the object graph without performing arbitrary computations. Extending Object Grammars with AG style attributes is an area for further research.

Modular language development is an active area of research. This includes work on modular extension of DSLs and modeling languages [34, 37, 38], extensible compiler construction [4, 11], modular composition of lexers [7] and parsers [6, 32], modular name analysis [10] and modular language embedding [31]. Object Grammars support a powerful form of language composition through the generic merge operation ($\lhd$) applied in tandem to both grammars and schemas. The merge operator covers language extension and unification as discussed in [12]. In essence, merge captures an advanced form of inheritance similar to feature composition [2, 3]. However, merge currently applies only syntactic and semantic *structure*. To achieve the same level of compositionality at the level of *behavior*, i.e. interpreters, is an important direction for further research.

7 Conclusion

Object Grammars are a formalism for bidirectional mapping between text and object graphs. Unlike traditional grammars, Object Grammars include a declarative specification of the semantic structure that results from parsing. The notation allows objects to be constructed and their fields to be bound. Paths specify cross-links in the resulting graph structure. Thus the result of parsing is a graph, not a tree. Object Grammars can also be used to format an object graph into text.

Our implementation of Object Grammars in Ensō supports arbitrary context-free grammars. This is required when composing multiple grammars together. We have shown how Object Grammars are used in Ensō to support modular definition and composition of DSLs.

Acknowledgements. We thank Atze van der Ploeg and the anonymous reviewers for their comments on earlier drafts of this paper.

References

1. Alanen, M., Porres, I.: A relation between context-free grammars and meta object facility metamodels. Technical Report 606, Turku Centre for Computer Science (2004)
2. Apel, S., Hutchins, D.: A calculus for uniform feature composition. ACM Trans. Program. Lang. Syst. 32(5) 19:1–19:33 (2008)

3. Apel, S., Kastner, C., Lengauer, C.: FeatureHouse: Language-independent, automated software composition. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 221–231 (2009)
4. Avgustinov, P., Ekman, T., Tibble, J.: Modularity first: a case for mixing AOP and attribute grammars. In: Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD), pp. 25–35. ACM (2008)
5. Bąk, K., Czarnecki, K., Wąsowski, A.: Feature and Meta-Models in Clafer: Mixed, Specialized, and Coupled. In: Malloy, B., Staab, S., van den Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 102–122. Springer, Heidelberg (2011)
6. Bravenboer, M., Visser, E.: Parse Table Composition. In: Gašević, D., Lämmel, R., Van Wyk, E. (eds.) SLE 2008. LNCS, vol. 5452, pp. 74–94. Springer, Heidelberg (2009)
7. Casey, A., Hendren, L.: MetaLexer: a modular lexical specification language. In: Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD), pp. 7–18. ACM (2011)
8. Chen, P.P.: The Entity-Relationship Model—Toward a Unified View of Data. ACM Transactions on Database Systems 1(1) (1976)
9. DevBoost: EMFText: concrete syntax mapper, http://www.emftext.org/
10. 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)
11. Ekman, T., Hedin, G.: The JastAdd system—modular extensible compiler construction. Sci. Comput. Program. 69(1-3), 14–26 (2007)
12. Erdweg, S., Giarrusso, P.G., Rendel, T.: Language composition untangled. In: Proceedings of the International Workshop on Language Descriptions, Tools and Applications (LDTA) (2012)
13. Eysholdt, M., Behrens, H.: Xtext: implement your language faster than the quick and dirty way. In: OOPSLA Companion (SPLASH), pp. 307–309. ACM (2010)
14. Flanagan, D., Matsumoto, Y.: The Ruby Programming Language. O'Reilly (2008)
15. Fondement, F., Schnekenburger, R., Gérard, S., Muller, P.A.: Metamodel-aware textual concrete syntax specification. Technical Report LGL-2006-005, EPFL (December 2006)
16. Fondement, F.: Concrete syntax definition for modeling languages. PhD thesis, EPFL (2007)
17. Frakes, W., Terry, C.: Software reuse: metrics and models. ACM Comput. Surv. 28(2), 415–435 (1996)
18. 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)
19. 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)
20. Jouault, F., Bézivin, J.: On the specification of textual syntaxes for models. In: Eclipse Modeling Symposium, Eclipse Summit Europe 2006 (2006)
21. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE), pp. 249–254. ACM (2006)
22. Klint, P., van der Storm, T., Vinju, J.: Rascal: A Domain Specific Language for Source Code Analysis and Manipulation. In: Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 168–177. IEEE (2009)
23. Krahn, H., Rumpe, B., Völkel, S.: Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MoDELS 2007. LNCS, vol. 4735, pp. 286–300. Springer, Heidelberg (2007)
24. Kurtev, I., Bézivin, J., Jouault, F., Valduriez, P.: Model-based DSL frameworks. In: OOPSLA Companion (OOPSLA), pp. 602–616. ACM (2006)

25. Loh, A.: Piping and instrumentation in Ensō. Language Workbench Challenge Workshop at Code Generation 2012 (March 2012), http://www.languageworkbenches.net/index.php?title=LWC_2012 (accessed June 11, 2012)
26. Loh, A., van der Storm, T., Cook, W.R.: Managed data: Modular strategies for data abstraction (submitted), http://www.cs.utexas.edu/~wcook/Drafts/2012/ensodata.pdf
27. Merkle, B.: Textual modeling tools: overview and comparison of language workbenches. In: OOPSLA Companion (SPLASH), pp. 139–148. ACM (2010)
28. Muller, P.A., Fondement, F., Fleurey, F., Hassenforder, M., Schneckenburger, R., Gérard, S., Jézéquel, J.M.: Model-driven analysis and synthesis of textual concrete syntax. Software and System Modeling 7(4), 423–441 (2008)
29. Object Management Group: Unified Modeling Language Specification, version 1.3. OMG (March 2000), http://www.omg.org
30. Paakki, J.: Attribute grammar paradigms—a high-level methodology in language implementation. ACM Comput. Surv. 27(2), 196–255 (1995)
31. Renggli, L., Denker, M., Nierstrasz, O.: Language Boxes: Bending the Host Language with Modular Language Changes. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 274–293. Springer, Heidelberg (2010)
32. Schwerdfeger, A.C., Van Wyk, E.R.: Verifiable composition of deterministic grammars. In: Proceedings of the Conference on Programming Language Design and Implementation (PLDI), pp. 199–210. ACM (2009)
33. Scott, E., Johnstone, A.: GLL parsing. Electr. Notes Theor. Comput. Sci. 253(7), 177–189 (2010)
34. Van Wyk, E.: Aspects as modular language extensions. In: Proc. of Language Descriptions, Tools and Applications (LDTA). Electronic Notes in Theoretical Computer Science, vol. 82.3. Elsevier Science (2003)
35. 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)
36. Visser, E.: Syntax Definition for Language Prototyping. PhD thesis, University of Amsterdam (September 1997)
37. Völter, M., Solomatov, K.: Language modularization and composition with projectional language workbenches illustrated with MPS. In: Proceedings of the International Conference on Software Language Engineering (SLE). Revised selected papers. LNCS, vol. 6563. Springer (2010)
38. Völter, M., Visser, E.: Language extension and composition with language workbenches. In: OOPSLA Companion (SPLASH), pp. 301–304. ACM (2010)
39. Wadler, P.: A Prettier Printer. In: The Fun of Programming. Palgrave Macmillan (2003)
40. Wimmer, M., Kramler, G.: Bridging Grammarware and Modelware. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 159–168. Springer, Heidelberg (2006)
41. Wirth, N.: What can we do about the unnecessary diversity of notation for syntactic definitions? Commun. ACM 20(11), 822–823 (1977)

Profile-Based Abstraction and Analysis of Attribute Grammar Evaluation[*]

Anthony M. Sloane

Department of Computing, Macquarie University, Sydney, Australia
`Anthony.Sloane@mq.edu.au`

Abstract. Attribute grammars enable complex algorithms to be defined on tree and graph structures by declarative equations. An understanding of how the equations cooperate is necessary to gain a proper understanding of an algorithm defined by an attribute grammar. Existing attribute grammar tools and libraries provide little assistance with understanding the behaviour of an attribute evaluator. To do better, we need a way to summarise behaviour in terms of attributes, their values, their relationships, and the structures that are being attributed. A simple approach to program profiling is presented that models program execution as a hierarchy of domain-specific profile records. An abstract event for attribute evaluation is defined and evaluators are modified to collect event instances at run-time and assemble the model. A flexible report writer summarises the event instances along both intrinsic and derived dimensions, including ones defined by the developer. Selecting appropriate dimensions produces reports that expose complex properties of evaluator behaviour in a convenient way. The approach is illustrated and evaluated using the profiler we have built for the Kiama language processing library. We show that the method is both useful and practical.

1 Introduction

Attribute grammars promote a view of tree and graph decoration based on declarative equations defined on context-free grammar productions [18]. An attribute is defined by equations that specify its value at a node *N* as a function of constant values, the values of other attributes of *N*, and the values of attributes of nodes that are reachable from *N*. Provided that sufficient equations are defined to cover any context in which the node can occur, we obtain a declarative specification of an algorithm that can be used to compute the attribute of any such node. This approach to computation on structures has been shown to be extremely powerful. Recent applications that use attribute grammars heavily are XML integrity validation [2], protocol normalization [3], Java compilation [4], image processing [7], and genotype-phenotype mapping [11].

Any single attribute equation is usually fairly simple, but its effect is intricately intertwined with that of many other equations. The value of an attribute *A* is ultimately determined not just by *A*'s equations, but by the equations of any attribute on which *A*'s

[*] Matt Roberts, Nikki Lesley and Suzana Andova provided helpful feedback on this work. Much of the project was performed while the author was a guest of Mark van den Brand's group at TU Eindhoven. The author thanks the anonymous reviewers for their useful suggestions.

equations transitively depend. Thus, the computations that cooperate to compute a value of *A* are potentially dispersed throughout the attribute grammar. This dispersal means that it is non-trivial to determine what an attribute value is or even how an attribute is calculated just by looking at the equations.

Powerful extensions of the original attribute grammar formalism make this problem even worse. Equations in the original conception of attribute grammars can only refer to attributes of symbols that occur in the context-free grammar production on which the equation is defined. In higher-order and reference attribute grammars the value of an attribute can itself be a reference to a node upon which attributes can be evaluated [15,27]. This extension is particularly useful when defining context-sensitive properties such as name binding or programmer-defined operator precedence. This power comes at a price, however, because it means that more parts of the grammar are in play when we are trying to understand a particular attribute and how it is computed.

The situation is complicated even more by the fact that many modern attribute grammar systems use a dynamically-scheduled evaluation strategy, which precludes accurate static analysis. Some classes of attribute grammar submit to static dependence analysis that enables evaluation strategies to be computed in advance of running the evaluator [12]. One can imagine tools based on this static analysis that would assist with understanding the evaluation process. More recently, however, attribute grammar tools and libraries have mostly used an approach where the evaluation strategy is determined at run-time [9,10,24,26]. This approach admits more grammars than does a static approach, some algorithms are easier to express, and features such as higher-order and reference attributes are easier to support. However, a dynamically-scheduled approach also means that an accurate static analysis is not possible. Instead, dynamic analysis support is necessary since evaluation will be influenced by the attribute values, which will in turn be influenced by the input.

We therefore favour dynamic methods for understanding our attribute evaluators. In this paper we focus on a profile-based approach where data is collected at run-time and summarised to reveal relationships between different aspects of the execution. An option is to profile the code that implements the attribute evaluator. The resulting profiles are unsatisfactory, however, as they operate at a much lower level than the attribute grammar and require the developer to know a great deal about the implementation approach. Instead, we develop a method where both data collection and profile reports are in terms of an *abstract model of evaluation* that is based on arbitrary data *dimensions*. For example, the developer can ask for a profile that shows run-time and evaluation counts for each attribute that was evaluated during the run. Multi-dimensional reports show how dimensions are related to each other. For example, a report that summarises first along the attribute dimension and then along the subject (tree node) dimension gives insight into the places in the structure where the attributes were evaluated.

Specifically, the contributions of the paper are as follows.

1. The design of a general profiling framework based around generating a hierarchical model of program execution from domain-specific program events which have properties in arbitrary dimensions (Section 3.1).
2. The design of a profile reporting scheme that is independent of the event types produced by a program and of the dimensions possessed by the events (Section 3.2).

3. An implementation of the profiling framework and reporting scheme for attribute evaluators that use our Kiama language processing library [24] (Section 3.3).
4. An evaluation of the performance of the Kiama profiler that shows that it is sufficient for interactive use on large inputs (Section 3.4).
5. Examples of the approach and demonstration of its utility. We use the *PicoJava* Java subset and the *Oberon-0* variant of Oberon (Sections 2 and Section 4).

The paper concludes with a review of related work (Section 5) and an examination of directions for future work (Section 6).

Code and documentation for our implementation of the profiling framework can be found at `http://bitbucket.org/inkytonik/dsprofile`. Kiama can be obtained from `http://kiama.googlecode.com`.

2 Understanding Attribute Evaluation

To place our discussion on a concrete footing, we first illustrate the difficulties of attribute grammar understanding using the problem of name analysis for Java-like languages. In this section, we describe part of a standard attribute grammar solution for this problem and discuss how profiling can help us understand this attribute grammar.

2.1 PicoJava

The attribution we consider performs name analysis for the PicoJava language. This attribution was originally written as an illustration of reference attributes in the JastAdd system [8]. The Kiama distribution contains a fairly direct translation of the JastAdd attribute grammar. We present the attribute equations in a simplified system-neutral notation, however, since the problems of understanding the grammar and the profiling solution are independent of the details of the specific attribute grammar notation.

PicoJava contains declarations and uses of Java-like classes and fields, but omits most of the expression, statement and method complexity of Java. The left side of Figure 1 shows a PicoJava program consisting of a block with a declaration of class `A`. Class `A` contains two nested classes: `AA` and `AA`'s sub-class `BB`. Statements are limited to simple assignments between named objects which are either fields of the current object or qualified accesses to fields of other objects.

The problem that is solved by the attribute grammar is to check the use of all identifiers. For example, the uses of `x` in the `a.x` and `b.x` expressions are legal because of the declaration of the `x` field in `AA` and the inheritance relationship between `AA` and `BB`. However, the use of `y` in `b.y` is illegal since `BB` and `AA` declare no `y` field, even though there is a `y` field in the enclosing class `A`.

The attribute grammar operates on an abstract syntax tree representation of a program. The right side of Figure 1 shows the tree for the program on the left side. A program consists of a block containing a list of the top-level declarations and statements. Declarations are either of variables or of classes. A variable declaration (`VarDecl`) specifies the name of the variable and its type. Class declarations (`ClassDecl`) specify the class name and an optional name of the superclass. The superclass component is either `Some(c)`, if the superclass is `c`, or `None`, if there is no declared superclass. Uses of variable or class names are `Use` constructs. Classes contain a recursive block for their local declarations and statements. There is no limit to the nesting of class declarations.

2.2 Name Analysis for PicoJava

The attribute grammar that implements name analysis for PicoJava defines one main attribute `decl` whose value is the declaration corresponding to a particular access of a name. `decl` is a reference attribute whose value is the actual `VarDecl` or `ClassDecl` node in the tree. `decl` is a *synthesized attribute*, meaning that it is defined in all productions that define the structure of accesses.

```
attribute decl : Access => Decl
Use: a:Access ::= u:Use           a.decl = u.lookup (u.name)
Dot: a:Access ::= Access u:Use    a.decl = u.decl
```

We first declare the type of the attribute. The `decl` attribute is defined on `Access` nodes and its type is `Decl`, the common superclass of `VarDecl` and `ClassDecl`. The arrow => appeals to an intuition that attributes are functions from nodes to values.

PicoJava accesses come in two varieties: a direct use of a single name (`Use`) or a field reference with respect to a nested object access (a `Dot` containing an `Access` and a `Use`). In the attribute grammar the two varieties are represented by the two context-free grammar abstract syntax productions shown. Productions are written as the production name, a colon, then the grammar rule. The two parts of a grammar rule are separated by `::=`. The symbols in the grammar rule can be given names so that they can be referred

```
                            Program (
                             Block (List (
{                             ClassDecl ("A", None (),
 class A {                     Block (List (
  int y;                        VarDecl (Use ("int"), "y"),
  AA a;                         VarDecl (Use ("AA"), "a"),
  y = a.x;                      AssignStmt (
  class AA {                     Use ("y"),
   int x;                        Dot (Use ("a"), Use ("x"))),
  }                             ClassDecl ("AA", None (),
  class BB extends AA {          Block (List (
   BB b;                          VarDecl (Use ("int"), "x")))),
   b.y = b.x;                   ClassDecl ("BB", Some (Use ("AA")),
  }                              Block (List (
 }                                VarDecl (Use ("BB"), "b"),
}                                 AssignStmt (
                                   Dot (Use ("b"), Use ("y")),
                                   Dot (Use ("b"), Use ("x")))))))))))))
```

Fig. 1. A PicoJava program (left) and its abstract syntax tree (right)

to by the attribute equations. For example, in the first production the `Access` symbol is given the name `a`.

Since there are two productions that define the `Access` symbol, we need two definitions for the `decl` attribute. Each production has an equation that describes how to compute the attribute in an actual tree construct that was derived by that production. Each equation has the form *attribute* = *expression*. References to attributes of particular symbols are written using a "dot" notation on the left-hand side of an equation and within the right-hand side expression (e.g., `a.decl`). This notation is also used to access intrinsic properties of tree nodes that are already in the tree when attribution begins (e.g., the `u.name` field of a `Use`).

In order to describe how to compute an attribute value, an attribute equation can refer to any symbols of the associated production to obtain data values from attributes or intrinsic properties. Expressions can use any facility of the host environment to compute with these values. The overall effect of the `decl` equations is that the relevant declaration is determined by evaluating the `lookup` attribute at the rightmost name use in the access. For example, when evaluating `decl` for the expression `a.b.c`, which is represented by the tree `Dot(Use(a),Dot(Use(b),Use(c)))`, we will evaluate `decl` for `b.c`, `decl` for `c` and, finally, `lookup("c")` at the `Use(c)` node.

2.3 Name Lookup

The `lookup` attribute implements a search to find the declaration that matches a particular name. The value of `n.lookup(s)` is a reference to the node that represents the declaration of `s` when viewed from the scope represented by the node `n`, or a reference to a special "unknown declaration" value if no such node can be found.

```
attribute lookup (String) : Any => Decl
```

We indicate that the attribute can be evaluated at any node using the `Any` common super-class of all tree node classes.

`lookup` is a *parameterised attribute*, since it depends on the name being sought. It is also an *inherited attribute* since its value will be determined by the context surrounding the node where it is evaluated, not by the inner structure of that node.

```
Use: a:Access ::= u:Use         u.lookup(name) = a.lookup(name)
Dot: Access ::= a:Access u:Use  u.lookup(name) =
                                    a.decl.type.remoteLookup(name)
```

The first of the two cases is when `lookup` is evaluated at a `Use` node. This corresponds to the use of an unqualified name, so we just continue the search at the parent node to search in the current scope. The other case is when the use is qualified, in which case it will occur as the child of a `Dot` node. We need to find the declaration of the object which is being accessed, get its type and perform a remote lookup there.

We omit the definitions of the `type` and `remoteLookup` attributes since those details are not necessary for our discussion. `type` returns a reference to a node representing the class type from a declaration. `remoteLookup` looks for a name in a class type from the perspective of a client of that type.

The remaining cases for `lookup` propagate requests coming from `Use` nodes to the appropriate parts of the tree and to trigger searches in blocks and super-classes.

```
...: ... ::= ... b:Block ...
    if (b.contains(name))
      b.lookup(name) = b.localLookup(name)

...: ... ::= ... c:ClassDecl ...
    if (c.superClass != null) && (c.superClass.contains(name))
      c.lookup(name) = c.superClass.remoteLookup(name)

...: p ::= ... n ...
    n.lookup(name) = p.lookup(name)
```

In these productions, we use an ellipsis `...` to indicate a part of a production whose structure is not important. Some of the equations are *conditional* in that they only apply if a Boolean condition is true. If the current node is a block and that block contains a definition of the name we are looking for, we perform a local lookup in that block to find the declaration. Otherwise, if we have reached a class declaration, that class has a super-class, and the super-class contains a definition of the name, then we perform a remote lookup in that class. The final case covers all other circumstances by just propagating the search to the parent node. We are guaranteed to eventually reach at least a block since all programs consist of one.

2.4 Understanding PicoJava Name Analysis

Name analysis attribution for PicoJava is a canonical example of reference attribute grammars [8]. The equations are not lengthy, but their operation is still quite hard to understand. Some of the difficulty is due to the inherent complexity of the problem being solved. The tasks of name and type analysis for a language like PicoJava are intertwined. For example, to lookup a name in the body of a class, we may need to search the superclass type, which involves performing name analysis on the name that appears in the superclass position of the class declaration, and so on, while avoiding problems such as cycles in the inheritance chain. (The latter check is hidden in the definition of the `superClass` attribute.)

Given this inherent complexity, it is somewhat surprising that the definitions are not longer than they are. The main reason for their brevity is the power of the dynamically-scheduled attribute grammar formalism to abstract away tree traversal details. When we are writing our equations, we can reason about how declarations, global, local and remote lookups, and types relate to each other, without having to work out a particular tree traversal that evaluates the attributes in the correct order. The attributes are evaluated when their values are first demanded and caching means that we don't need to worry about which particular use of an attribute asks for it first. In contrast, a solution based on tree traversals implemented by visitors, for example, would have to explicitly plan which attributes should be evaluated at which time. Developing such a plan is a non-trivial task for this problem.

```
     202 ms total time
      47 ms profiled time (23.5%)
     206 profile records

  By attribute:

   Total Total  Self  Self  Desc  Desc Count Count
      ms     %    ms     %    ms     %           %
      41  88.0    11  24.7    30  63.3    27  13.1  decl
      30  63.7    10  22.8    19  40.9    32  15.5  lookup
      15  32.9     9  19.0     6  13.9    19   9.2  localLookup
       5  10.6     0   1.8     4   8.8    18   8.7  unknownDecl
       4   8.9     4   8.9     0   0.0    33  16.0  declarationOf
       3   7.9     2   4.8     1   3.1     7   3.4  remoteLookup
       3   6.5     2   4.9     0   1.6     2   1.0  isSubtypeOf
```

```
  By type for lookup:

   Total Total  Self  Self  Desc  Desc Count Count
      ms     %    ms     %    ms     %           %
      33  60.8     7  13.2    26  47.7    12   5.8  Use
      17  31.8     1   3.2    15  28.6     4   1.9  VarDecl
       4   8.3     2   4.2     2   4.1     5   2.4  Block
       2   3.7     0   1.1     1   2.6     4   1.9  ClassDecl
       1   2.2     0   0.7     0   1.5     3   1.5  Dot
       1   2.1     0   0.8     0   1.3     4   1.9  AssignStmt
```

Fig. 2. Extracts of profiles produced when the PicoJava name analyser processes the program in Figure 1: attribute dimension (top); attribute and node type dimensions (bottom)

It is not possible to completely ignore tree traversal. When we are developing and debugging the equations, we need help to understand how they function. It is not enough to just look at each equation by itself, since the effect is achieved by a combination of many equations. Having some information about what happens at run-time can reveal much about how the attribute grammar works. As a simple example, if we knew that our name analyser never evaluated the `superClass` attribute when processing the program in Figure 1, we would know that the equations are not correctly implementing our intent.

2.5 Profiling PicoJava Name Analysis

The rest of this paper describes a method for producing run-time profiles of the execution of attribute evaluators. Before we present the detail, we give a couple of examples to illustrate useful profiles of the PicoJava name analysis attribute grammar.

The simplest profile we can imagine is one that shows us which attributes are evaluated during a run. The top profile in Figure 2 shows an attribute profile of the PicoJava name analyser that was produced as it processed the program in Figure 1.[1] The first part of the profile gives the total run-time and the time that is accounted for by the profiled attributes. 206 attribute instances were evaluated in this run. The table summarises the run-time by apportioning it to the attributes. Each row shows the total time taken by the evaluation of the attribute and the portions attributable to the equations of the attribute itself (`Self`) and of the attributes that those equations used (`Desc` for "descendants"). The final data columns show the number of times each attribute was evaluated.

The profile shows that the `decl` attribute and the attributes it uses consume the vast majority of the time, closely followed by `lookup` and `localLookup`. The profile reveals a number of areas where further investigation might be warranted. The `localLookup` attribute consumes almost a third of the time which seems excessive. We might investigate whether replacing a linear search by a hashed lookup would improve performance. Also, the `unknownDecl` attribute is used to return a special object to represent the case where a declaration cannot be found. It is worrying that computing this special object consumes ten percent of the time.

The top profile of Figure 2 shows just a single profile dimension: the attribute that was evaluated. Our profiles can summarise execution across more than one dimension to reveal more detail. For example, we might want to know the types of the nodes at which the `lookup` attribute was evaluated. The bottom profile in Figure 2 shows the `lookup` part of a multi-dimensional profile using the attribute and node type dimensions. In this table the rows summarise a particular combination of the `lookup` attribute and node type. For example, the first line summarises the cases where the `lookup` attribute was evaluated at `Use` nodes. Since the tree in Figure 1 contains three `Dot` nodes, it is comforting to see from the fifth line of the table that we looked up names three times at such nodes.

Profiles such as those shown in Figure 2 reveal a lot about the execution of an attribute evaluator in a form that is easy to absorb. Varying our choice of dimension means that we can tailor the profiles to the particular investigation that we are carrying out. We now consider how these sorts of profiles are produced in detail, before looking at more complex examples.

3 Attribute Grammar Profiling

Our approach to producing profiles such as those shown in the last section is to instrument programs to generate *domain-specific events* while they run. The events are grouped to form a *record-based model* of the execution in domain-specific terms. Reports are generated from the model by grouping records along developer-specified *dimensions*. We consider only profiling for attribute grammars in this paper, but the method is general. By varying the events that are generated and the dimensions that are available, profilers can be built for any domain.

Section 3.1 describes the data collection method and the record-based model of execution. Section 3.2 explains how the model is used to produce reports. The implementation

[1] Elapsed time is collected in nanosecond units, but is presented in the profiles as milliseconds, so there may be some rounding errors.

of the approach for our Kiama language processing library is discussed in Section 3.3 and its performance is analysed in Section 3.4.

3.1 Data Collection and Execution Modelling

The data collection approach is based on a simple event model. We distinguish between `Start` events that signal the beginning of some program activity and `Finish` events that signal the end of an activity. We assume that the program can be modified so that `Start` and `Finish` events will be generated at appropriate times.

Each event captures the event kind (`Start` or `Finish`), the time at which it occurred, the domain-specific type of the event, and a collection of arbitrary data items associated with the event instance. Each data item is tagged with a unique dimension. The dimensions that an event has at generation time are its *intrinsic dimensions*, to differentiate them from *derived dimensions* that are calculated later.

In the attribute grammar case, a single *attribute evaluated* event type is sufficient. We assume that a `Start` instance of this event type is generated just before the evaluation of an attribute begins, and that a corresponding `Finish` instance is generated just after evaluation of an attribute ends. Attribute evaluation events have the following intrinsic dimensions:

– *attribute*: the attribute that was evaluated,
– *subject*: the node at which the attribute was evaluated,
– *parameter*: the value of the attribute's parameter (if any),
– *value*: the value that was calculated by the attribute's equations, and
– *cached*: whether the value was calculated or came from the attribute's cache.

The attribute, subject and parameter dimensions must be present in both the `Start` and `Finish` events. We call this subset of the intrinsic dimensions the *identity dimensions*. They are used by the profiler to recognise when a `Finish` event matches a `Start` event that was seen earlier. The value and cached dimensions are present only in the `Finish` event since their values are available only after the evaluation has been completed.

After the execution is completed, we collect the events to create a list of *profile records* that describe the execution. When we see a `Finish` event we match it with the corresponding `Start` event by comparing the identity dimensions. Each matching `Start-Finish` pair is combined to form a single profile record. A record contains the event type, the time taken between the occurrence of the `Start` event and the occurrence of the `Finish` event, and the union of all of the intrinsic dimensions of the two events.

We also require that the events are generated in a last-in-first-out manner so that we can automatically construct a hierarchical model of execution. In other words, if we see a `Finish` event, it must be the case that the most recently seen `Start` event that does not yet have a corresponding `Finish` event is the one that corresponds to the new `Finish` event. If this condition holds, we can regard the records that are created between a `Start` event and its corresponding `Finish` event as the *descendants* of the new profile record. This hierarchical relationship is used to derive dimensions that relate records to each other to summarise attribute dependencies (Section 4.4).

To make this discussion more concrete, consider the execution of the PicoJava name analysis attribute evaluator. Among the events generated by this evaluator will be some

Kind	*When*	*Attribute*	*Subject*	*Param*	*Value*	*Cached*
`Start`	4	`decl`	Node 1			
`Start`	4	`lookup`	Node 1	“a”		
`Start`	5	`lookup`	Node 2	“a”		
`Finish`	6	`lookup`	Node 2	“a”	Node 3	false
`Finish`	7	`lookup`	Node 1	“a”	Node 3	false
`Finish`	10	`decl`	Node 1		Node 3	false
`Start`	14	`decl`	Node 1			
`Finish`	15	`decl`	Node 1		Node 3	true

Record	*Time*	*Attribute*	*Subject*	*Param*	*Value*	*Cached*	*Descendants*
1	1	`lookup`	Node 2	“a”	Node 3	false	
2	3	`lookup`	Node 1	“a”	Node 3	false	1
3	6	`decl`	Node 1		Node 3	false	2
4	1	`decl`	Node 1		Node 3	true	

Fig. 3. Start and finish events from a program run (top) and the profile records that summarise the attribute evaluations signalled by the events (bottom)

that document the evaluation of the `decl` and `lookup` attributes. A possible execution results in the events shown in the top table of Figure 3. This trace excerpt describes four attribute evaluations. The first `Start` event marks the start at time step four of the evaluation of the `decl` attribute at Node 1. That evaluation requires an evaluation of the `lookup` attribute with parameter `"a"` at Node 1, and again at Node 2, which yields a value of Node 3, which we assume is the declaration node for `a`. The first evaluation of the `decl` attribute at Node 1 finishes at time step ten and did not use the `decl` attribute cache. If the value of the `decl` attribute at Node 1 is subsequently demanded again, represented by the final two events, its value will be obtained much more quickly using the attribute cache.

Four profile records will be created, one for each attribute evaluation (bottom table of Figure 3). For example, record three tells us that the first evaluation of `decl` took a total of six time units and required the evaluation represented by record two. Record two in turn took three time units and required the evaluation represented by record one.

3.2 Report Generation

The record list produced by the data collection process is a domain-specific model of the execution. A report is produced from this model on the basis of one or more *report dimensions*. The report generation process proceeds by considering the records one-by-one and allocating them to *report buckets* according to their report dimension values. All of the records with the same report dimension values end up in the same bucket and their execution time is accumulated. The descendant information allows us to allocate the elapsed time to either the attribute evaluation represented by a record (self) or to the other evaluations demanded by that evaluation (descendants). When all of the buckets have been assembled, a table is printed where the rows represent the different buckets and are sorted in decreasing order or execution time.

For example, if we choose the attribute dimension and summarise the data from the records in Figure 3, we get the following profile report.

```
Total Total  Self  Self  Desc  Desc Count Count
   ms     %    ms     %    ms     %           %
    7 100.0     3  42.9     4  57.1     2  50.0  decl
    4  57.1     4  57.1     0   0.0     2  50.0  lookup
```

Since we are using the attribute dimension we get two buckets, one for each of the two attributes that we have recorded.

Multi-dimensional reports are produced by an analogous process. We first report on the basis of the first dimension. Then each bucket from the first dimension table is further summarised according to the second dimension. The process continues until there are no more dimensions to consider.

3.3 Implementation

The profiles presented in this paper were collected using an implementation we built as a library in the Scala language. Profiles can be generated from code written in any Java Virtual Machine language but specific support is provided for Scala and Java.

We manually instrumented our Kiama language processing library [23,24], which is written in Scala, to use the profiling library to collect information about attribute evaluations. We augmented the attribute grammar evaluation code of Kiama to generate events as described above. While the new code had to be manually inserted, its total size is very small: only ten new method calls are needed in a module of more than five hundred lines. These calls produce the `start` and `finish` events. Their parameters are the event dimension names and values. This profiling support will be part of an upcoming release of our Kiama language processing library.

Users of Kiama need to only add one call to run their code under profiler control and to generate a report when their code is finished. All other instrumentation is in Kiama so the application code is not further affected.

An important implementation decision we made was to not encode the available dimensions and the types of their values into the profiling framework. A dimension is just represented by a string and a dimension value can be anything. Including more type information would enable a greater level of safety in the event generation and recording code. However, it would tie the framework to particular events and their dimensions. Adding new ones would require updating the interfaces or a more complex event representation with generic access to typed dimension values. Our approach is much simpler and is extremely flexible. Instrumentation code is one line at each event generation site. New events and dimensions can be added without recompiling the profiler.

Around 100 lines of support code in Kiama implements derived dimensions and access to the names of the attributes. In fact, the last of these is the only non-trivial part of the implementation. Unfortunately, Scala does not have full reflection yet so we are not able to easily recover the attribute names. As a work-around, we currently require the attributes to be defined as lazy values and some relatively fragile code examines the run-time stack to determine the names when the attributes are constructed. We plan to replace this code with a more robust implementation and allow non-lazy attribute definitions when proper reflection is available in the 2.10 release of Scala.

A profiling library component of less than 300 lines of code implements event capture, record representation, and report generation. The current implementation stores profile records as instances of a custom class. A generic format such as XML or JSON could easily be used instead and might be beneficial if the data was to be exported. As it stands, the event data is only used internally by the library, so this generality is not needed. We have not used any form of compression to reduce the space needed to store the profile records, since it has not been necessary for the test cases we have tried. It is possible that a need for space optimisation will be found when very large attribute grammars are profiled. If space usage becomes a serious problem, an on-the-fly approach could be the solution, where aggregation is performed as events are generated rather than at the end of the execution.

Events are time-stamped using the JVM `java.lang.System.nanoTime` method which has nanosecond precision. As in all profiling systems, the measured times vary from run to run depending on the machine load, but the relative times are stable. Precise nanosecond times are unlikely to be very useful since they present too much detail, so the profiler reports times in millisecond units.

Profiles that use intrinsic dimensions can be generated directly from the profile records. Derived dimensions can be added by overriding the default implementation of a library method that looks up dimension values. The method is given the dimension name and a reference to the profile record. The default implementation simply looks up the name in the record's dimension collection. An overriding implementation can return any value it likes. Sections 4.3 and 4.4 contain examples of derived dimension profiles.

The display of aggregated values can also be customised. By default, the profiler uses the standard Java `toString` method to obtain a string representation of a value. That implementation can be replaced by arbitrary code. For example, we could display subject trees using a pretty-printer instead of using the default representation. Since these sorts of values can take up a significant amount of space, they are unlikely to fit in the profile report tables. The report writer automatically detects when the value strings will not fit. Each such value is allocated a reference number which is used in the table. The actual value string is printed with the reference number below the table.

3.4 Performance

The performance of a profiling library is not important in production, but can make a difference to the efficiency of the development process. To explore the performance of our implementation, we conducted some experiments using an attribute grammar that is much bigger than the PicoJava one. The Kiama Oberon-0 example was developed for the tool challenge associated with the 2011 Workshop on Language Descriptions, Tools and Applications. Oberon-0 is the imperative language subset of the Oberon family of languages and was originally described by Wirth [28]. The challenge compiler parses and analyses Oberon-0 programs, then translates correct ones into equivalent pretty-printed C code. Our test case was compiling all of our fifty-two Oberon-0 test programs in a single run of the compiler. None of these programs is very large, but the compiler performs more than 32,000 evaluations of fourteen attributes while processing these files, so it is a serious test of the profiling system.

Running the Oberon-0 compiler on this test with profiling completely disabled takes about five to six seconds of elapsed time. Adding the event generation code to the library, but still with no report generation, doesn't make a difference that is noticeable when running from the command line. We also modified the program to collect the run-time for just the core compiler driver, thereby removing the time for other operations such as class loading. We ran all of the Oberon-0 tests in a single run as before, repeating the run ten times initially to warm up the virtual machine. Then we ran twenty-four tests, discarded both the slowest two and the fastest two results to remove any outliers, and averaged over the remaining twenty measurements. The results showed that the event generation by itself slows the core of the compiler down by a factor of about 1.4. While this is a significant difference, the command-line experiment shows that the slowdown is swamped by the time taken by other operations performed by the program. Thus, we believe that the instrumentation is practical for gathering data from large test runs.

We also investigated the time taken to produce profile reports. Producing a profile for the attribute dimension increases the run-time from around eight seconds with profiling turned on but no report generation, to about twelve seconds with report generation as well. Adding a second subject dimension increases the total time to over twenty-two seconds. Most of this time is taken by printing the many tree fragments, which illustrates that report generation time is highly dependent on the chosen dimensions. We have not performed any optimisation of the core of the report generator so it is likely that some improvement could be obtained. Nevertheless, our experiments show that the current performance is practical for typical interactive uses during development.

4 More Complex Profiles

To further demonstrate the power of our profiling approach and to provide a context for more detailed discussion, we conclude the core of the paper by considering more complex profiles. In particular, we consider different dimensions that improve our ability to understand the execution of our attribute evaluators. The profiles are produced from executions of the PicoJava and Oberon-0 compilers.

4.1 Parameterised Attributes

Parameterised attributes are a powerful feature of modern attribute grammar systems, but their operation can be opaque since the parameter values are not part of the equations. Seeing the parameter values that are used can greatly increase understanding and reveal problems. A profile along the attribute and parameter dimensions shows us exactly which parameters are being used and how often. For example, the PicoJava name analyser yields the following profile for the attribute `localLookup`. Parameter values are optional, so they are shown as either `Some(v)`, representing the value `v`, or `None`, which denotes that a parameter was not used.

```
Total Total  Self  Self  Desc  Desc Count Count
   ms     %    ms     %    ms     %           %
    8  22.1     4  11.2     4  11.0     5   2.4  Some(int)
    3   9.7     2   5.9     1   3.8     1   0.5  Some($unknown)
    0   1.4     0   1.0     0   0.4     3   1.5  Some(BB)
    0   0.8     0   0.7     0   0.1     3   1.5  Some(y)
    0   0.7     0   0.6     0   0.1     3   1.5  Some(x)
    0   0.5     0   0.5     0   0.0     2   1.0  Some(AA)
    0   0.2     0   0.2     0   0.0     1   0.5  Some(a)
    0   0.2     0   0.2     0   0.0     1   0.5  Some(b)
```

Quite a lot of time is spent looking up the pre-defined type name `int` and something called `$unknown`. The latter is only looked up once and still manages to take more time than the lookups of "real" names. An examination of the source code triggered by this profile reveals that `$unknown` is the name given to the unknown declaration. As we observed in Section 2.5, some improvements could be made to the handling of unknown declarations and apparently this profile shows another symptom. We would also hope to reduce the time spent looking up pre-defined names.

4.2 Structured Attributes

Kiama has *structured attributes* that are groups of attributes that are associated with each other in some way. One kind of structured attribute is a *chain*, which is inspired by a similar construct in the LIGA attribute grammar system [13]. A chain abstracts a pattern of attribution that threads a value in a depth-first left-to-right fashion throughout a tree. The idea is that the system provides the default threading behaviour and the attribute grammar writer can customise the equations at various places in the tree to update the chain value. Kiama chains are implemented by a pair of attributes: one to calculate the value of the chain that comes in to a node from its parent, and one to calculate the value that goes back out of the sub-tree to the parent. The developer can provide functions to transform the incoming value as it heads into a sub-tree or as it leaves the sub-tree.

The Oberon-0 attribute grammar uses a chain to encode an environment that propagates symbol information from declarations to uses. Declarations add information to the chain and uses of names access the chain to look up information. This approach contrasts with the name analysis approach used in the PicoJava example where parameterised attributes are used to search for the declaration information to bring it to the uses where it is needed. Profiles can help us compare these two approaches to this problem. We can profile the PicoJava name analyser along the attribute and parameter dimensions to assess the overhead of looking up each name individually. We can profile the environment attribute in the Oberon-0 compiler to find out the costs of propagating it.

4.3 Derived Dimensions

All of the profiles we have seen so far use the intrinsic dimensions whose values are already present in profile records. A powerful extension is to produce profiles based on dimensions that are derived from the profile records. Derived dimensions summarise the execution in any way we like, including in ways that are specific to the application.

For example, suppose that we are interested in knowing where the attribute evaluations occur within the tree. We might wish to know whether particular attributes are evaluated more at boundary nodes (the root and the leaves) or at the other (inner) nodes. The profile records do not have an intrinsic dimension that gives us this information directly, but we can calculate this new location dimension from the subject dimension.

A simple implementation of the location dimension suffices to return one of the strings "Root", "Inner" or "Leaf", depending on the location of the subject tree node within the tree. Here is a typical profile that results from using the attribute dimension with this new location dimension. We show the profile for the input half of the environment chain from the Oberon-0 attribute grammar.

```
Total Total  Self  Self  Desc  Desc Count Count
   ms     %    ms     %    ms     %           %
  185  26.5    52   7.5   132  18.9  3237  10.0  Leaf
  133  19.0    44   6.3    89  12.8  5455  16.8  Inner
   58   8.3    58   8.3     0   0.0    81   0.3  Root
```

The profile shows that this particular attribute is evaluated quite a lot at the leaves, which we would expect since the names reside at the leaves and attribution associated with them is the primary client of the environment. Evaluation at inner nodes is necessary to transport the environment from the declarations where it is established to the leaves. The many evaluations at the root are more of a mystery, since we would expect to only determine an incoming environment at the root once for each input program, since that value just contains pre-defined names. (Recall that there are fifty-two programs in the full Oberon-0 test suite.) Examination of the compiler implementation shows that attributed trees are transformed and the transformed trees are sometimes re-attributed, so this repeated evaluation at a root location also makes sense. Some of the programs only have one root evaluation as they are erroneous and are not translated.

Location is a very simple derived dimension but the idea can be taken as far as necessary to reveal just those aspects of the execution needed to diagnose a problem or to expose the way that some attribution works. A particularly powerful application of derived dimensions is custom views of attribute values. We can create dimensions to show the values aggregated according to any useful criteria. For example, an attribute might hold a pretty-printed version of some part of the tree. We could aggregate evaluations of that attribute along a dimension that calculates the number of lines in the pretty-printed text. A profile along this derived dimension would allow us to analyse the impact of the complexity of the pretty-printing task on the run-time.

4.4 Attribute Dependencies

It is sometimes desirable to examine the dynamic dependencies that are induced by the application of the attribute equations to a particular input. Dynamic dependencies reveal exactly how the attributes depend on each other in a particular execution. In contrast, static dependencies that can be obtained by examining the equations themselves are an over-approximation of the dependencies that are actually used.

Kiama's *depends-on* derived dimension aggregates attribute evaluations according to the attributes on which they directly depend.[2] For example, here is a profile along the attribute and depends-on dimensions for the idntype attribute from the Oberon-0 compiler. The idntype attribute holds the type of an identifier use. (We omit the timings from this table and just show the bucket counts to save space.)

```
754   2.3  IdnUse(@).entity, NamedType(?).deftype
155   0.5  IdnUse(@).entity
 13   0.0  IdnUse(@).entity, IntExp(?).tipe
 22   0.1  IdnUse(@).entity, RecordTypeDef(?).deftype
  4   0.0  IdnUse(@).entity, AddExp(?).tipe
 24   0.1  IdnUse(@).entity, ArrayTypeDef(?).deftype
  1   0.0  IdnUse(@).entity, DivExp(?).tipe
  1   0.0  IdnUse(@).entity, IdnExp(?).tipe
```

Each row of this table reports a particular pattern of dependence. For instance, the first line says that in 754 cases the attribute depended on the entity attribute of an IdnUse node and the deftype attribute of a NamedType node. The parenthesised characters after the node types indicate where the given node was in the tree relative to attribute evaluation node. An at-sign means at the same node and a question mark means at a node that is not directly connected to the attribute evaluation node.[3]

The profile reveals a number of things about the calculation of the idntype attribute. Firstly, each case involves the entity attribute of the same node, which makes sense since the entity is to what the identifier refers. We find the type in different ways depending on the kind of entity the name represents. Secondly, on 155 occasions the attribute does not depend on any other attributes because the type can be obtained directly from the entity. In the AddExp, DivExp and IdnExp cases the type is obtained from an expression, which will happen if the named entity is a defined constant. Finally, if the name refers to some other kind of entity, its type will be determined from the type of its declaration which will be a ArrayTypeDef.

Profiles of this kind are particularly useful for diagnosing issues with the distribution of test cases. They are a convenient way to gather statistics about how many instances of particular circumstances occur in a test run, since only some kinds of expression are present. For example, from the profile above we might conclude that more test cases are required to ensure that every possible kind of constant expression is tested. If the language supports type definitions other than type aliases, record, and array types, we should add tests for the other cases, since they are missing from the profile.

A dependency profile can be used with node location profiles (Section 4.3) to summarise the pattern of dependence of an attribute with a view to simplifying that attribute's equations. For example, suppose we find that an attribute only depends on itself at its parent node or on nothing at the root. This pattern of dependence is common and can be abstracted out using an attribute decorator [14]. A down decorator takes care

[2] Another dependencies dimension considers all transitive dependencies and generates visualisations for display by GraphViz.

[3] depends-on profiles can also indicate references to the parent node, to children nodes, or to the next or previous nodes in a sequence.

of the transport of the attribute value down the tree from the root to where it is needed. The developer need only explicitly state the computation that should happen at the root. Thus, profiling can assist with refactoring of attribute grammars into simpler forms.

5 Discussion and Related Work

The vast majority of previous work in the profiling area has been concentrated on execution profiles for programs, inspired by systems that include the seminal `gprof` tool [6]. From the perspective of our work, `gprof` and descendant tools collect events that correspond to function calls. The call graph of the program corresponds to our descendant relationship between profile records. Our profile reports are inspired by those of `gprof`. We also distinguish between the time taken by an evaluation and those that it uses, as `gprof` distinguishes between the time taken by a the caller and the callees.

Our approach is more general, since `gprof` and similar tools solely use the function dimension, whereas we show the utility of domain-specific and derived dimensions to provide different views of the execution. `gprof` focuses on execution time and function call counts so issues of efficiency are paramount. In contrast, our flexible dimension-based approach means that the values themselves are often more important than times.

Abstraction in Profiling Systems. Instead of profiling at the function level, our approach raises the level of abstraction. Abstraction increases the generality of the profiling system and significantly reduces the size of the collected data compared to instruction and function-level profilers.

Sansom and Peyton Jones describe a profiler for higher-order functional languages including Haskell [20]. The execution of higher-order programs, particularly lazy ones, is not obvious since the compilation process is non-trivial and execution order often does not correspond clearly to the source code. They allow developers to add "cost centres" that aggregate data in a program-specific manner. Thus, the source-level profile data can be lifted to a higher level. In our case, the data is always at a higher level. Their cost centres are each associated with source code fragments rather than separated into `Start` and `Finish` events as in our approach. Thus, the identification of an abstracted piece of program execution is more flexible in our approach because the two events do not have to be associated with the same source code.

Nguyen *et al.* describe a domain-specific language for automating the regulation of profile data collection, processing and feedback [17]. The language allows some abstraction away from the details of the profile exploration process. However, it is different from our work in that it operates at a low level and is intended for analysing performance of programs and system kernels in a similar fashion to `gprof`.

Rajagoplan *et al.* consider profiling for event-based programs such as graphical user interfaces [19]. Events in their work are intrinsic to the functioning of the system, whereas in our work they are solely part of the profiling system. They look for patterns in the events which allows them to abstract away from the execution somewhat, but they do not consider a general abstraction mechanism.

Systems such as DTrace [16], the Linux Trace Toolkit [29] and the Java Virtual Machine Tool Interface can be used to collect a large amount of trace data about program execution. Some customisation of the data collection is usually possible. For example, DTrace allows the developer to write custom probes that are inserted and executed efficiently by the infrastructure. A general event tracing mechanism of this kind could be used to collect data about attribute evaluation. However, these approaches are quite heavyweight and have a great deal of machine or system dependence to achieve efficiency. Our work shows that a lightweight, simple approach is sufficient for monitoring the execution of language processors.

Bergel *et al.* describe a model-based domain-specific profiling approach [1]. Instrumentation code augments domain model code via an existing event mechanism or by using the host language's reflection framework. Custom profilers use the information gathered to display profiles and visualisations that relate directly to domain data and operations. The reliance on meta-programming features of the framework distinguishes their approach from ours. We pay the price of having to instrument the attribute grammar library code by hand, but as a result we make no demands on the underlying runtime. Our profile library is independent of any particular domain and the reports have a generic format, whereas their profilers are deliberately tailored to particular model code and particular kinds of observations that they want to make.

Profiling Attribute Grammars. Saraiva and colleagues have investigated the efficiency of attribute grammar evaluation approaches, notably as part of work to improve the efficiency of evaluators that are constructed as circular programs [5]. Their experiments focused on course-grained measures such as heap usage, rather than the kind of fine-grained analysis considered in this paper. In earlier work, Saraiva compared the performance of functional attribute evaluators [21]. He examined properties such as hash table size, cache misses and the number of equality tests performed between terms for both full and incremental evaluation of attributes. Some of these measures have analogues in our approach. The implementation appears to be custom to the particular experiment rather than a general facility as in our library.

Söderberg and Hedin show how attribute profiles can be used to analyse caching behaviour in JastAdd [25]. They calculate an *attribute instance graph* that is an attribute dependence graph with evaluation counts on the edges. Edges labelled with counts greater than one point to attributes for which caching might be advantageous. Their attribute dependence graph is similar to our collection of profile records and dependency relationships, except that our records are independent of the attribute evaluation domain. Our use of arbitrary dimensions to extend the power of profiles goes beyond the aim of Söderberg and Hedin's work which was to look solely at caching issues.

We developed the Noosa execution monitoring system for the Eli system, including the attribute grammar component [22]. Noosa is a debugging system, not a profiler, but it also uses an event-based approach to record information about the execution of a program. Noosa doesn't group events in the same way as the Kiama profiler, since it uses events primarily to specify domain-specific breakpoints. The focus is on controlling the execution as it happens rather than on summarising it after it is done. Noosa can be used to examine the values of attributes of interest with reference to the abstract syntax tree, but it cannot be used to summarise the execution along other dimensions.

6 Conclusion and Future Work

We have described a new general approach to domain-specific profiling and its application to profiling dynamically-scheduled attribute grammar evaluators. The approach is easy to implement and its execution overhead is low enough for interactive use. We have described applications of the profiler to attribute grammar understanding, test case coverage, and refactoring.

Our current implementation does not handle Kiama's *circular attributes* that are evaluated until they reach a fixed point. We plan to add support for circular attributes as we have described for regular and parameterised ones. We will also add dimensions that enable the operation of the fixed point computation to be examined. For example, a useful dimension would be the number of times that a circular attribute had to be evaluated before it reached a fixed point. It would also be useful to be able to distinguish the run-time devoted to each iteration in the computation of a circular attribute.

One direction of future work will be to deploy the profiler for use in other attribute grammar systems. The approach used with Kiama should easily transfer to other systems based on dynamic scheduling, but minor modifications should allow it to be used with other evaluation approaches. For example, a statically-scheduled tree walking attribute evaluator could be instrumented automatically by the scheduler. There might be scope to add new dimensions, such as one that captures information about node visits.

We also plan to use our framework to investigate applications of profiling other aspects of language processing. A student is also working on a user interface for interactive and graphical access to the profiling data.

References

1. Bergel, A., Nierstrasz, O., Renggli, L., Ressia, J.: Domain-Specific Profiling. In: Bishop, J., Vallecillo, A. (eds.) TOOLS 2011. LNCS, vol. 6705, pp. 68–82. Springer, Heidelberg (2011)
2. Bouchou, B., Halfeld Ferrari, M., Lima, M.A.V.: Attribute Grammar for XML Integrity Constraint Validation. In: Hameurlain, A., Liddle, S.W., Schewe, K.-D., Zhou, X. (eds.) DEXA 2011, Part I. LNCS, vol. 6860, pp. 94–109. Springer, Heidelberg (2011)
3. Davidson, D., Smith, R., Doyle, N., Jha, S.: Protocol Normalization Using Attribute Grammars. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 216–231. Springer, Heidelberg (2009)
4. Ekman, T., Hedin, G.: The JastAdd extensible Java compiler. In: Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications, pp. 1–18. ACM, New York (2007)
5. Fernandes, J.P., Saraiva, J., Seidel, D., Voigtländer, J.: Strictification of circular programs. In: Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2011, pp. 131–140. ACM, New York (2011)
6. Graham, S.L., Kessler, P.B., Mckusick, M.K.: gprof: A call graph execution profiler. In: Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction, SIGPLAN 1982, pp. 120–126. ACM, New York (1982)
7. Han, F., Zhu, S.-C.: Bottom-up/top-down image parsing with attribute grammar. IEEE Transactions on Pattern Analysis and Machine Intelligence 31(1), 59–73 (2009)
8. Hedin, G.: Reference Attributed Grammars. Informatica 24(3), 301–317 (2000)
9. Hedin, G., Magnusson, E.: JastAdd: an aspect-oriented compiler construction system. Science of Computer Programming 47(1), 37–58 (2003)

10. 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)
11. Karim, M.R., Ryan, C.: A New Approach to Solving 0-1 Multiconstraint Knapsack Problems Using Attribute Grammar with Lookahead. In: Silva, S., Foster, J.A., Nicolau, M., Machado, P., Giacobini, M. (eds.) EuroGP 2011. LNCS, vol. 6621, pp. 250–261. Springer, Heidelberg (2011)
12. Kastens, U.: Ordered attribute grammars. Acta Informatica 13, 229–256 (1980)
13. Kastens, U., Waite, W.M.: Modularity and reusability in attribute grammars. Acta Informatica 31, 601–627 (1994)
14. 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)
15. Magnusson, E., Hedin, G.: Circular reference attributed grammars–their evaluation and applications. Science of Computer Programming 68(1), 21–37 (2007)
16. Mauro, J., Gregg, B., Mynhier, C.: DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD. Prentice Hall Professional (2011)
17. Nguyen, P., Falkner, K., Detmold, H., Munro, D.S.: A domain specific language for execution profiling & regulation. In: Proceedings of the Australasian Conference on Computer Science, pp. 123–132. Australian Computer Society, Inc. (2009)
18. Paakki, J.: Attribute grammar paradigms—a high-level methodology in language implementation. Computing Surveys 27(2), 196–255 (1995)
19. Rajagopalan, M., Debray, S.K., Hiltunen, M.A., Schlichting, R.D.: Profile-directed optimization of event-based programs. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, PLDI 2002, pp. 106–116. ACM, New York (2002)
20. Sansom, P.M., Peyton Jones, S.L.: Formally based profiling for higher-order functional languages. ACM Trans. Program. Lang. Syst. 19(2), 334–385 (1997)
21. Saraiva, J.: Purely Functional Implementation of Attribute Grammars. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands (December 1999)
22. Sloane, A.M.: Debugging Eli-Generated Compilers with Noosa. In: Jähnichen, S. (ed.) CC 1999. LNCS, vol. 1575, pp. 17–31. Springer, Heidelberg (1999)
23. Sloane, A.M.: Lightweight Language Processing in Kiama. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2009. LNCS, vol. 6491, pp. 408–425. Springer, Heidelberg (2011)
24. Sloane, A.M., Kats, L.C.L., Visser, E.: A pure embedding of attribute grammars. Science of Computer Programming (in press, 2012)
25. Söderberg, E., Hedin, G.: Automated Selective Caching for Reference Attribute Grammars. In: Malloy, B., Staab, S., van den Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 2–21. Springer, Heidelberg (2011)
26. 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)
27. Vogt, H.H., Swierstra, S.D., Kuiper, M.F.: Higher order attribute grammars. In: PLDI 1989: Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation, pp. 131–145. ACM Press (1989)
28. Wirth, N.: Compiler Construction. Addison-Wesley (1996) (revised November 2005)
29. Yaghmour, K., Dagenais, M.: The Linux trace toolkit. Linux Journal (May 2000)

Termination Analysis for Higher-Order Attribute Grammars[⋆]

Lijesh Krishnan and Eric Van Wyk

Department of Computer Science and Engineering
University of Minnesota, Minneapolis MN, USA
{krishnan,evw}@cs.umn.edu

Abstract. This paper describes a conservative analysis to detect non-termination in higher-order attribute grammar evaluation caused by the creation of an unbounded number of (finite) trees as local tree-valued attributes, which are then themselves decorated with attributes. This type of non-termination is not detected by a circularity analysis for higher-order attribute grammars. The analysis presented here uses term rewrite rules to model the creation of new trees on which attributes will be evaluated. If the rewrite rules terminate then only a finite number of trees will be created. To handle higher-order inherited attributes, the analysis places an ordering on non-terminals to schedule their use and ensure a finite number of passes over the generated trees. When paired with the traditional completeness and circularity analyses and the assumption that each attribute equation defines a terminating computation, this analysis can be used to show that attribute grammar evaluation will terminate normally. This analysis is applicable to a wide range of common attribute grammar idioms and has been used to show that evaluation of our specification of Java 1.4 terminates.

1 Introduction

Silver [14] is an extensible attribute grammar system designed to support the modular specification of languages. Silver specifications define host and extension concrete syntax as well as domain-specific semantics such as optimizations, transformations, and error-checking. It has been used to write extensible grammars for mainstream languages such as Java 1.4 [15]. Attribute grammars (AGs) were introduced by Knuth in 1968 [8] as a means to assign semantics to syntax trees, by associating tree nodes with named values known as attributes. Once the parser constructs the program syntax tree, the attribute evaluator evaluates its undefined instances one at a time. In 1989 Vogt *et al.* introduced higher-order attribute grammars (HOAGs) [17] in which syntax trees can be computed and passed around a syntax tree as attribute values. These trees may also be decorated with attributes which are then evaluated.

Static analyses of attribute grammars detect problems before evaluation and ensure that the attribute grammar is well-defined. According to Vogt *et al.*,

⋆ This work is partially supported by NSF Awards No. 0905581 and No. 1047961.

$P_R : R \longrightarrow X$

$P_X : X \longrightarrow$
  local $L_A :: A =$ if $f(...)$
    then $P_A()$ else $P_a(a)$

$P_A : A \longrightarrow$
  local $L_X :: X = P_X()$

$P_a : A \longrightarrow a$

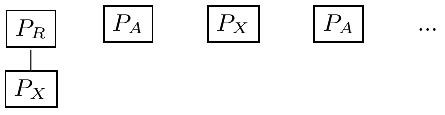


Fig. 1. A higher-order attribute grammar (specified in [17]) for which tree creation may not terminate, and an example of a non-terminating tree creation sequence

a *well-defined* higher-order attribute grammar is one that: is *complete* so that every synthesized, inherited and local instance on a node has a definition; is *non-circular* so that no attribute value depends on itself, either directly or indirectly; and has a *finite number of new trees* created during attribute evaluation. For the first two conditions, Vogt *et al.* specify extended versions of Knuth's tests for completeness and circularity, which prevents abnormal termination of attribute evaluation. But unlike with attribute grammars without higher-order attributes, attribute evaluation for a complete, non-circular HOAG may not terminate since creating an unbounded number of trees during higher-order attribute evaluation can lead to non-termination. For example, Fig. 1 gives a grammar (borrowed from [17]) for which an infinite number of trees may be created. This explains why Vogt *et al.* included the third condition in their definition of well-definedness. (However, "well-defined" is often used to describe grammars that meet only these first two criteria: completeness and non-circularity. Vogt's Ph.D. dissertation adopts this use of the term [16].)

Circularity analysis prevents, among other problems, specifications that would cause the construction of infinite syntax trees, but it does not prevent the creation of an infinite number of *finite* trees, as seen in the example in Fig. 1. Any analysis to detect the creation of an unbounded number of trees will be conservative; consider the conditional expression in the example. An analysis to guarantee that no evaluation sequences exist with infinitely many tree creation steps, combined with the completeness and circularity tests, would be sufficient to ensure higher-order attribution termination. While Vogt *et al.* describe a condition required to ensure non-termination, it does not seem to have been implemented or evaluated. The analysis presented here is the first, to our knowledge, that uses the structure of the equations and not just the attribute dependencies imposed by them in an analysis for termination of tree creation.

In this paper, we fill this gap with a conservative analysis to ensure that tree creation during attribution terminates. This analysis uses rewrite rules and non-terminal orderings to model tree construction and check for the possibility of non-termination of tree construction. The analysis is on a restricted, but useful HOAG class, and assumes a general evaluation model. We use existing

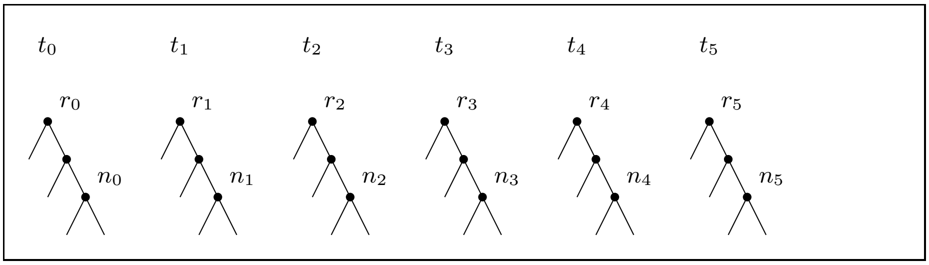


Fig. 2. A tree creation sequence from the original tree t_0, in which each non-initial tree t_{i+1} is created as a local attribute on its predecessor t_i at node n_i

termination analyses on term rewriting systems to show termination of higher-order attribute evaluation. The restrictions and the assumption that the grammar is complete and non-circular mean that for any improper evaluation sequence, there is an infinite tree creation sequence starting from the original tree, in which each non-initial tree is created as a local attribute on its predecessor. In Fig. 1 an example of a non-terminating tree creation sequence is shown for that example grammar. Fig. 2 shows this diagrammatically; here the tree t_{i+1} with root node r_{i+1} is created on node n_i of tree t_i rooted at node r_i. Showing all such sequences terminate would ensure that evaluation terminates normally.

We define a procedure to generate, for an attribute grammar, a set of rewrite rules that model local tree creation during attribution, in the absence of inherited attributes. Thus for each grammar, we generate rules that rewrite trees to the values of local higher-order attributed trees that may be created during attribute evaluation. That is, any sub-tree rooted at n_i can be rewritten by the rules to any tree t_{i+1} that might be the value of one of its local higher order attributes. In other words,

$$\langle \text{sub-tree rooted at } n_i \rangle \Longrightarrow^{+} t_{i+1}$$

For example, the (non-terminating) rules generated for the grammar in Fig. 1 are as follows:

$$\{\ P_X() \Longrightarrow P_A(),\ P_X() \Longrightarrow P_a(a),\ P_A() \Longrightarrow P_X()\ \}$$

We can show that for any tree creation sequence arising from the evaluation of higher-order attributes, we can construct a rewrite sequence of the same length. Thus if the rules terminate, no infinite tree creation sequence exists. We thus have a guarantee of termination of tree creation, and thereby of attribute evaluation, in the absence of higher-order inherited attributes. We use existing tools such as AProVE [5] to verify that the rules terminate.

The problem of non-terminating higher-order *inherited* attribute evaluation is handled by ordering the grammar non-terminals so that the type of each inherited higher-order attribute value is "smaller" than the non-terminal it decorates. In all other cases of higher-order attribute evaluation, the attribute type is no "larger" than the non-terminal of the node on which it was created. The existence

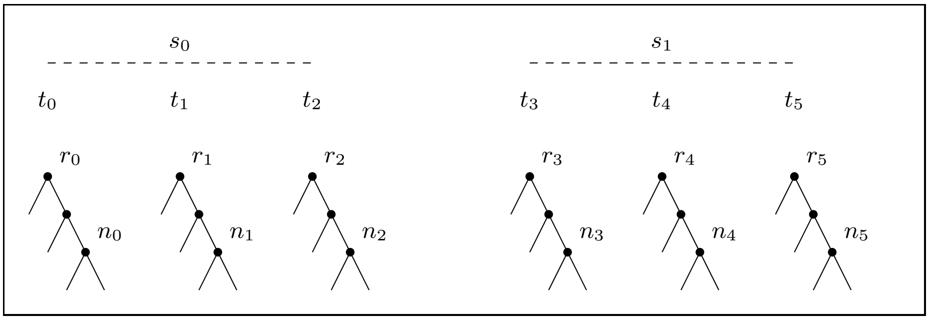


Fig. 3. A tree creation sequence from the original tree t_0, in which each non-initial tree t_{i+1} is created as a local attribute on its predecessor t_i. The sequence consists of a sequence of sub-sequences marked by the use of inherited attributes.

of such an ordering ensures that the number of higher-order inherited accesses in any tree creation sequence is finite. An infinite tree creation sequence such as that shown in Fig. 1 and modeled in Fig. 2 consists of a sequence of tree creation sub-sequences in which the non-terminals at the roots of the trees in each sub-sequence, are of the same "size", as shown in Fig. 3. Within such "constant" tree creation sub-sequences, the rewrite rules can still be used to model tree creation, so that if the rules terminate, then every such constant sub-sequence must terminate. This implies, once again, that no infinite tree creation sequence exists. Thus if the grammar's rules are terminating and its non-terminals can be ordered as desired, then attribute evaluation will terminate.

Contributions and Outline: Our primary contribution is an analysis that detects potential non-termination of tree construction in higher-order attribute grammars. This analysis is based on rewrite rules and an ordering on grammar non-terminal symbols to ensure termination of tree construction and is, to the best of our knowledge, the first of its kind. We also evaluate this analysis on the ableJ specification [15], an attribute grammar specification for Java 1.4.

This paper is structured as follows. Section 2 presents background material on higher-order attribute grammars and describes the restricted class of grammars handled by the termination analysis. Section 3 describes how non-termination of attribute evaluation is equivalent to disproving the existence of infinite local tree creation sequences. Section 4 describes how term rewriting rules can be used to model tree creation in the absence of higher-order inherited attributes. Section 5 describes how the analysis handles higher-order inherited attributes by constructing an ordering on the non-terminals that limits the number of inherited accesses. A discussion and evaluation are provided in Section 6. Section 7 describes related work. Finally, Section 8 discusses future work and concludes.

2 A Restricted Class of Higher-Order Attribute Grammars

In this section we describe RHOAGs, the class of higher-order AGs handled by our termination analysis. The design of this class is driven by two main goals. First, it is simple enough to conveniently illustrate the main ideas behind the analysis, but still includes most of Silver's important features and is expressive enough to specify rich, complex grammars. Our attribute grammar for Java 1.4, ABLEJ [15], was converted to this form with a few modest modifications. Second, the restrictions allow us to reduce the termination problem to that of disproving infinite local tree creation sequences. Many features in Silver (such as forwarding [13]) can be translated into this form without a loss of expressivity.

In Fig. 4, we give a formal definition of the attribute grammars in RHOAGs. As in standard AGs [8], there are restrictions on which attribute occurrences may be defined on a given production. Synthesized and local occurrences are defined on their node's production. Inherited occurrences are defined on the production of their node's parent, unless their node is the root of a tree evaluated as a local attribute. In the latter case, the inherited occurrence is defined on the production of the local's parent node on which the local was evaluated.

Unlike standard grammars, definitions and expressions in RHOAGs are further restricted in the following ways. There are no inherited occurrences on the root node, for simplicity. A production may not refer to its own synthesized occurrences, or its children's inherited occurrences; since it computes these values there is no need to do so. Attributes may only be accessed on a node's children, its locals, or its own tree (in the case of inherited attributes). All other kinds of accesses, such as nested accesses of attributes on attributes, are disallowed by the syntax, allowing us to isolate the creation of new syntax trees during evaluation to local attribute evaluation. Functions are primitive and do not take trees as arguments or return them as results.

We use a number of additional terms and notations that we briefly describe here. Trees are typically referred to in this analysis by their root nodes, which are elements of the set N of tree nodes. Attribute instances have the type $N \times A$. We write $n\#a$ for an instance of attribute a on node n in some tree. A *tree term* or simply *term* is a textual representation of a tree that consists of the production name constructing its root node and the terms of its child trees enclosed in parentheses and separated by commas. This is similar to the conventions of algebras and term-algebras. For example, "$P_R(P_X())$" is the term denoting the first tree in Fig. 1. We use $term(n)$ to denote the term representing a tree rooted at node n. The set of terms is denoted by $Term$. An *attribution* of a tree, Γ, maps attribute instances to their values. An attribute value may be an element of the set PV of primitive values, a tree term in $Term$, a tree node in N, or be (as of yet) undefined, in which case it is denoted by $\perp$.

Fig. 5 gives an informal description of the process of higher-order attribute evaluation for a given syntax tree t. Attribute evaluation starts with a syntax tree, usually constructed by a parser, with all of its attribute instances undefined ($\perp$). Attribution begins by evaluating attributes whose definitions are constants,

$G = \langle NT, T, P, S\rangle$ is a context-free grammar.

– NT is a set of non-terminals.
– T is a set of terminals.
– P is a set of productions with signatures of the form $NT \to (NT \cup T)^*$.
– $S \in NT$ is a start non-terminal.

$AG = \langle G, PT, PV, A, @, F, D\rangle$ is an attribute grammar.

– PT is a set of primitive types.
– PV is a set of primitive values with types in PT.
– $A = A_S \cup A_I \cup A_L$ is a set of attributes.
  - A_S is a set of synthesized attributes of type $NT \cup T \cup PT$.
  - A_I is a set of inherited attributes of type $NT \cup T \cup PT$.
  - A_L is a set of local attributes of type NT.
– $@ \subseteq ((A_S \cup A_I) \times NT) \cup (A_L \times P)$ is the occurs-on relation.
– F is a set of primitive functions with type signatures of the form $PT^* \to PT$.
– D returns the set of attribute definitions associated with a given production.

For a production $p \in P$ with signature $X_0 \longrightarrow X_1 ... X_{n_p}$, $D(p)$ contains definitions of the form

– $X_0.a_S = E$ where $a_S \in A_S, a_S@X_0$
– $X_i.a_I = E$ where $1 \leq i \leq n_p, a_I \in A_I, a_I@X_i$
– $l = E$ where $l@p$
– $l.a_I = E$ where $l@p, a_I \in A_I, a_I@L_i$ and L_i is the type of l

The expressions E on the right-hand sides of these definitions have the form

– X_i where $0 \leq i \leq n_p$
– $X_0.a_I$ where $a_I \in A_I, a_I@X_0$
– $X_i.a_S$ where $1 \leq i \leq n_p, a_S \in A_S, a_S@X_i$
– $l.a_S$ where $l@p, a_S \in A_S, a_S@L_i$ and L_i is the type of l
– c where $c \in T$
– $q(E_1, ..., E_{n_q})$ where $q \in P$
– $f(E_1, ..., E_{n_f})$ where $f \in F$
– `if` E_C `then` E_T `else` E_E

Fig. 4. Formal definition of the restricted class RHOAG of higher-order AGs

or which depend solely on attributes on terminal symbols (such as *lexeme*) which are set by the parser. In each evaluation step, an undefined *evaluable* occurrence (*i.e.*, one whose required attribute instances have all been evaluated) is selected and evaluated. Instances are evaluated one at a time until there are no more undefined evaluable attribute instances. If the grammar passes the circularity and completeness tests, the process will end only when all attribute instances have been evaluated. We assume that during attribute evaluation, attribute accesses, function calls and tree-creating steps terminate atomically with valid values. Conditional expressions are evaluated lazily as expected.

```
SET $T$ to $\{ t \}$.
WHILE there is an evaluable attribute occurrence $n\#a$ in a tree in $T$

  – IF $a$ is a synthesized or local attribute
      • SET $e$ to $n\#a$'s defining expression $e$, specified in $n$'s production.
  – ELSE
      • SET $e$ to $n\#a$'s defining expression $e$, specified in $n$'s parent node's production.
  – SET $v$ to the evaluated value of $e$.
  – IF $a$ is a local attribute
      • ADD an attributed version of the tree term $v$ to $T$.
      • SET $n\#a$ to the root of this tree.
  – ELSE
      • SET $n\#a$ to $v$.
```

Fig. 5. An informal description of the process of attribute evaluation process for higher-order attribute grammars

The semantics of local attribute evaluation is different from the semantics of other kinds of higher-order attribute evaluation (i.e., synthesized or inherited occurrences). When a local instance is evaluated, the computed tree term value is converted into a full-fledged syntax tree with its own (undefined) attribute instances. This new tree is added to the set of trees that defines each evaluation state. Further, new trees are added only as local attributes due to syntactic restrictions. Attribute evaluation terminates only when all instances on all trees are defined. When an unbounded number of trees are created there are an unbounded number of attribute instances. This leads to the type of non-termination that is detected by the analysis presented here and not detected by higher-order circularity analyses.

A *proper evaluation sequence* terminates with a valid attribution to every attribute instance in the original tree, and any tree created during evaluation. An *improper evaluation sequence* either terminates abnormally due to absent definitions, circularities in attribute definitions, non-terminating function calls, or other errors in the grammar definition; or does not terminate due to the creation of an infinite number of trees. Our evaluation model is more general than most standard evaluators, including Silver's. This means that for a given syntax tree, while any attribute evaluation sequence possible in Silver is also included in this model, there are sequences in this model that are not possible in Silver. Silver, and other current systems like JastAdd [4] and Kiama [12], use a *demand-driven* attribute evaluation algorithm in which only the attribute values that are needed are computed. The model presented here is simpler but more general and includes all possible models of attribute grammar evaluation. It is therefore possible that for an input tree, evaluation is non-terminating in this model, while it is terminating in Silver.

3 Reducing Infinite Evaluation to Infinite Tree Creation

We can reduce the problem of guaranteeing termination of attribution to that of disproving the existence of infinite tree creation sequences.

Theorem 1: For a tree in a complete, non-circular grammar with terminating functions, if there is an improper evaluation sequence, then there is an infinite tree creation sequence.

For a complete, non-circular grammar with terminating functions, any evaluation sequence that does not terminate normally contains an infinite number of local tree creation steps. This is because at every step of a valid evaluation sequence, either every tree is attributed, or there is an undefined evaluable attribute occurrence (because the definitions are non-circular). This means that an evaluation sequence that does not terminate normally has an infinite number of evaluation steps. It must further evaluate an infinite number of attribute instances, since in every step, an undefined instance is evaluated. As each tree has only a finite number of attribute instances, an infinite number of trees must be created via local creation steps.

The syntax trees at each evaluation step can be organized into a tree of trees, known as the *tree of locals* (TOL). Each node of a TOL represents a syntax tree. The root node represents the original program syntax tree. A TOL node's child nodes are the local trees that have been created on any of the nodes of its own syntax tree. A path in a TOL is a *tree creation sequence*, i.e., a sequence of trees starting from the program syntax tree, in which each non-initial tree is the value of a local that has been evaluated on its predecessor. An evaluation state's syntax trees can be organized into a TOL since every local is created on an existing tree and every local is evaluated only once. The TOL of the initial evaluation state contains one node, labeled with the program syntax tree. The TOL of each later step either is the same as that of its previous step, or has one extra node and edge. In every local-evaluating step, a node for the new syntax tree is added to the TOL, as a child of the TOL node corresponding to the local's parent syntax tree. In all other steps, the TOL stays the same, though the attribution to an instance of an existing syntax tree changes. The sequences of trees in Fig. 1 and Fig. 2 are tree creation sequences and are paths in the TOL.

Thus for an evaluation sequence with an infinite number of local tree creation steps, we can construct an infinite trees of locals, with increasing numbers of nodes and edges. Each TOL node has a finite number of children, as each syntax tree has a finite number of AST nodes, each with a finite number of locals. König's Lemma states that an infinite tree in which each node has a finite number of children, has an infinite path. Thus corresponding to an infinite sequence in a TOLs with increasing numbers of nodes and edges, we can construct an infinite path of trees starting from the original tree, in which each non-initial tree is created as a local on its predecessor. Theorem 1 is proved formally in [9] and is the topic of much of the rest of this paper.

4 Modeling Tree Creation with Rewrite Rules

We use rewrite rules to model tree creation so that termination of the rewrite rules implies termination of all tree creation sequences. For a given grammar, we derive a set of rewrite rules based on the higher-order attribute definitions in its productions. Each generated rule is of the form $p(x_1, ..., x_{n_p}) \Longrightarrow r$ where

- p is a production in P with signature $X_0 \longrightarrow X_1...X_{n_p}$,
- $x_1, ..., x_{n_p}$ are rewrite rule variables, and
- the rule's right-hand side r contains production names, terminal symbols and rewrite rule variables.

For a production $p \in P$ with signature $X_0 \longrightarrow X_1...X_{n_p}$, we generate a set of rules of the form $\{ \ p(x_1, ..., x_{n_p}) \Longrightarrow r \mid r \in R(e) \ \}$ for every expression e that is the RHS of a higher-order definition in $D(p)$. Here $R(e)$ returns a set of rule right-hand sides for a given higher-order expression e, as defined below:

e	$R(e)$
X_i	$\{x_i\}$
$X_i.a_S$	$\{x_i\}$
$X_0.a_I$	$\{\texttt{INH}\}$
$l.a_S$	$R(e_L)$ where $(l = e_L) \in D(p)$
$q(e_1, ..., e_{n_q})$	$\{q(r_1, ..., r_{n_q}) \mid \ r_i \in R(e_i), \ 1 \leq i \leq n_q\}$
c	$\{c\}$
`if` e_C `then` e_T `else` e_E	$R(e_T) \cup R(e_E)$

A set of rewrite rules is created for each production in the grammar, based on its attribute definitions. Each rule has a different RHS based on the RHS of the higher-order attribute definitions. Explicit tree creation sub-expressions containing production names, terminal symbols and signature variables are represented as such in the rules' right hand sides. Conditional expressions are handled by generating separate rules for each sub-expression, which means multiple rules may therefore be generated for a single definition or sub-expression. Accesses to synthesized occurrences on a child are represented by the child's signature variable. Accesses to synthesized occurrences on local attributes are replaced by rewrite sub-terms generated from the local's definition. Local attributes are thereby inlined when generating rewrite rules; this is possible for non-circular attribute grammars. Function symbols are not present in higher-order sub-expressions. The rules thus retain production names and the structure of higher-order values, but do not keep track of the specific attribute instances that are accessed in each tree-creating expression. They abstract away local attributes, conditional expressions and specific attribute instance names, to generate a much simpler, albeit approximate model of the tree creation process. Note that while we generate rewrite rules for inherited accesses, these are meant to be place-holders. The

```
synthesized attribute pp :: String occurs on Stmt, Expr ;
production doWhile s::Stmt ::= s1::Stmt e::Expr {
   s.pp = "do " ++ s1.pp ++ " while ( " ++ e.pp ++ " );";
   local attribute fs::Stmt = consStmt (s1, while (e, s1));
}
production ifThen s::Stmt ::= e::Expr s1::Stmt {
   s.pp = "if ( " ++ e.pp ++ " ) " ++ s1.pp;
   local attribute fs::Stmt = ifThenElse (e, s1, emptyStmt ());
}
production while s::Stmt ::= 'while' '(' e::Expr ')' s1::Stmt { ... }
production consStmt s::Stmt ::= s1::Stmt s2::Stmt { ... }
production assign  s::Stmt ::= id::Id_t '=' e::Expr ';' { ... }
production ifThenElse
  s::Stmt ::= 'if' '(' e::Expr ')' s1::Stmt 'else' s2::Stmt { ... }
production emptyStmt s::Stmt ::= ';' { ... }
```

- doWhile $(s_1,\ e) \Longrightarrow$ consStmt $(s_1,$ while $(e,\ s_1))$
- ifThen $(e,\ s_1) \Longrightarrow$ ifThenElse $(e,\ s_1,$ emptyStmt $())$

Fig. 6. A fragment of a higher-order attribute grammar, and the rules generated to model tree creation during evaluation

rules do not model higher-order inherited attribute evaluation. This is handled separately as described in Sec. 5.

Fig. 6 shows a fragment of a higher-order attribute grammar, and the rules generated for its definitions. The left hand side of each rewrite rule is a simple production pattern consisting of the production name and a variable in each child slot. It rewrites tree (sub-) terms with that production at the root (the local's parent sub-tree) to the evaluated values of local tree terms. Thus the rules can be used to derive the value of each evaluated local from its parent's sub-tree. This means that for any tree creation sequence, we can construct a rewrite sequence of the same length. Thus if the rewrite rules terminate, then no infinite tree creation sequence exists. We use existing tools such as AProVE that check term rewriting systems for termination to see if the generated rules terminate.

In any tree creation sequence, every non-initial tree can be derived via a non-empty rewrite sequence, from the sub-tree of its predecessor on which it was evaluated.

Lemma 1: Given a tree creation sequence $t_0,\ t_1,\ t_2, ...,$ if $n_i \in \mathit{nodes}(t_i)$ is the node on which tree t_{i+1} with root node r_{i+1} is evaluated, we have $\mathit{term}(n_i) \Longrightarrow^{+} \mathit{term}(r_{i+1})$.

A proof for this theorem is given in [9]. The intuition behind it can be understood by referring to Fig. 1 and breaking each sequence into two parts, the first term, and the rest of the sequence generated from that first term by the rewrite rules. The first term in this generated rewrite sequence represents the tree value after all conditions in its defining expression have been evaluated. The rules model conditional expressions by generating multiple rules corresponding to the two clauses in the expression. The first term thus is a partially evaluated version of the expression in which all conditions have been evaluated, but in which attribute instance accesses are not computed and are represented by the sub-trees on which they are evaluated. This term is derivable from the tree term of the attribute instance's defining node, via the one rule that corresponds to the actual value of the conditional expression. In the first term in the sequence, attribute accesses off children or locals in the tree's defining expression are represented by the rewrite term of the child or local, respectively. Since the generated rules also model the evaluation of these attribute instances of the child or local tree terms, additional rewrites can be performed on these sub-terms to generate the actual evaluated attribute instance value. The fact that these additional rewrites can be performed to generate the correct value can be shown inductively on the structure of the defining expression. The inductive assumption is that all previously evaluated tree values have corresponding valid rewrite sequences from their parent sub-trees to the created values.

Given that we can construct sequences that derive each tree from its predecessor, we can construct a rewrite sequence that models the entire tree creation sequence. As each tree is evaluated on a node of its predecessor, the first term of the rewrite sequence that models this tree creation step is a sub-term of the predecessor tree. Thus the predecessor tree itself can be rewritten to a term in which the parent sub-tree is replaced by the local. This new term contains the new local as a sub-term, and therefore the process can be continued for the next evaluated local. As the rewrite sequence corresponding to each local's evaluation is non-empty, we can construct a rewrite sequence to model the entire tree creation sequence that is at least as long as the tree creation sequence. We can state this formally as the following theorem (proved in [9]).

Lemma 2: For a tree creation sequence $t_0,\ t_1,\ t_2, ...,$ there is a function $R : N \longrightarrow \mathit{Term}$ so that for $i > 0,\ R(t_{i-1}) \Longrightarrow^{+} R(t_i)$.

Thus we have the following theorem for the case in which there are no higher-order inherited attributes.

Theorem 2: If the rewrite rules generated for a complete, non-circular grammar terminate, no improper evaluation sequence exists, in the absence of higher-order inherited attributes.

The proof is by contradiction. By Theorem 1, for any improper evaluation sequence for a tree t_0, there is an infinite tree creation sequence $t_0,\ t_1,\ t_2,\ \ldots$ By Lemma 2, we can define a function R so that for $i > 0,\ R(t_{i-1}) \Longrightarrow^{+} R(t_i)$. Thus there is an infinite rewrite sequence using the generated rules. Therefore the rewrite rules are non-terminating, which is a contradiction. Thus, if the rules terminate, there is no improper evaluation sequence.

5 Ordering Trees to Limit Inherited Accesses in Tree Sequences

The term rewriting rules described in the previous section cannot model tree creation in the presence of higher-order inherited attributes. Rules containing just production names and child variables cannot accurately model tree creation with inherited attributes. They do not incorporate the contextual information required to evaluate inherited attributes (attributes on the parent node). Since we can no longer derive each local from its parent's sub-tree, Lemma 1 no longer holds. Therefore, we can no longer construct a rewrite sequence corresponding to each tree creation sequence, and so Lemma 2 no longer holds.

We present an analysis that checks the grammar for restrictions that ensure that the number of inherited accesses in a tree creation sequence is bounded. We have shown above that rewrite rules model tree creation sub-sequences in which there are no inherited attributes. Thus if the rewrite rules terminate and the grammar satisfies the restrictions that ensure finite inherited accesses, then there are no infinite tree creation sequences and hence evaluation terminates.

The restrictions check if there is an ordering $\succeq$ on the non-terminals in the grammar that ensures a finite number of tree creation steps using higher-order inherited attributes.

Ordering Non-Terminals to Limit Inherited Attribute Accesses: Suppose there exists a well-founded, reflexive and transitive ordering $\succeq$ on the grammar's non-terminals that satisfies the conditions enumerated below. Fig. 7 gives a formal specification of the desired non-terminal ordering.

1. Non-terminal symbols are non-increasing from the root node of a syntax tree to its leaves. In other words, if there is a production with X as its left-hand side and a right-hand side containing Y, then $X \succeq Y$.
2. The root non-terminal of any tree value is no larger than the node on which it was evaluated. Thus, if a local attribute of type Y occurs on a production with LHS X, then $X \succeq Y$. Similarly, if a synthesized attribute of non-terminal type Y occurs on X, then $X \succeq Y$.
3. Finally, if an inherited attribute of non-terminal type Y occurs on X, then X is larger than Y. That is, inherited occurs-on declarations are non-circular.

$\succeq$ is a relation on NT such that

1. $\succeq$ is reflexive.
2. $\succeq$ is transitive.
3. $\succ$ is well-founded.
4. $X \succeq Y$ if
   $(\exists p \in P \;.\; X = \texttt{lhs}(p),\; Y \in \texttt{rhs}(p)) \vee$
   $(\exists p \in P,\; l@p \;.\; X = \texttt{lhs}(p),\; Y = \texttt{type}_{\texttt{a}}(l)) \vee$
   $(\exists a_S \in A_S \;.\; a_S@X,\; Y = \texttt{type}_{\texttt{a}}(a_S))$.
5. $X \succ Y$ if $\exists a_I \in A_I \;.\; a_I@X,\; Y = \texttt{type}_{\texttt{a}}(a_I)$.

Fig. 7. An ordering on a grammar's non-terminals that ensures that the trees in any tree creation sequence are non-increasing

where

- $X \approx Y$ is the same as $X \succeq Y \wedge Y \succeq X$.
- $X \succ Y$ is the same as $X \succeq Y \wedge \neg X \approx Y$.
- $t_1 \succeq t_2$ is the same as $\mathit{symbol}(t_1) \succeq \mathit{symbol}(t_2)$. In other words, trees are compared using their root non-terminals.

Constructing the Desired Non-Terminal Ordering: Below we specify a sound and terminating procedure that, for a given attribute grammar, attempts to define an ordering on its non-terminals that satisfies the conditions in Fig. 7. Before doing so we introduce the following abbreviated notations to specify the conditions that the constructed ordering must satisfy.

- $S(X,\; Y) \equiv (\exists p \in P \;.\; X = \texttt{lhs}(p),\; Y \in \texttt{rhs}(p)) \vee$
  $(\exists p \in P,\; l@p \;.\; X = \texttt{lhs}(p),\; Y = \texttt{type}_{\texttt{a}}(l)) \vee$
  $(\exists a_S \in A_S \;.\; a_S@X,\; Y = \texttt{type}_{\texttt{a}}(a_S))$.
- $I(X,\; Y) \equiv \exists a_I \in A_I \;.\; a_I@X,\; Y = \texttt{type}_{\texttt{a}}(a_I)$.

We need a procedure that constructs an ordering $\succeq$ as required by Fig. 7 on a given attribute grammar's non-terminals so that for any $X, Y \in NT$, $S(X,\; Y)$ implies $X \succeq Y$ and $I(X,\; Y)$ implies $X \succ Y$. Procedure A in Fig. 8 gives such a procedure.

The intuition behind Procedure A can be understood as follows. For any tree creation sequence, we can construct a path in G_{SCC} of the nodes that correspond to the strongly connected components (SCC) of the non-terminal symbols at the roots of the trees in the sequence. For "non-inherited" tree creation steps, we either stay at the same SCC node (in those cases in which $X \approx Y$), or move to another SCC node (in those cases in which $X \succ Y$). For all other tree creation steps, we move to another SCC node. Thus if the graph has no cycles, then there can only a finite number of "inherited" tree creation steps.

Procedure A:

1. Construct a directed graph G_{NT} whose vertexes correspond to the non-terminals in NT, and with an edge $\langle X,\ Y\rangle$ if and only if $S(X,\ Y)$.
2. Construct a directed graph G_{SCC} whose vertexes correspond to the strongly connected components (SCC) of G_{NT}, and with an edge $\langle S_X,\ S_Y\rangle$ if and only if there exist $X \in S_X$ and $Y \in S_Y$ where either $I(X,\ Y)$, or $S(X,\ Y)$ and $\neg S(Y,\ X)$.
3. If G_{SCC} has cycles, return failure, the required ordering doest not exist.
4. The desired ordering is defined as follows: $X \succeq Y$ if vand only if there is a (possibly trivial) path in G_{SCC} from X's SCC to Y's SCC. Note that this means that $X \approx Y$ if and only if X and Y are in the same SCC in G_{NT}.

Fig. 8. A procedure that, for a given attribute grammar, attempts to define an ordering on its non-terminals that satisfies the conditions in Fig. 7

Lemma 3: The ordering $\succeq$ generated by Procedure A for a given attribute grammar satisfies the conditions in Fig. 7.

As an example, we consider an attribute grammar for a simple imperative language. Syntax trees for programs in this language have non-terminals that include statements (`Stmt`) which derive expressions (`Expr`) and types (`Type`). The symbol table for looking up variable names is implemented as a higher-order inherited attribute of type `Env` that decorates statements and expressions, as expected. Since `Env` occurs as an inherited attribute on the non-terminals appearing in the syntax tree, the algorithm in Fig. 8 computes the following:

$$\texttt{Stmt} \approx \texttt{Expr},\ \texttt{Stmt} \approx \texttt{Type} \text{ and } \texttt{Stmt} \succ \texttt{Env} \text{ (and } \texttt{Expr} \succ \texttt{Env},\ \texttt{Type} \succ \texttt{Env})$$

If such a language also allows names to be bound to types, as in the `typedef` construct in C, we might add a type environment (`TypeEnv`) that maps type names to type representations (`TypeRep`). If this type environment was an inherited attribute on the environment (that would be used to look up type names that were bound to variables names in `Env`) then it would add another level to the ordering of non-terminals, resulting in the following:

$$\texttt{Env} \succ \texttt{TypeEnv} \text{ and } \texttt{Env} \succ \texttt{TypeRep} \text{ and } \texttt{TypeEnv} \approx \texttt{TypeRep}$$

Proving termination: If such an ordering exists, then every tree created as a local attribute is no larger (with respect to $\succeq$) than the tree on which it was created. Thus the trees in any tree creation sequence (corresponding to a path in the tree of locals) are non-increasing, as stated in the following lemma.

Lemma 4: Assume $\succeq$ exists as described in Fig. 7. For any evaluation sequence with a tree creation sequence $t_0,\ t_1,\ t_2, ...$ we have $t_i \ \succeq\ t_{i+1}$, for all $i \geq 0$.

We define a *constant* tree creation sequence as a tree creation sequence in which there are no steps in which the generated tree is strictly smaller (with respect to $\succeq$) than its parent tree. It is thus a tree creation sequence $t_0, t_1, t_2, ...$ where $t_0 \approx t_1 \approx t_2 \approx ...$

Lemma 5: If $\succeq$ exists as described in Fig. 7, then for any infinite tree creation sequence, there exists an infinite constant tree creation sequence.

Since the trees in a constant tree creation sub-sequence are not created using inherited attribute accesses, we can show that the rewrite rules defined in the previous section are sufficient to model such sub-sequences.

Lemma 6: For a constant tree creation sequence $t_0,\ t_1,\ t_2,\ ...$, there is a function $R : N \longrightarrow \mathit{Term}$ so that for $i > 0,\ R(t_{i-1}) \Longrightarrow^+ R(t_i)$.

The proof of this is similar to that for Lemma 2. The only additional complexity is that the rewritten terms are "pruned" versions of the actual tree terms, in which all sub-trees rooted at nodes less than the roots of trees in the constant tree creation sequence, are replaced by the `INH` place-holder term. In effect, we ignore the parts of the tree with non-terminals less than the roots of the trees in the constant tree creation sequence. This is necessary as the rewrite rules, lacking the context in which inherited attributes are evaluated, represent inherited accesses with the placeholder `INH`. Thus if higher-order inherited attributes are present, we can no longer generate the actual local trees using the rules. We can prove inductively that for constant tree creation steps, we can rewrite the pruned parent sub-tree tree to the pruned new local tree (that is not `INH`) via a non-empty rewrite sequence. The full proof is given in [9].

Thus if there exists an infinite constant tree creation sequence, then there exists an infinite rewrite sequence. So if the rules terminate, then all constant tree creation sequences terminate. Ordering non-terminals in this way therefore allows us to limit the number of inherited accesses in a tree creation sequence, and then use rewrite rules to model the parts of the sequence that do not use inherited attributes. We thus have the following theorem to show termination in the presence of higher-order inherited attributes.

Theorem 3: If the rewrite rules generated for a complete, non-circular grammar terminate, and the non-terminals can be ordered as desired, no improper evaluation sequence exists.

The proof is by contradiction. By Theorem 1, for any improper evaluation sequence for a tree t_0, there is an infinite tree creation sequence $t_0,\ t_1,\ t_2,\ ....$ If the non-terminals can be ordered as required, then for an infinite tree creation sequence, there is an infinite constant tree creation sequence $t'_0,\ t'_1,\ t'_2,\ ...$, by Lemma 5. By Lemma 6, we can define a function R so that for $i > 0,\ R(t'_{i-1}) \Longrightarrow^+ R(t'_i)$. Thus there is an infinite rewrite sequence using the generated rules. Therefore the rewrite rules are non-terminating, which is a contradiction. Thus if the rules terminate, and the non-terminals can be ordered as desired, there is no improper evaluation sequence.

Proofs of all theorems and their supporting lemmas are given in [9].

6 Discussion and Evaluation

The inspiration for using rewrite rules to model tree creation comes from a feature in Silver that can be rewritten using standard higher-order attribute grammars and thus is not part of the RHOAGs class. This feature is *forwarding* [13] and it has some similarities to rewriting in an attribute grammar setting. To use forwarding, a production in the grammar computes a new syntax tree and designates it as being "semantically equivalent" to the construct defined by the production. For example, the `doWhile` production in Fig 6 creates the local tree `fs` that is the translation of a do-while construct, as a sequence of the loop body and a while-loop with the same condition and body. This encodes the idea that "`doWhile(`c`) {`s`}`" is equivalent to "s`; while(`c`) {`s`}`". To use forwarding, this production could designate the `consStmt(s1, while(e, s1))` tree as its forwards-to tree. During attribute evaluation, if the "forwarding" production does not explicitly specify a definition for an attribute that is requested, then the production "forwards" that attribute query to the semantically equivalent forwards-to tree. The attribute is evaluated on that tree and returned to the original forwarding-tree, to be returned as the value of the attribute for the original query.

The concept of forwarding is useful in extensible language design [15], especially in avoiding defining attributes that can more easily be defined by translation to a "core" language. For example, an attribute used to translate this sample language to machine code may be defined on the `while` production, but not on the `doWhile` production that forwards to the equivalent while-loop construct described above. In some sense, forwarding is non-destructive rewriting of the forwarding-tree to the forwarded-to-tree for attribute evaluation. Our initial interest was in showing that the process of forwarding would eventually terminate and forwarding's resemblance to rewriting led us in this direction. Forwarding can be translated away to higher-order attributes if extra "copy" rules are generated for each missing synthesized attribute equation that was handled by forwarding. For this analysis, it is possible to do this transformation without the loss of accuracy. Note that for other analyses, such as for completeness and circularity, translating away forwarding results in less precise analyses and thus this is not done for those analyses [13].

The problem of showing termination, whether for attribute evaluation or term rewriting systems, is undecidable, so our goal is an approximate solution. To some extent, it is similar to showing termination of the attribute evaluator program written in a programming language such as Haskell or Java. Silver translates attribute grammar specifications to Java and we could attempt to analyze that generated code. However analysing such lower-level implementations would discard much of higher-level domain-specific information about AGs that is useful in the analysis. We desire a simpler abstraction on which to perform analyses such as termination, one that incorporates our domain knowledge of AGs as well as information gained from our implementations of AG specifications for real-world programming languages. Our experience with grammars such as ableJ shows us that sticking to a few simple and reasonable guidelines regarding higher-order attribute definitions, will guarantee termination of higher-order attribution.

The design of the generated rewrite-rules that model tree creation is guided by several factors. Since a constantly failing analysis would technically be sufficient in a conservative analysis, the generated rules must be useful and be able to show termination of sophisticated grammars such as ableJ. Finally, we must be able to formally show that the rules are correct for our purposes; if the generated rules are terminating, then there are no infinite tree creation sequences and therefore attribution in general terminates. The nature of the problem requires us to walk a fine line between developing an analysis that is simple enough to be easily proven correct and efficiently executed, but is not so imprecise that it cannot show termination for a large class of useful and interesting grammars. In our experience with ableJ the rules we define satisfy both these conditions.

The termination analysis is conservative, in that there are terminating grammars for which the analysis fails. This follows first from the fact that both AProVE and the ordering generator are conservative. Further, the constructed rewrite rules are conservative in how they model tree creation since they derive more terms than are actually generated during higher-order evaluation. For example, attribute instance names are not retained in the sub-terms corresponding to synthesized attribute accesses. Thus the rules derive sequences corresponding to the evaluation of every possible higher-order synthesized instance on the child or in-lined local. Only one of these will correspond to the actual run-time evaluation sequence in which a particular attribute instance is evaluated. Similarly, only one of the rules generated for the conditional expressions in a definition will model the tree created once all conditions are evaluated. While the rules are approximate in that they may not be able to show termination for all terminating grammars, they are correct. They can show termination for non-trivial grammars such as ableJ in a reasonable amount of time.

While the class RHOAG is expressive enough to include grammars similar to ableJ, there are some grammars and Silver features that are beyond the scope of our analysis. Firstly, it does not handle certain kinds of attribute definitions in which the generated tree has the same production name as the defining production and the child trees are retrieved via accesses to synthesized attributes on child nodes. An example would be a synthesized attribute that stores the "base" versions of "extended" statements such as the `doWhile` and `ifThen` productions in Fig. 6. For "base" productions such as `while` and `ifThenElse` this attribute would be constructed with the same production. Unfortunately the rules we generate for such definitions are always non-terminating and therefore not useful. Finally, since the analysis assumes that the grammar is non-circular, it does not handle reference attribute grammars since the circularity problem is undecidable in that setting.

7 Related Work

In Knuth's original work on attribute grammars [8] he gave a circularity analysis. In the attribute grammar community it is not uncommon to extend this analysis when new features are added to the attribute grammar paradigm. In their specification of higher-order attribute grammars, Vogt *et al.* extended Knuth's circularity analysis to that setting. In the original work on *forwarding* the completeness analysis was combined with the circularity analysis to accommodate the need to use global attribute dependency information in the completeness analysis [13,1]. In the case of reference and remote attributes [6,2], it is not possible to have a precise circularity analysis, as the problem is undecidable [2].

In Vogt *et al.*'s original paper on HOAGs they define "well-definedness" to include termination of tree construction and state a lemma [17, Lemma 3.2, page 142] that imposes a condition that would ensure termination of tree construction. This condition requires that non-terminal attributes (the terminology used for higher-order attributes) do not appear more than once in a path in their formulation of trees. This is similar to our imposition of an ordering on non-terminals for higher-order inherited attributes. But our use of rewrite rules for local and synthesized higher order attributes is less restricting and allows for a more precise analysis. In other work [16] they drop this termination requirement for grammars to be considered "well-defined".

There is also a trove of related work in the area of program termination for functional and imperative programming and by necessity we cannot attempt to cover this area. One track of work began with the work by Lee, Jones and Ben-Amram[10] on the "size change" principle applied to first order functional programs. Here, the *size* of a parameter for each function call must always decrease over a domain with a fixed smallest value. This was later extended to higher-order functional programs [11]. This work was done for call-by-value languages. While some techniques in these works might also be useful in an analysis on higher order attribute grammar termination, they are not immediately applicable since they are designed for imperative or functional programming languages. We could, in principle, translate the attribute grammar specification to a functional program [7] (in fact, Silver formerly translated attribute grammar specifications to Haskell). But as described above, an analysis that uses the domain knowledge captured in the attribute grammar constructs is simpler and the use of such domain knowledge often leads to more precise analyses.

8 Future Work and Conclusion

There are several possible avenues for future work. First, we will look at relaxing the restrictions on the class RHOAG of higher-order attribute grammars handled by our termination analysis. We would like to handle grammars with attribute definitions that construct trees with the same production name as the attribute's production. We would also like to handle grammars that make use of Silver's `case` constructs. Finally we would like to relax the analysis' and its correctness proof's assumptions of grammar non-circularity in order to handle reference attribute grammars (RAGs).

The rules we generate are simple and do not make full use of the analytical power of tools such as AProVE. We expect that much simpler analyses could be used to show termination for the rules generated for most grammars. One possible approach would be to order productions (based on tree creating definitions) and then verify that the generated rules satisfy the corresponding recursive path ordering (RPO) [3]. This would provide a possible avenue for modularizing the analysis. We expect to be able to specify a modular condition (on extension specifications) that ensures the existence of the desired production ordering in the composed grammar. We also expect to be able to specify a modular condition on extensions that guarantees that the composed grammar's non-terminals can be ordered to limit the number of inherited accesses in tree creation sequences. Such modular analyses are important in our Silver extensible language framework, which aims at the development of modular and composable language extensions.

To conclude, in this paper we described an analysis that checks attribute grammars for termination of higher-order attribute evaluation. With the higher-order circularity and completeness tests, this analysis ensures that attribute evaluation terminates normally. We have implemented this analysis in Silver and have run this analysis on ableJ, an extensible specification of the Java 1.4 programming language. Future work will include enlarging the class of HOAGs handled by the termination analysis, and developing a modular version for use in our extensible language framework.

References

1. Backhouse, K.: A Functional Semantics of Attribute Grammars. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 142–157. Springer, Heidelberg (2002)
2. Boyland, J.T.: Remote attribute grammars. J. ACM 52(4), 627–687 (2005)
3. Dershowitz, N.: Orderings for term-rewriting systems. Theoretical Computer Science 17, 279–301 (1982)
4. Ekman, T., Hedin, G.: The JastAdd system - modular extensible compiler construction. Science of Computer Programming 69, 14–26 (2007)
5. Giesl, J., Thiemann, R., Schneider-kamp, P., Falke, S.: AProVE: A system for proving termination. In: Extended Abstracts of the 6th International Workshop on Termination, WST 2003, pp. 68–70 (2003)

6. Hedin, G.: Reference attribute grammars. Informatica 24(3), 301–317 (2000)
7. Johnsson, T.: Attribute Grammars as a Functional Programming Paradigm. In: Kahn, G. (ed.) FPCA 1987. LNCS, vol. 274, pp. 154–173. Springer, Heidelberg (1987)
8. Knuth, D.E.: Semantics of context-free languages. Mathematical Systems Theory 2(2), 127–145 (1968), corrections in 5, 95–96 (1971)
9. Krishnan, L.: Composable Semantics Using Higher-Order Attribute Grammars. Ph.D. thesis, University of Minnesota, Department of Computer Science and Engineering, Minneapolis, Minnesota, USA (to appear, 2012), draft available at http://melt.cs.umn.edu/pubs/krishnan2012PhD/
10. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: Proc. of the 28th ACM Symposium on Principles of Programming Languages, POPL 2001, pp. 81–92. ACM Press (2001)
11. Sereni, D., Jones, N.D.: Termination Analysis of Higher-Order Functional Programs. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 281–297. Springer, Heidelberg (2005)
12. Sloane, A.M.: Lightweight Language Processing in Kiama. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2009. LNCS, vol. 6491, pp. 408–425. Springer, Heidelberg (2011)
13. Van Wyk, E., de Moor, O., Backhouse, K., Kwiatkowski, P.: Forwarding in Attribute Grammars for Modular Language Design. In: Nigel Horspool, R. (ed.) CC 2002. LNCS, vol. 2304, pp. 128–142. Springer, Heidelberg (2002)
14. 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)
15. Van Wyk, E., Krishnan, L., Bodin, D., Schwerdfeger, A.: Attribute Grammar-Based Language Extensions for Java. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 575–599. Springer, Heidelberg (2007)
16. Vogt, H.: Higher order attribute grammars. Ph.D. thesis, Department of Computer Science, Utrecht University, The Netherlands (1989)
17. Vogt, H., Swierstra, S.D., Kuiper, M.F.: Higher-order attribute grammars. In: ACM Conf. on Prog. Lang. Design and Implementation, PLDI 1989, pp. 131–145 (1989)

Metamodelling for Grammarware Researchers

Richard F. Paige, Dimitrios S. Kolovos, and Fiona A.C. Polack

Department of Computer Science, University of York, UK
{richard.paige,dimitris.kolovos,fiona.polack}@york.ac.uk

Abstract. A *metamodel* is variously defined as a model of a model, a definition of a language, a description of abstract syntax, and a description of a domain. It is all of these things and more. Metamodels can be confusing, and explaining *why* they are constructed, *what* you can do with them, and *how* they are built can be challenging, especially when trying to bridge the gap between the modelware and grammarware communities. In this example-driven mini-tutorial, we introduce the key concepts and ideas behind metamodelling and explain why metamodels are useful, and particularly how they differ from grammar-based approaches to language development. We give some tips on how grammarware researchers can explain *what they do* to modelware researchers, and vice versa, in the spirit of interdisciplinarity and improving collaboration.

1 Introduction

We provide an introduction to metamodelling for grammarware researchers. We assume that such researchers are comfortable and experienced with defining and implementing grammars, using Extended Backus-Naur Form (EBNF), parser generator tools, and grammar-based manipulation of languages (e.g., for compilation, analysis, extraction and comparison). We also assume that these researchers have little to no experience with metamodelling, and may be confused as to why metamodelling is even done. However, such researchers may be interested in finding ways to explain their research to modellers, and may seek to better understand modelling research.

The entry barrier to metamodelling can be quite high, not least because of the cumbersome 'metamodel-speak' terminology (which can be particularly challenging for non-native English speakers), and the lack of standard definitions. In this tutorial, we aim to lower the entry barrier.

We start the tutorial, in Section 2, with some basic definitions, illustrated with very small examples of metamodels, constructed in different languages. In Section 3 we discuss reasons for constructing metamodels, as well as the key differences and similarities between metamodelling and grammar-based approaches. We also explain a typical metamodelling process, which will help to clarify some of the key differences between metamodelling and grammar-based approaches to language design, and briefly note the important standards for metamodelling. Finally, in Section 4, we present two examples of metamodels and illustrate how they may be used.

2 What Is a Metamodel?

Much paper has been expended on definitions of the term *metamodel*. Some of the key literature in the area includes Bézivin's papers on software modernisation [2] and the classic *On the Unification Power of Models* [3]. A further influential reference is Atkinson and Kühne's *Model-Driven Development: a Metamodelling Foundation* [1]. The Object Management Group (OMG) has published numerous standards, including its MDA Foundation Model [8] which provides OMG definitions of metamodelling. As of yet, there is no consensus among experts on a precise definition of metamodel, and differences in vocabulary and terminology across the definitions serves to promote confusion.

Since our purpose is to lower the entry barrier to metamodelling, we adopt a simple definition, not incompatible with those used by many other researchers, but expressed in a hopefully clear and unambiguous way.

Definition. A *metamodel* is a description of the abstract syntax of a language, capturing its concepts and logic, using modelling infrastructure.

A language (for software or systems engineering) has an abstract syntax, a concrete syntax, and a semantics. A language description is a formal expression that is amenable to processing by software. The concepts captured in a metamodel are the important terms that the language is defined to express. The logic defines how concepts can be combined to produce meaningful expressions in the language. The modelling infrastructure is an important element of the definition; without it, there would be no difference between a metamodel and a grammar. We will see some examples of modelling infrastructure shortly; this infrastructure supports the *unification principle* of metamodelling: that is, models and metamodels (and indeed operations upon each) are treated uniformly. To put it simply, metamodels *are also models*.

Any machine processable language can be given a metamodel, including textual and visual languages. Let's look at two small examples.

2.1 Example 1: A Metamodel for Eiffel (a Textual Language)

Consider, firstly, Listing 1.1, which is a part of an object-oriented program written in the Eiffel language [13].

Listing 1.1. Fragment of an Eiffel program

```
class CITIZEN

 inherit PERSON

 feature {ANY}
   spouse: CITIZEN
   children, parents: SET[CITIZEN]
   single: BOOLEAN is do
```

```
      Result := (spouse=Void)
    end

  feature {BIG_GOVERNMENT}
    divorce is
      require not single
      do .. end
      ensure single and (old spouse).single

  invariant
    single or spouse.spouse=Current;
    parents.count <= 2

end -- CITIZEN
```

An EBNF grammar for Eiffel can be found in Eiffel reference books, and the Eiffel compiler implements various parsing algorithms for Eiffel. However, if we identify a suitable metamodelling infrastructure, we can also define a metamodel for Eiffel, describing the important concepts and logic of Eiffel.

For now, it is sufficient to assume that "metamodelling infrastructure" is some technology that allows *entities* (holding data) to be defined, related and instantiated. Thus, for example, an object-oriented programming language might suffice (this is not completely correct, but it will suffice for now; we clarify this shortly). The metamodel uses the semantics of the metamodelling infrastructure, a form of referential semantics of the concepts and logic of languages defined by the metamodel. For example, the metamodel for Eiffel can be described in Java. Listing 1.2 gives part of the metamodel, focusing on the important concepts in Eiffel: classes and features (operations and attributes), invariants and parents.

Listing 1.2. Part of an Eiffel metamodel specified in Java

```
  class EIFFEL_CLASS {
    public ArrayList<EIFFEL_FEATURE> features;
    public ArrayList<EIFFEL_INVARIANT> invariants;
    public ArrayList<EIFFEL_CLASS> parents;
  }
```

What does this Java class describe? Firstly, it describes an entity (called EIFFEL_CLASS) that consists of several public fields. Each field is an ArrayList. The first field describes the *features* of EIFFEL_CLASS (the Eiffel attributes and operations), the second the *invariants*, and the third the parent classes. The generic parameters of the ArrayLists are important, and are key parts of the metamodel: they represent descriptions of other important entities in the metamodel, viz., EIFFEL_FEATURE and EIFFEL_INVARIANTs. Note that EIFFEL_INVARIANT encodes the concepts *and* logic of Eiffel invariants, particularly their abstract syntax; an evaluation function would be needed to evaluate the invariant on an object state.

The Java program in Listing 1.2 is a description of the abstract syntax of parts of the Eiffel language. Describing the abstract syntax is the first step in a metamodelling process that leads to the development of a rich modelling language and toolset (e.g., with semantics, model editing support and interoperability with other languages). In practice, in developing modelware, describing abstract syntax is often the *only* step that's taken. We present the metamodelling process later.

Why might we want to create a metamodel for Eiffel? Widely accepted grammars/EBNF, with parsers, exist. Reasons for creating a metamodel include desiring to use model management technology and tools (e.g., model refactoring tools, model merging tools), or to enable interoperation between existing Eiffel grammar-based tools with model management tools. Other motivations are discussed later.

2.2 Example 2: Metamodel for ER Diagrams (a Visual Language)

Now let's see a second example, a metamodel for a visual language. Often, when working with databases, we construct data models. A very simple data model consists of a number of entities, containing attributes, as well as references to other entities. We might express data models using some form of entity-relationship (ER) diagrams. A metamodel for a (simplified) ER diagram language can be described as shown in Fig. 1. Here, instead of using Java as the modelling infrastrcture to express the metamodel, we use UML [10].

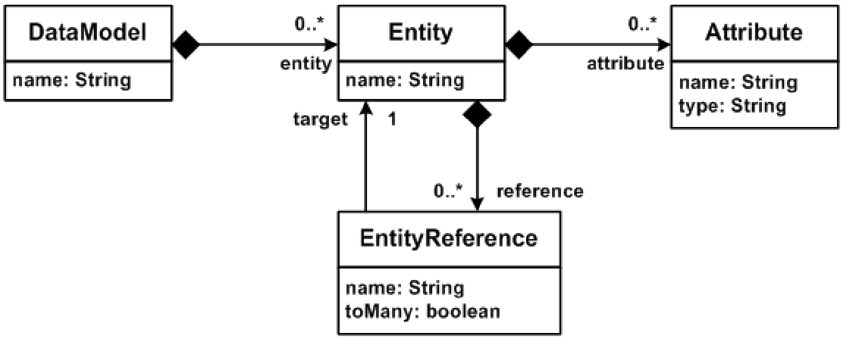


Fig. 1. A data metamodel in UML

This UML class diagram describes concepts *DataModel*, *Entity*, *Attribute* and *EntityReference*. Each concept has a number of attributes (particularly ones describing *name*s of concepts). The concepts are also related. In particular, *DataModel* is composed of zero or more *Entity* concepts, and an *Entity* is, in turn, composed of a number of *Attribute* concepts. The *EntityReference* concept is a little different, in that it *refers* to exactly one *Entity* concept (the arrow without a diamond, pointing at *Entity*).

Once again, this metamodel captures the abstract syntax of an ER diagram language; it says nothing about the concrete syntax or the semantics that the end-users of the ER language might apply. By decoupling the definition of abstract and concrete syntax, we allow ourselves flexibility in designing and deploying different concrete syntaxes that conform to the abstract syntax, but also go only part of the way towards producing something that is helpful to end-users.

There is an important point to note about the example metamodel shown in Fig. 1: it allows models to be instantiated that we may not want. Consider the *Entity* concept; it has a *name* field. When we construct an ER diagram, we populate it with a number of entities (each of which are instances of the *Entity* concept). Each of these entities must have a name. However, there is *nothing* in the metamodel shown in Fig. 1 that requires entities to have unique names, nothing to prevent us from using the same name for each and every entity. We probably want to enforce unique entity naming in the metamodel and language definition. To do this, traditionally we need to augment the description of abstract syntax with *well-formedness rules*, i.e., constraints on the metamodel that prevent ill-formed models from being constructed. If we were using Java to express a metamodel, we could write a simple method that traversed the ArrayLists to ensure that all entities had different names. When using UML concrete syntax to express a metamodel, we typically use the OMG standard Object Constraint Language (OCL) for this purpose. An example of an OCL constraint for this metamodel is shown in Listing 1.3. It states that in the context of any *DataModel*, names of entities must be unique (there are a number of different ways to express this).

Listing 1.3. OCL well-formedness rule

```
context DataModel inv:
   self.entity->collect(name).size() = self.entity.size()
```

2.3 Mathematical Definitions

A lightweight yet mathematical definition of metamodel was given in [4]; we restate it (lightly changed) here. First, we define a notion of a *model*.

Definition: a *model* M is a triple (G, Ω, μ) where G is a directed multigraph, Ω is a *reference model* associated with a (potentially different) directed multigraph G_Ω, and μ is a *mapping function* that associates elements of G with nodes of G_Ω.

The mapping between model and reference model is called *conformance*; this is the fundamental relationship involved in defining metamodels. Models are said to *conform* to one or more metamodels. The issue of *when* models should conform to a metamodel is one for debate, and typically depends on the tools that are being used to construct the models. Generally, a model must conform to a metamodel for a tool to be able to load and process the model.

To understand what follows, it is helpful to remind onesself about the unification principle mentioned earlier: a metamodel is also a model.

The definition given is very broad, and allows arbitrary notions of conformance to be defined. In practice in MDE, typically only three definitions of conformance are used.

- *Terminal Model:* model M is called a *terminal model* and its reference model Ω is a metamodel. A classic example of this is where M is a UML class diagram and Ω is the UML metamodel.
- *Metamodel:* in this situation, M is itself a metamodel (e.g., the UML metamodel from the OMG) and its reference model Ω is called a *metametamodel.* A classic example of this situation is where M is the UML metamodel and Ω is MOF, the OMG standard through which metamodels are defined. Ecore (which is part of Eclipse EMF) is an implementation of a simplified form of MOF.
- *Metametamodel:* M is a metametamodel (e.g., MOF or Ecore). How are these defined, i.e., what is its Ω? Metametamodels are self-defining: that is, they are used to implement themselves. As such Ω is M. MOF and Ecore are both self-defining. Interestingly, Ecore in comparison with the UML metamodel is relatively straightforward (in terms of number of concepts and their relationships)[1].

2.4 Summary

The example metamodels presented in this section, in Java and in UML's class diagram syntax, illustrate how to define the abstract syntax of languages using mechanisms other than EBNF. We return to discussion of the characteristics of metamodelling infrastructures, as well as the overall metamodelling process, shortly. Before that, we motivate metamodelling, and address the question *why construct metamodels in the first place?* In doing so, we also aim to suggest the key differences between metamodelling and grammarware.

3 Why Metamodel?

Metamodels are constructed for a number of reasons. A common one is to precisely describe a language so that tools can be implemented to support that language (e.g., editors); this of course is also a reason why EBNFs and context-free grammars are created. Indeed, both grammarware and modelware share a number of common use cases; some typical grammarware use cases were presented in [14]. We create metamodels and grammars in order to:

- present and process large amounts of documentation in a structured way;
- generate valid, well-formed text from a variety of input sources (models, metamodels, programs);

[1] This is not the case for MOF, which has a complex relationship with the UML metamodel, and won't be discussed further here.

– enable traceability use cases, where we want to link machine-processable artefacts (models, programs, grammars, metamodels) to each other and to external artefacts, such as documentation, web pages, and requirements;
– document and support language evolution over time;
– precisely define languages in a way that allows us to use (software) tools to check sentences written in the languages;
– compare languages, in terms of their features, their structures, and their patterns.

What can you do with a metamodel once you have constructed one? There are a number of possibilities, including the following.

– You can create well-formed models that *conform* to the concepts and logic expressed in the metamodel. This is analogous to creating well-formed sentences that conform to an EBNF in grammarware.
– You can transform models that conform to your metamodel into other forms, e.g., into models that conform to a different metamodel. This is called *model transformation*. A classic example of this is transforming a UML class model into a relational database model. This is roughly analogous to program transformation, but not quite – for reasons we'll get to shortly.
– You can check that models satisfy desirable (or mandatory) properties. We have seen one approach to this earlier, when we wrote an OCL constraint (Listing 1.3). As another example, consider the datamodel metamodel shown earlier. It allows all kinds of models to be created, except those where two different entities have the same name. We may want to disallow further models, for example, ones where names of *Entity* concepts and $EntityReference$ concepts are the same. We could express this property as a set of OCL rules applied to the metamodel, or we could transform our model into another form that allows such properties to be easily checked (e.g., a theorem prover). This is roughly analogous to writing a type (or property) checker for an abstract syntax tree, e.g., using a tree walker; however, in metamodelling the property checking generally abstracts away from internal algorithms needed to traverse trees or data structures.
– You can generate text from models that conform to your metamodel. For example, you can produce documentation, or code, or web pages. This is generally distinguished from *model transformation* as it produces a non-model artefact (which is also why we said the analogy between model and program transformation is inexact).
– You can compare models that conform to the same metamodel, or to different metamodels. Comparison could be done as a precursor to version control on models, or to highlight differences between models, or as part of a testing process. *Model comparison* is roughly equivalent to the process of program comparison.

3.1 (Meta)modelware versus Grammarware

It appears that what you can do with metamodels you can do with grammars (and vice versa). At the very least, there are many similarities between the two approaches. So what are the key differences between metamodelware and grammarware? Again, this is a matter of debate, but there are a number of important differences worth mentioning.

- **Trees and Graphs:** Metamodelling, and metamodelling infrastructure, is designed to be applicable to both tree-based languages and graph-based languages; that is, languages where the abstract syntax is a tree, or a graph. By contrast, grammarware is best applied to tree-based languages (which doesn't mean that you can't apply grammar technology to graph-based languages, only that it may be more awkward to do so).
- **Abstract vs. Concrete Syntax First:** When defining a new language, the modelware engineer starts with the definition of the abstract syntax of the language, while the grammarware engineer starts with the definition of (one of) the concrete syntaxes of the language. The modelware community generally takes a modelling approach to language definition, starting with a conceptual model of a language (i.e., abstract syntax). By comparison, the grammarware community generally (though not exclusively) starts with a *design* (concrete syntax).
- **Metamodel-Concrete Syntax vs. Grammar-AST:** For textual languages, modelware and grammarware approaches converge as after defining the abstract syntax of the language (metamodel), the modelware engineer will also need to define the textual concrete syntax of the language. On the other hand, after defining the grammar of the language, the grammarware engineer will need to define the DOM/AST of the language so that any further manipulation happens on a semantically rich structure rather than on a homogeneous concrete syntax tree.
- **Standards:** In the modelware space, there is a widely accepted de facto standard for defining the abstract syntax of languages (EMF/Ecore). In the grammarware world, there is a variety of EBNF-derived grammar definition languages which are not generally consistent with each other, though there have been efforts on standardising EBNF [12], which do not seem to have had the same impact as metamodelling standards.
- **EMF Standard Tools:** The dominance of EMF as a de facto standard has lead to the development of a large number of model/DOM management languages and tools which can work with models conforming to any language defined using MOF/Ecore, regardless of its concrete syntax(es). Such tools include model-to-model and model-to-text transformation languages, model validation and refactoring engines, model comparison and merging tools etc. In our view, the absence of such a dominating framework for constructing consistent DOM/AST implementations in the grammarware hampers the development of such language-agnostic tools.
- **Concrete Syntax Tools:** Once the abstract syntax of a language is defined in the modelware world, there is a selection of mature tools that the engineer can use to develop textual (e.g., Xtext, EMFText), graphical (e.g., GMF, Graphiti), or hybrid concrete syntaxes for the language. To our knowledge there are no widely accepted toolkits for developing alternative graphical syntaxes for textual languages in the GW world.

– **Metamodels as Models:** Metamodels and models are *unified* by their metamodelling infrastructure, and can generally be treated interchangeably. To put it another way, by construction, a metamodel can be treated like just any other kind of model.

This last point is the key one: in Model-Driven Engineering, everything is a model, including metamodels, the infrastructure upon which metamodels are defined (we discuss this later), transformations, comparisons, properties, etc. This unified treatment of the universe of discourse has both advantages and disadvantages. For one, it is conceptually elegant: everything is a model, you just need to understand *what type of model it is* in order to process it effectively. Second, it allows remarkably generic and flexible programs to be written that process models. Third, it removes many of the semantic gaps that arise between software engineering artefacts. However, there is a price to be paid, including the inevitable inefficiencies that arise when processing models that have deep structure, as well as the hidden complexity of models constructed using metamodelling infrastructure.

3.2 Metamodelling Process

Metamodelling generally follows a well-defined process that has been developed over a long time, in particular as a result of the development of UML, as well as significant experience in building domain-specific languages. The objective of the metamodelling process is to develop a specification (ideally, but not necessarily, with supporting tools) for a language. The process is generally iterative, and is also non-trivial: a complex modelling domain (like object-oriented software) will invariably lead to a complex modelling language, and hence a complicated metamodel or set of metamodels. Nevertheless, the process that is followed for developing a complex metamodel is generally no different than that for a less complex one. The basic steps, derived from [5], are typically as follows.

1. Select a metamodelling infrastructure (see Section 3.3).
2. Define an abstract syntax using said infrastructure.
3. Define well-formedness rules and any operations on the metamodel.
4. Define one or more concrete syntaxes that conform to the abstract syntax.
5. Define semantics.
6. Define mappings to other languages, e.g., by using transformations.

In language developments, the metamodelling process may end at different steps. A proof-of-concept or prototype might stop after the second or third step. For deployment in production, the first five steps might be carried out (for example, the semantics of the language might need to be defined in order to support execution or simulation). To support working with legacy/brownfield[2] development, as well as integration with other software development tools, all six steps may need to be carried out.

[2] Greenfield development of software starts from scratch, with no dependence on previous software/requirements; brownfield development involves consideration and interoperation with existing systems - that is, the usual case in software engineering!

3.3 Metamodelling Standards and Infrastructure

The first step in the typical metamodelling process is to select a metamodelling infrastructure. There are a number of standard infrastructures available today.

- The OMG's metamodelling stack is based on the Meta-Object Facility (MOF) [9]. MOF is reflective (it defines itself), and is used to define metamodels such as the UML, as well as many others. The OMG stack is generally called a *four-level* stack: at the top of the stack is MOF, which defines metamodels that sit at the next level of the stack. Below metamodels are models, which are instances of metamodels. Finally, models themselves can be instantiated, and these instances are at the bottom of the stack.

  MOF is sufficient to define the abstract syntax of languages, but does not provide native support for, for instance, well-formedness rules. MOF actually consists of two versions (including *Essential MOF* and *Complete MOF*) and are generally used in concert with OCL to define well-formedness rules. UML's metamodel is defined in terms of MOF and OCL. As of yet, the OMG does not have a widely accepted standard for defining concrete syntax or semantics, though some metamodellers use HUTN [7] as a concrete syntax, and some use UML directly to define the semantics of languages. Work is underway at the OMG on a Diagram Definition standard to support some concrete syntax [11]. An implementation of MOF is available via NetBeans, as its Metadata Repository (MDR).
- The Eclipse Modelling Framework (EMF) supports the Ecore standard, which is a simplified implementation of MOF. Ecore is the de facto standard for metamodelling currently, and is used in the Eclipse implementation of UML, along with many other general-purpose and domain-specific language tools.
- MetaDepth is a so-called *deep metamodelling* infrastructure [6]. It avoids a well-known issue with the OMG metamodelling approach, wherein some concepts (particularly objects) appear in multiple levels of the stack (objects, for example, are concepts appearing both in metamodels and models). Other approaches to avoiding this problem include the Golden Braid architecture [5].

4 Examples

In this section we briefly present two more examples of metamodels. For the first example, we present an abstract syntax and a concrete syntax, defined and implemented using Eclipse's EMF and GMF. For the second example, we present both abstract and concrete syntaxes, and brief details of a *model transformation* that uses the metamodel to support a real scenario.

4.1 Conference Language

This example presents a domain-specific language for defining schedules for a conference. The idea is to provide a customised editor for domain-experts (conference managers) who know about important concepts like participants, tracks (presentations on a particular theme) and slots where talks can be scheduled in a track. These domain-experts are not knowledgeable about metamodelling. The goal is to build a simple editor that supports creation of *conference models* that take into account important conference timetabling concepts.

Having decided to use the EMF modelling infrastructure, the next step in the process of Section 3.2 is to define an abstract syntax. This requires us to think about the key concepts and structures of a conference timetable. What are these? There are *tracks*, consisting of a number of *slots* into which *talks* can be scheduled. Talks have *participants* (we may need to be sure that we avoid clashes, as a participant may need to give several talks). There is also the critical conference session – lunch.

Based on this, we can define an abstract syntax metamodel. In the process of defining this metamodel, we identify a number of recurring concepts: some concepts have names, and some concepts include timing information. These recurring concepts are extracted and abstracted, using inheritance (which MOF/Ecore supports).

We implemented our metamodel using the Ecore metamodelling language of EMF. A graphical view of the metamodel is shown in Fig. 2. The types used for the fields (*EInt* and *EString*) are built-in Ecore types.

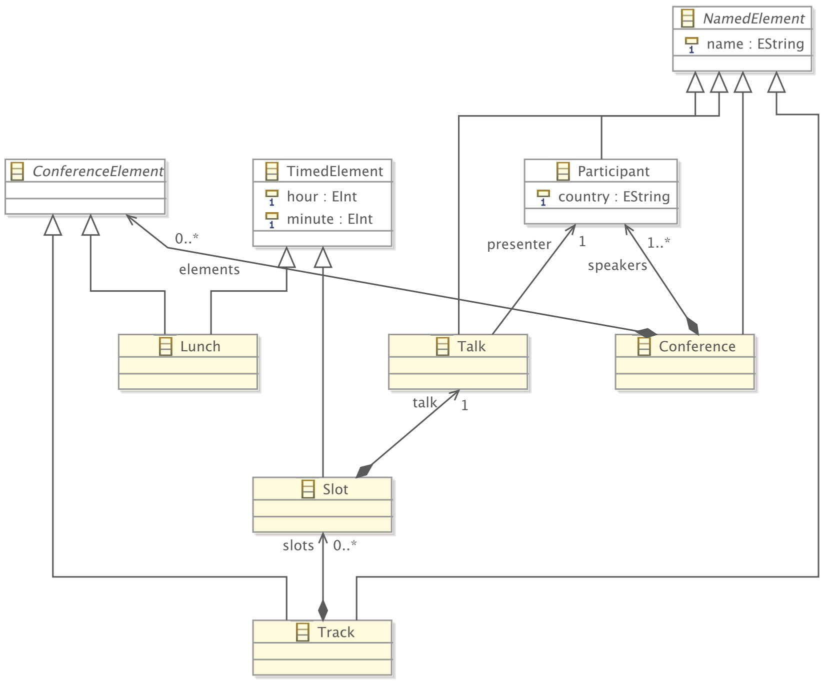


Fig. 2. Metamodel, in Ecore's EMF notation, defining the abstract syntax for Conference timetabling

The next step would be to define any well-formedness rules on the abstract syntax; this could be done using OCL. For example, we may want to state that each speaker at the conference is a presenter of a talk scheduled in a slot (in effect, this ensures that we haven't missed anyone, i.e., that all talks have a registered speaker who is scheduled into a slot). This could be expressed in OCL as follows.

Listing 1.4. OCL well-formedness rule for conference timetable

```
context Conference inv:
  self.speakers->includes(
    self.elements->select(t|Track).slots.talk.presenter)
```

The next step of the process of Section 3.2 is to define a concrete syntax, based on the abstract syntax. We typically do this in collaboration with the end-users/domain-experts. As we have chosen EMF as our metamodelling infrastructure, it is sensible to use Eclipse's Graphical Modelling Framework (GMF) as a mechanism to define our concrete syntax. We also choose to build a graphical syntax, as conference timetablers may be more comfortable with this. An example graphical concrete syntax, implemented using Eclipse's GMF, is shown in Fig. 3.

This graphical syntax conforms to the abstract syntax of Fig. 2. Indeed, the *Lunch* slot (annotated with a food icon) is an instance of the *Lunch* metaclass; all other concepts and relationships are instances of abstract syntax concepts. We have chosen to represent *Track*s as rounded rectangles (with calendar annotations), and *Slot*s as dashed rectangles (with clock annotations), for example.

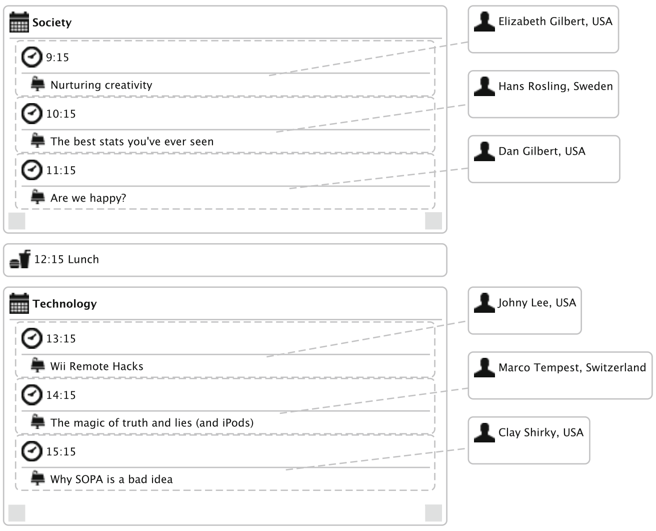


Fig. 3. Conference timetabling concrete syntax

GMF provides the support needed to specify and implement this concrete syntax – with substantial effort. In fact, for this editor, we made use of the EuGENia[3] toolset to automatically generate the concrete syntax editor from an annotated version of the abstract syntax of the language that appears in Listing 1.5 (this time using a textual syntax, as opposed to the graphical syntax of Ecore demonstrated earlier in Figure 2).

Listing 1.5. Definition of a graphical syntax for the Conference DSL

```
@namespace(uri="conference", prefix="conference")
package conference;

@gmf.node(label="name")
abstract class NamedElement {
  attr String[1] name;
}

@gmf.diagram
class Conference extends NamedElement {
  val ConferenceElement[*] elements;
  val Participant[+] speakers;
}

abstract class ConferenceElement {
}

class Track extends ConferenceElement, NamedElement {

  @gmf.compartment(layout="list")
  val Slot [*]  slots ;
}

@gmf.node(border.style="dash")
class Slot  extends TimedElement {

  @gmf.compartment(layout="list", collapsible="false")
  val Talk[1]  talk;
}

class Talk extends NamedElement {

  @gmf.link(style="dash")
  ref Participant [1]  presenter;
}

@gmf.node(label="name,country", label.pattern="{0}, {1}")
class Participant  extends NamedElement {
  attr String[1] country;
```

[3] http://www.eclipse.org/epsilon/doc/eugenia/

```
}

@gmf.node(label="hour,minute", label.pattern="{0}:{1} Lunch")
class Lunch extends ConferenceElement, TimedElement {
}

@gmf.node(label="hour,minute", label.pattern="{0}:{1}")
class TimedElement {
  attr int[1] hour;
  attr int[1] minute;
}
```

To demonstrate the orthogonality between the abstract and concrete syntaxes, we have also developed a textual syntax for the same language using the EMF-Text toolkit[4]. Listing 1.6 demonstrates the model of Figure 3 represented using this textual syntax.

Listing 1.6. The model of Figure 3 expressed using a textual syntax

```
CONFERENCE "TED"

TRACK "Society" :
    AT 09:15 : TALK "Nurturing creativity" PRESENTED BY "Elizabeth Gilbert"
    AT 10:15 : TALK "The best stats you've ever seen"
        PRESENTED BY "Hans Rosling"
    AT 11:15 : TALK "Are we happy?" PRESENTED BY "Dan Gilbert"

AT 12:15 LUNCH

TRACK "Technology" :
    AT 13:15 : TALK "Wii Remote Hacks" PRESENTED BY "Johny Lee"
    AT 14:15 : TALK "The magic of truth and lies (and iPods)"
        PRESENTED BY "Marco Tempest"
    AT 15:15 : TALK "Why SOPA is a bad idea" PRESENTED BY "Clay Shirky"

REGISTERED SPEAKERS :
    "Elizabeth Gilbert" FROM USA,
    "Hans Rosling" FROM Sweden,
    "Dan Gilbert" FROM USA,
    "Johny Lee" FROM USA,
    "Marco Tempest" FROM Switzerland,
    "Clay Shirky" FROM USA
```

To define the concrete syntax, we needed to write an extended grammar that refers to the abstract syntax of the language, and from this grammar EMF generated a parser that can parse text that conforms to the grammar to in-memory models that conform to the abstract syntax of the Conference language, and vice-versa, also, it generates IDE tooling such as a sophisticated editor supporting code completion, syntax highlighting etc. The extended grammar

[4] http://www.emftext.org

appears in Listing 1.7[5]. It is instructive to reflect between this listing and the metamodel presented in Fig. 3 (and the corresponding concrete syntax definition, based on the metamodel, in Listing 1.5).

Listing 1.7. The EMFText grammar defining a textual syntax for the Conference DSL

```
SYNTAXDEF conference
FOR <conference>

START Conference

OPTIONS {
    licenceHeader =”licence.txt”;
    reloadGeneratorModel = ”true”;
    generateCodeFromGeneratorModel = ”true”;
    tokenspace = ”1”;
    disableLaunchSupport = ”true”;
    disableDebugSupport = ”true”;
}

TOKENSTYLES {
    ”QUOTED_34_34” COLOR #404040;
}

RULES {
    Conference ::=
        ”CONFERENCE” #1 name[’”’,’”’]
        !0 ( !0 elements )*
        !0 ”REGISTERED” ”SPEAKERS” ”:” !0 speakers (”,” !0 speakers)*;

    Participant ::= name [’”’,’”’] #1 ”FROM” #1 country [];

    Talk ::= ”TALK” #1 name[’”’,’”’] #1 ”PRESENTED” ”BY” presenter[’”’,’”’] !0;

    Track ::= ”TRACK” #1 name[’”’,’”’] ”:” !0 (slots)*;

    Slot ::= ”AT” #1 hour[] #0 ”:” #0 minute[] ”:” #1 talk;

    Lunch ::= ”AT” hour[] #0 ”:” #0 minute[] #1 ”LUNCH” !0;
}
```

Here it is worth stressing that due to the architecture of EMF, programs that work with Conference models (e.g., to validate them, transform them etc) are agnostic of the actual concrete syntax in which the models are concretely described.

[5] Although we have intentionally kept the Conference language – and as a result, its textual concrete syntax – simple, it should be mentioned that EMFText has been used to implement the complete textual syntax of Java 5.

4.2 Proposal Language

Our second example illustrates a task that many academics have encountered – writing grant proposals. In particular, in European project proposals, there are numerous *tables* that have to be produced, summarising the *deliverables* of the project, the research *milestones*, the *work packages* that break up the project into parts, the *tasks* associated with these work packages, and the *partners* that carry out these tasks. The information is repeated multiple times in different ways: summary tables (e.g., capturing all work packages and the effort associated with the project), task tables for each work package (summarising the tasks and effort associated with each task and work package), Gantt chart, etc. It is easy for information to become inconsistent: invariably a significant part of debugging a project proposal is ensuring that the tables and Gantt chart are consistent.

To help manage project proposals more effectively, we develop a metamodelling toolset to support construction of *project models* that capture the key details. A single model is used to describe project work packages, tasks, partners, etc., and is used thereafter to automatically generate the content needed in the project proposal. The content can be inserted into the proposal directly.

We work through this metamodelling example using parts of the process from Section 3.2. We first present an abstract syntax metamodel, capturing the key concepts of the domain. This is presented in Fig. 4.

We would next create well-formedness constraints (e.g., that work packages have distinct names). These are generally straightforward. An important constraint might be that the start date of each task is before the end date, and that each partner leads at least one work package.

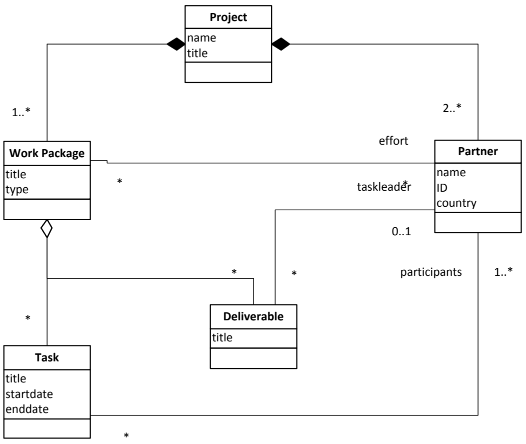


Fig. 4. Project content metamodel

Listing 1.8. OCL well-formedness rules for proposal system

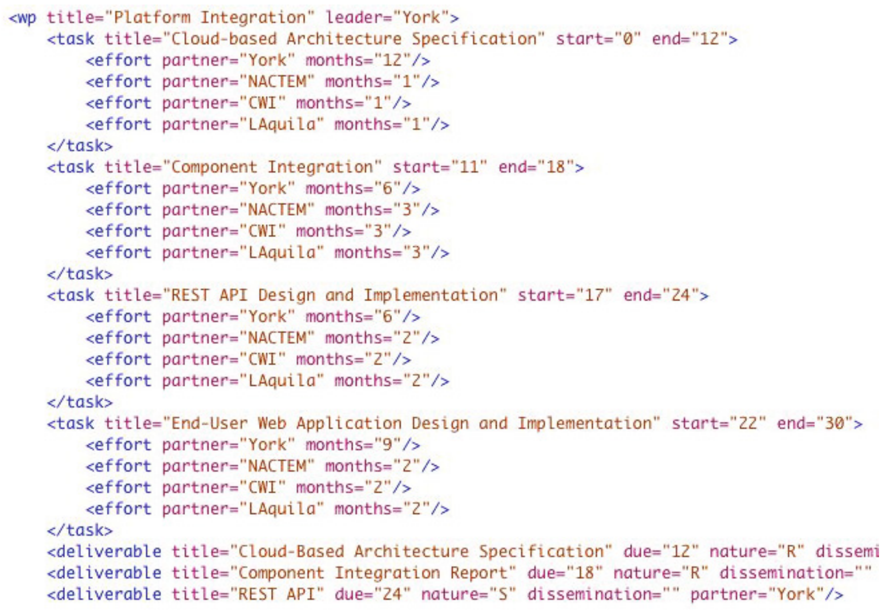
```
context Task inv:
  self.startdate < self.enddate

context Project inv:
  self.workpackage.leader.size() = self.partner.size()
```

Now we consider concrete syntax. We choose to use XML as the concrete syntax of this language. XML is widely used and understood, numerous smart editors exist to create and validate it, and all our partners were already familiar with it. Each concept in the abstract syntax is mapped directly to an XML concept. To illustrate this, Fig. 5 shows a partial model of a European project, including details of tasks and partners.

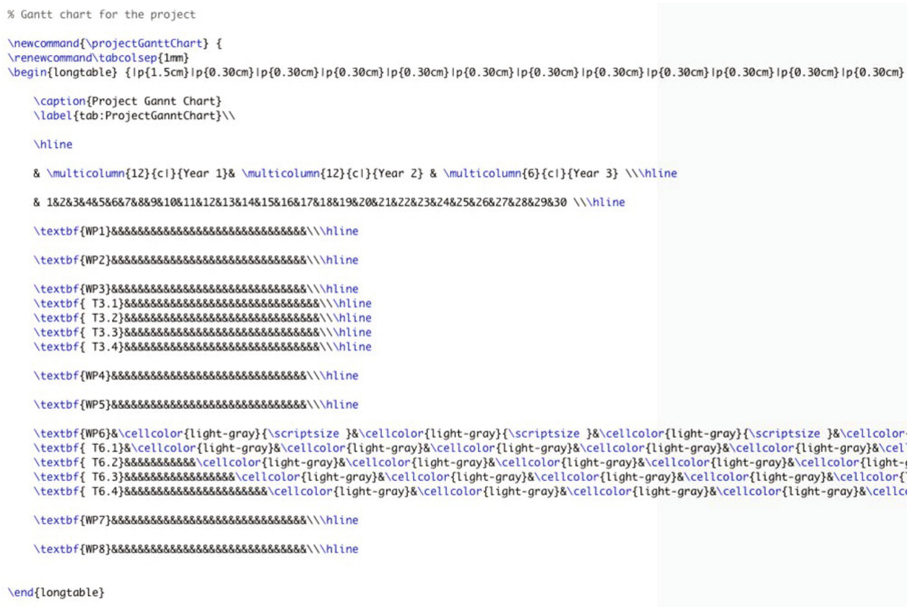
From models that are expressed in this modelling language, we can use Model-Driven Engineering (MDE) techniques and tools to automatically generate the information that we need for our project proposal. For example, we have implemented MDE transformations that automatically generate Gantt charts (in LaTeX format) that show the main deliverables from the work packages of the project, shown in Fig. 6. The Gantt chart that is produced by the typesetting macros is shown in Fig. 7.

From the same project model, we can also automatically generate work package effort tables, as well as summary tables of deliverables and work packages. By construction, these are consistent.



```
<wp title="Platform Integration" leader="York">
    <task title="Cloud-based Architecture Specification" start="0" end="12">
        <effort partner="York" months="12"/>
        <effort partner="NACTEM" months="1"/>
        <effort partner="CWI" months="1"/>
        <effort partner="LAquila" months="1"/>
    </task>
    <task title="Component Integration" start="11" end="18">
        <effort partner="York" months="6"/>
        <effort partner="NACTEM" months="3"/>
        <effort partner="CWI" months="3"/>
        <effort partner="LAquila" months="3"/>
    </task>
    <task title="REST API Design and Implementation" start="17" end="24">
        <effort partner="York" months="6"/>
        <effort partner="NACTEM" months="2"/>
        <effort partner="CWI" months="2"/>
        <effort partner="LAquila" months="2"/>
    </task>
    <task title="End-User Web Application Design and Implementation" start="22" end="30">
        <effort partner="York" months="9"/>
        <effort partner="NACTEM" months="2"/>
        <effort partner="CWI" months="2"/>
        <effort partner="LAquila" months="2"/>
    </task>
    <deliverable title="Cloud-Based Architecture Specification" due="12" nature="R" dissemi
    <deliverable title="Component Integration Report" due="18" nature="R" dissemination=""
    <deliverable title="REST API" due="24" nature="S" dissemination="" partner="York"/>
```

Fig. 5. Project instance in XML



```
% Gantt chart for the project

\newcommand{\projectGanttChart} {
\renewcommand\tabcolsep{1mm}
\begin{longtable} {|p{1.5cm}|p{0.30cm}|p{0.30cm}|p{0.30cm}|p{0.30cm}|p{0.30cm}|p{0.30cm}|p{0.30cm}|p{0.30cm}|p{0.30cm}|p{0.30cm}|p{0.30cm}

    \caption{Project Gannt Chart}
    \label{tab:ProjectGanntChart}\\

    \hline

    & \multicolumn{12}{c|}{Year 1}& \multicolumn{12}{c|}{Year 2} & \multicolumn{6}{c|}{Year 3} \\\hline

    & 1&2&3&4&5&6&7&8&9&10&11&12&13&14&15&16&17&18&19&20&21&22&23&24&25&26&27&28&29&30 \\\hline

    \textbf{WP1}&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&\\\hline

    \textbf{WP2}&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&\\\hline

    \textbf{WP3}&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&\\\hline
    \textbf{ T3.1}&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&\\\hline
    \textbf{ T3.2}&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&\\\hline
    \textbf{ T3.3}&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&\\\hline
    \textbf{ T3.4}&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&\\\hline

    \textbf{WP4}&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&\\\hline

    \textbf{WP5}&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&\\\hline

    \textbf{WP6}&\cellcolor{light-gray}{\scriptsize }&\cellcolor{light-gray}{\scriptsize }&\cellcolor{light-gray}{\scriptsize }&\cellcolor
    \textbf{ T6.1}&\cellcolor{light-gray}&\cellcolor{light-gray}&\cellcolor{light-gray}&\cellcolor{light-gray}&\cellcolor{light-gray}&\cel
    \textbf{ T6.2}&&&&&&&&&&\cellcolor{light-gray}&\cellcolor{light-gray}&\cellcolor{light-gray}&\cellcolor{light-gray}&\cellcolor{light-
    \textbf{ T6.3}&&&&&&&&&&&&&&&&\cellcolor{light-gray}&\cellcolor{light-gray}&\cellcolor{light-gray}&\cellcolor{light-gray}&\cellcolor{
    \textbf{ T6.4}&&&&&&&&&&&&&&&&&&&&&\cellcolor{light-gray}&\cellcolor{light-gray}&\cellcolor{light-gray}&\cellcolor{light-gray}&\cellc

    \textbf{WP7}&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&\\\hline

    \textbf{WP8}&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&\\\hline


\end{longtable}
```

Fig. 6. LaTeX macros generated automatically from project model

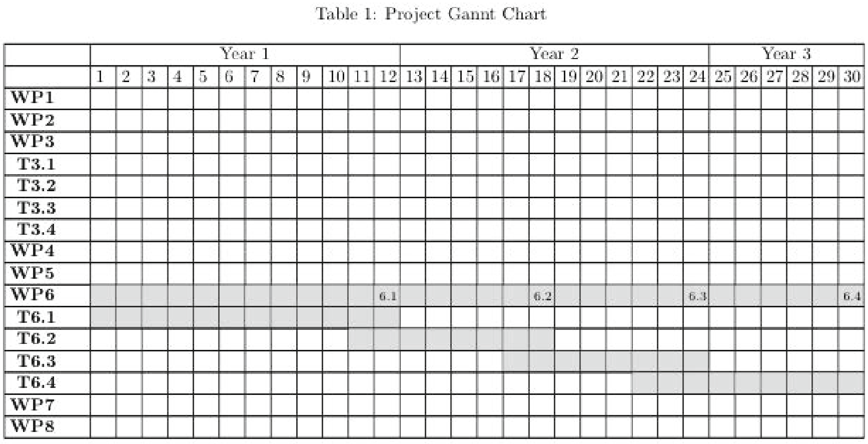


Table 1: Project Gannt Chart

	Year 1												Year 2												Year 3					
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
WP1																														
WP2																														
WP3																														
T3.1																														
T3.2																														
T3.3																														
T3.4																														
WP4																														
WP5																														
WP6	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨ 6.1	▨	▨	▨	▨	▨	▨ 6.2	▨	▨	▨	▨	▨	▨ 6.3	▨	▨	▨	▨	▨	▨ 6.4
T6.1	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨																		
T6.2											▨	▨	▨	▨	▨	▨	▨	▨												
T6.3																	▨	▨	▨	▨	▨	▨	▨	▨						
T6.4																						▨	▨	▨	▨	▨	▨	▨	▨	▨
WP7																														
WP8																														

Fig. 7. Generated Gantt chart

5 Conclusions

We have given an overview of metamodelling, concentrating on small examples and key lessons for grammarware researchers. In particular, we have tried to highlight the key differences between building a language via grammars, and building a language via metamodels. In practice, this involves different technology choices, leading to different implications. With metamodelling, the unification power of modelling is critical: everything (including models and metamodels) can be treated as models, thus allowing engineers to use and reuse tools and infrastructure for multiple purposes. Eclipse is a good example of such flexible and reusable infrastructure. The downside of metamodelling is its hidden complexity: behind every model, there may be a complex metamodel, but also complex metamodelling infrastructure, which can make it difficult to understand how models have been implemented, but also how models and metamodels should change over time. Nevertheless, the power of metamodelling and its infrastructure can lead to practical automation of numerous repetitive tasks, using generic and standardised tools.

References

1. Atkinson, C., Kühne, T.: Model-driven development: A metamodeling foundation. IEEE Software 20(5), 36–41 (2003)
2. Bézivin, J.: Model engineering for software modernization. In: WCRE, p. 4 (2004)
3. Bézivin, J.: On the unification power of models. Software and System Modeling 4(2), 171–188 (2005)
4. Bézivin, J., Bouzitouna, S., Del Fabro, M.D., Gervais, M.-P., Jouault, F., Kolovos, D.S., Kurtev, I., Paige, R.F.: A Canonical Scheme for Model Composition. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 346–360. Springer, Heidelberg (2006)
5. Clark, T., Sammut, P., Willans, J.: Applied metamodelling: a foundation for language driven development, 2nd edn. (2004), http://eprints.mdx.ac.uk/6060/1/Clark-Applied_Metamodelling_%28Second_Edition%29%5B1%5D.pdf
6. de Lara, J., Guerra, E.: Deep Meta-modelling with MetaDepth. In: Vitek, J. (ed.) TOOLS 2010. LNCS, vol. 6141, pp. 1–20. Springer, Heidelberg (2010)
7. Object Management Group. Human-Usable Textual Notation (HUTN) Specification (2004), http://www.omg.org/technology/documents/formal/hutn.html
8. Object Management Group. A proposal for an MDA foundation model (April 01, 2005), http://www.omg.org/cgi-bin/doc?ormsc/05-04-01
9. Object Management Group. Meta-object facility 2.0 core specification (2006), http://www.omg.org/spec/MOF/2.0/
10. Object Management Group. Unified Modelling Language (UML) 2.2 Specification (2009), http://www.omg.org/spec/UML/2.2/
11. Object Management Group. Diagram definition v1.0 ftf specification (2011), http://www.omg.org/cgi-bin/doc?ptc/2011-07-13
12. ISO. ISO/IEC 14977:1996 Extended BNF (1996), http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=26153
13. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice-Hall (1997)
14. Zaytsev, V.: BNF WAS HERE: What Have We Done About the Unnecessary Diversity of Notation for Syntactic Definitions. In: Mernik, M., Bryant, B. (eds.) Programming Languages Track, Proceedings of the 27th ACM Symposium on Applied Computing, SAC 2012, Riva del Garda, Trento, Italy, vol. II, pp. 1910–1915. ACM (March 2012)

Temporal Constraint Support for OCL[*]

Bilal Kanso and Safouan Taha

SUPELEC Systems Sciences (E3S) - Computer Science Department
3 rue Joliot-Curie, F-91192 Gif-sur-Yvette cedex, France
{Bilal.Kanso,Safouan.Taha}@supelec.fr

Abstract. The Object Constraint Language is widely used to express precise and unambiguous constraints on models and object oriented programs. However, the notion of temporal constraints, controlling the system behavior over time, has not been natively supported. Such temporal constraints are necessary to model reactive and real-time systems. Although there are works addressing temporal extensions of OCL, they only bring syntactic extensions without any concrete implementation conforming to the OCL standard. On top of that, all of them are based on temporal logics that require particular skills to be used in practice.

In this paper, we propose to fill in both gaps. We first enrich OCL by a pattern-based temporal layer which is then integrated into the current Eclipse's OCL plug-in. Moreover, the temporal constraint support for OCL, that we define using formal scenario-based semantics, connects to automatic test generators and forms the first step towards creating a bridge linking model driven engineering and usual formal methods.

Keywords: OCL, Object-oriented Programming, Temporal constraints, Eclipse/MDT, Model-Driven Engineering, Formal Methods.

1 Introduction

The Object Constraint Language (OCL) is an expression-based language used to specify constraints in the context of object-oriented models [2]. It is equivalent to a first-order predicate logic over objects, but it offers a formal notation similar to programming languages. OCL may complete the specification of all object-oriented models, even if it is mostly used within UML diagrams.

The OCL constraints may be invariants that rule each single system state, or preconditions and postconditions that control a one-step transition from a pre-state to a post-state upon the call of some operation. Thus, it is not possible to express constraints of dynamic behavior that involve different states of the model at different instants. This is essentially due to the absence of the notion of time and events in OCL. This limitation seems to form the main obstacle which the use of OCL faces today in the verification and validation areas. The standard OCL published in [2] does not provide any means of featuring temporal quantification, nor of expressing temporal properties such as safety or liveness.

[*] This work was funded by the French ANR TASCCC project (ANR-09-SEGI-014) [1].

Adding a temporal layer to the OCL language forms a primordial step towards supporting the automatic verification and validation of object-oriented systems.

In this paper, we propose a temporal extension of OCL that enables modelers/developers to specify temporal constraints on object-oriented models. We do so by relying on *Dwyers*'s patterns [3]. A temporal constraint consists in a pattern combined with a scope. A pattern specifies the behavior that one wants to exhibit/avoid, while a scope defines the piece of execution trace to which a given pattern applies. This allows us to write temporal OCL constraints without any technical knowledge of formalisms commonly used to describe temporal properties such as LTL or CTL logics. We integrated this extension into the Eclipse/MDT current OCL plug-in.

In this work, we are also interested in formal methods applied to oriented-object systems for guiding software testing. Indeed, we will use the temporal properties that were formally written in our OCL temporal extension, as test purposes. Test purposes are commonly used to focus on testing particular aspects of models, avoiding other irrelevant ones [4,5]. Thinking of functional and security properties when writing test purposes is a common practice, but it has not been automated. Despite the interest of test purposes in the process of test case derivation, the lack of formal methods for their description and tools for their automatic generation forms one of the serious obstacles which the use of testing techniques faces today in the industrial areas.

To support test purposes specification, we enrich our language with formal scenario-based semantics; the behavior to be tested is expressed as a temporal OCL expression, and then automatically translated into a regular expression. This latter is generic enough to be used by a large family of generation techniques of test cases from object-oriented models. After its integration into the Eclipse/MDT current OCL plug-in, our language provides a framework not only to constrain dynamic behavior of object-oriented systems, but other to generate functional tests for objects and verify their properties. The language is indeed used in the validation of smart card product security [1]. It provides a means to express security properties (provided by *Gemalto*) on UML specification of the *GlobalPlatform*, the latest generation smart card operating system. In this work [6], the test requirements are expressed as OCL temporal constraints described in our proposed language and then transformed into test scenarios. These are then animated using the *CertifyIt* tool, provided by the *Smartesting* company to generate test cases.[1]

This paper is organized as follows. Section 2 presents the OCL language while Section 3 discusses its limitations on the expression of temporal aspects. Section 4 recalls the related works. Section 5 describes our proposal for extending OCL to support time and events. Section 6 provides the formal scenario-based semantics of our language. Section 7 describes the implementation of the proposed extension in the Eclipse's OCL plug-in and Section 8 presents the use of our proposal as a test purpose framework within the TASCCC project. Finally, Section 9 concludes and presents the future work.

[1] www.globalplatform.org, www.gemalto.com, www.smartesting.com

2 Object Constraint Language (OCL)

OCL is a formal assertion language, easy to use, with precise and unambiguous semantics [2]. It allows the annotation of any object-oriented model, even if it is most used within UML diagrams. OCL is very rich, it includes fairly complete support for:

– *Navigation operators* to navigate within the object-oriented model,
– *Set/Sequence operations* to manipulate sets and sequences of objects,
– *Universal/Existential Quantifiers* to build first order (logic) statements.

We briefly recall these OCL capabilities by means of an example. The UML class diagram in Figure 1 represents the structure of a simple *software system*. This system has a *free_memory* attribute corresponding to the amount of free memory that is still available, and the following three operations:

– *load(app: Application)*: downloads the application *app* given as a parameter.
– *install()*: installs interdependent applications already loaded. Different applications can be loaded before a single call of *install()*, but only applications having all their dependencies already loaded are installed.
– *run(app: Application)*: runs the application *app* given as a parameter that should be both already loaded and installed.

A system keeps references to the previously installed applications using the association end-point *installed_apps*. An *Application* has a *size* attribute and keeps references to the set of applications it depends on using the association end-point *dependencies*. We will use this illustrative example along this work.

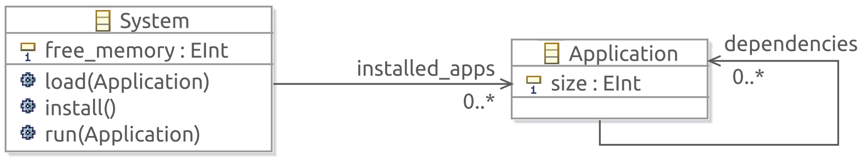


Fig. 1. A model example

Exp 1 describes three typical OCL expressions. The first expression *all_apps_dependencies_installed* verifies that every installed application has its dependencies installed as well. The *all_dependencies* expression is a recursive function that builds the transitive closure of the (noncyclic) *dependencies* association. The *may_install_on* expression is a boolean function which has a system as parameter and verifies that installing the application with its dependencies fits into the system's free memory.

```
context System
def: all_apps_dependencies_installed: Boolean =
    self.installed_apps->forAll(app:Application | self.installed_apps->
    includesAll(app.dependencies))

context Application
def: all_dependencies: Set(Application) =
    self.dependencies.all_dependencies->asSet()->including(self)

def: may_install_on(sys: System): Boolean =
  (self.all_dependencies - sys.installed_apps).size->sum() < sys.free_memory
```

Exp. 1. OCL Expressions

Exp 1 illustrates the OCL ability to navigate the model (*self.installed_apps*, *app.dependencies*), select collections of objects and manipulate them with functions (*including()*, *sum()*), predicates (*includesAll()*) and universal/existential quantifiers (*forAll()*) to build boolean expressions.

3 OCL Limitations

3.1 OCL is a First-Order Predicate Logic

OCL boolean expressions are first order predicate logic statements over a model state. They are written with a syntax which is similar to programming languages. Such OCL expressions are evaluated over a single system state, which is a kind of a snapshot given as an object diagram at some point in time. An object diagram is a particular set of objects (class instances), slots (attribute values), and links (association instances) between objects. For example, an equivalent first order statement of *all_apps_dependencies_installed* expression is:

$$\forall s \in Sys, \forall a, b \in App,\ (s, a) \in Ins \wedge (a, b) \in Dep \Rightarrow (s, b) \in Ins$$

where a *state* (object diagram) is a tuple $(Sys, App, Ins, Dep, free, size)$

- Sys is the set of *System* objects
- App is the set of *Application* objects
- $Ins \subseteq Sys \times App$ is the set *installed_apps* links, $(s, a) \in Ins$ iff the *Application* instance a is installed on the *System* instance s
- $Dep \subseteq App \times App$ is the set *dependencies* links, $(a, b) \in Dep$ iff the *Application* instance a depends on the *Application* instance b
- $free : Sys \to \mathbb{N}$ is the function that associates each *System* instance s to the amount of free memory available
- $size : App \to \mathbb{N}$ is the function that associates each *Application* instance a to its memory size.

The first order logic allows quantification over finite and infinite domains[2] contrary to the OCL language which has no free quantification over infinite domains such as $\mathbb{Z}$ or $\mathbb{N}$. Indeed, in OCL, one distinguishes three kinds of domains:

[2] Note that the first order logic over the set theory (with possibly many infinite sets) is undecidable.

– Set of objects.
– Set of some Primitive Type values.
– *Time* that is the set of all instants of the model's life. It corresponds to $\mathbb{N}$ if time is discrete, $\mathbb{Q}$ if time is dense or $\mathbb{R}$ if time is continuous.

The OCL expressions presented in Exp 1 are typical examples of OCL quantification (*forAll(), exists()*) over sets of objects (e.g. *self.dependencies*) and sets of primitive type values (e.g. *self.all_dependencies.size* of PrimitiveType::Integer). Since these sets are selections/subsets of an object diagram, they are finite by construction. Hence, there is no limitation to use OCL quantifiers over them. However, since *Time* is intrinsically infinite, quantification over it is restricted within OCL. This last point will be detailed in the next subsections.

3.2 Temporal Dimension

As previously mentioned, the OCL expressions are evaluated over a single system state at some point in time. But, the OCL language also provides some implicit quantification over time by means of OCL rules. An OCL *rule* is a temporal quantification applied to an OCL boolean expression, and may be an invariant of a class, a pre- or a post-condition of an operation.

The expression within an invariant rule has be to be satisfied throughout the whole life-time of all instances of the context class. The first expression in Exp 2 specifies the invariant which requires, in all system states, a nonempty free memory and the installation of dependencies of all installed applications. The precondition and postcondition are used to specify operation contracts. A precondition has to be true each time the corresponding operation is called, and a postcondition has to be true each time right after the corresponding operation execution has terminated. The second expression in Exp 2 describes the rule that provides the *load(app: Application)* contract. It requires that the application given as a parameter is not already installed and there is enough memory available to load it. Then, it ensures that the *free_memory* attribute is updated using the *@pre* OCL feature.

```
context System
inv : self.free_memory > 0 and all_apps_dependencies_installed = true

context System::load(app: Application):
pre : self.installed_apps->excludes(app) and self.free_memory > app.size
post: self.free_memory = self.free_memory@pre - app.size
```

Exp. 2. OCL rules

The operation parameters can be used within a pre or a post-condition rule, but the *@pre* OCL feature is only used within a post-condition rule. When *@pre* is used within the boolean expression of a post-condition rule, it is evaluated over two system states, one right before the operation call and one right after its execution. In other words, OCL expressions describe a single system state or a one-step transition from a previous state to a new state upon the call of some operation. Therefore, there is no way to make OCL expressions involving different states of the model at different points in time. OCL has a very limited temporal dimension.

To illustrate the temporal limits of OCL, let us consider the following temporal properties for the example presented in Figure 1:

safety_1: each application can be loaded at most one time
safety_2: an application load must precede its run
safety_3: there is an install between an application loading and its run
liveness: each loaded application is installed afterwards

Such temporal properties are impossible to specify in OCL without at least enriching the model structure with state variables. In temporal logics [7], we formally distinguish the safety properties from the liveness ones. *Safety* properties for bad events/states that must not happen and *liveness* properties for good events/states that should happen. As safety properties consider finite behaviors, they can be handled by modifying the model and adding variables which save the system history. If we consider the first safety property, one solution is to save within a new attribute *loaded_apps* the set of applications already loaded, but not yet installed and then check in the load(app: Application) precondition that the loaded application is neither installed, nor loaded:

```
context System::load(app: Application):
pre : self.installed_apps->excludes(app) and
      self.loaded_apps->excludes(app) and self.free_memory > app.size
```

Even if specifying complementary temporal OCL constraints must not alter the model, such case-by-case techniques are of no use when specifying liveness properties that handle infinite behaviors.

In this work, we are mainly interested in temporal constraints from the temporal logics point of view, when they are ruling the dynamic behavior of systems. They specify absence, presence and ordering of the system life-time steps. A step may be a state that holds for a while or an event occuring at some point of time.

3.3 Events

An event is a predicate that holds at different instants of time. It can be seen as a function $P : Time \to \{true, false\}$ which indicates at each instant, if the event is triggered. The subset $\{t \in Time \mid P(t)\} \subseteq Time$ stands then for all time instants at which the event P occurs. When quantifying time, we need to select such subsets of $Time$ that correspond to events. We commonly distinguish five kinds of events in the object-oriented paradigm:

Operation call instants when a sender calls an operation of a receiver object
Operation start instants when a receiver object starts executing an operation
Operation end instants when the execution of an operation is finished
Time-triggered event that occurs when a specified instant is reached
State change that occurs each time the system state changes (e.g when the value of an attribute changes). Such an event may have an OCL expression as a parameter and occurs each time the OCL expression value changes.

OCL only provides an implicit universal quantification over *operation call* events within pre-conditions and a universal quantification over *operation end* events within post-conditions. However, it lacks the finest type of events which is *state change*. State change events are very simple, but powerful construct. It can replace other types of events. Suppose we add a chronometric clock that is now a part of our system. This common practice will create a new object *clock* within our system that has a *time* attribute. Each change of that attribute will generate a state-change event. A time-triggered event of some specified *instant* will be then one particular state-change in which the OCL boolean expression *clock.time = instant* becomes true.

To replace operation call, start and end events using the state-change event, we need to integrate the stack structure within the system model. We do not recommend this technique that is in contradiction to the model-driven engineering approach because it pollutes the system model with platform specific information and ruins all the abstraction effort.

3.4 Quantification

OCL has no existential quantification over time or events. For example, the second safety property we previously proposed needs existential quantification: **it exists** a *load()* operation call that precedes a *run()* operation call.

The other quantification limitation we identified is that OCL sets its few temporal quantification constructs within OCL rules, prior to the quantification over objects within the OCL expressions. Again, the second safety property needs quantifying over objects prior to quantifying over time: **for all** application instance *app*, **it exists** a *load(app)* operation call that precedes a *run(app)* operation call. We intend to relate the load event of the particular application *app* with its run. This quantification order is the way to define the relations we may need between events.

4 Related Work

Several extensions have been proposed to add temporal constraints to the OCL language. [8] presents an extension of OCL, called TOCL, with the basic operators of the common linear temporal logic (LTL). Both future and past temporal operators are considered. This paper only provides a formal description of the extension based on *Richters*'s OCL semantics [9]. It gives no explanation of how all presented formal notions can be implemented. In [10,11], authors propose an extension of OCL for modeling real-time and reactive systems. A general notion of time and event is introduced, providing a means to describe the temporal behavior of UML models. Then, OCL is enriched by (1) the temporal operator `@event` (inspired by the OCL operator `@pre`) to refer to the expression value at the instant when an event occurs, and (2) the time modal operators always and sometime. [12] proposes a version of CTL logic, called BOTL, and shows how to map a part of OCL expressions into this logic. Indeed, there is no extension of OCL by temporal operators, but a theoretical precise mapping of a part of OCL into BOTL. [13] provides an OCL extension, called EOCL, with CTL temporal operators. This extension is strongly inspired by BOTL [12], and allows model checking EOCL properties on UML models expressed as abstract state machines. A tool (SOCLE), implementing this extension, is briefly presented with verification issues in mind; however, there is no tool available at the project site [14]. Similarly, Flake et al. [15] formalize UML Statechart within the Richters's OCL semantics and extend OCL with Clocked CTL in order to provide a sound basis for model-checking. [16] proposes templates (e.g. after/eventually template) to specify liveness properties. A template is defined by two clauses: a cause and a consequence. A cause is the keyword after followed by a boolean expression, while a consequence is an OCL expression prefixed by keywords like eventually, immediately, infinitely, etc. These templates are formally translated into observational μ-calculus logic. This paper gave no means to OCL developers to implement such templates. It only formally addresses some liveness properties; other liveness and safety properties are not considered. [17] adds to OCL unary and binary temporal operators such as until and always and [18] proposes past/future temporal operators to specify business components. Both [17] and [18]'proposals are far from being used in the context of concrete implementation conforming to the standard OCL [2]. For instance, in [18], an operator may be followed by user-defined operations (with possible side effects) that are not concretely in conformance with the standard OCL. Table 1 summarizes the state of the art and emphasizes the need for a complete approach.

Table 1. Related work

Approach	Temporal Layer	Event Constructs	Quantification Order	Tooling	Formal Semantics
Ziemann et al. [8]	LTL + past	no	no	no	trace semantics
Calegari et al. [10,11]	future/past modal operators	yes	no	not conforming to OCL standard	trace semantics
Distefano et al. [12]	CTL	no	no	no	BOTL
Mullins et al. [13]	CTL	no	no	not conforming to OCL standard	inspired by BOTL
Bradfield et al. [16]	template clauses (response pattern)	no	no	no	observational μ-calculus
Ramakrishnan et al. [17] Conrad et al. [18]	future/past modal operators	no	no	no	no
Flake et al [15]	Clocked CTL	state-oriented	no	no	trace semantics

Among temporal constraints we have the particular case of timings properties that are commonly used within the real-time systems development. Timings are static duration constraints between event occurrences, they are necessary to specify WCETs (Worst Case Execution Time), deadlines, periods... There are surprisingly many efforts to annotate statically this kind of constraints using UML profiles. MARTE [19] is an UML extension defining stereotypes (RTSpecification, RTFeature, RTAction) that annotate classes and operations to specify their timings.

5 OCL Temporal Extension

After identifying the OCL limitations that are absences of temporal operators, event constructs and free quantification (see Section 3), and after reviewing most existing OCL temporal extensions (see Section 4), we give in the following our contribution about OCL temporal extension:

– A *pattern-based language* contrary to most of OCL temporal extensions that are based on temporal logics such as LTL or CTL (see Section 4). The technicality and the complexity of these formalisms give rise naturally to difficulties even to the impossibility, in some cases, of using them in practice [3];
– Enrichment of OCL by the notion of *events* that is completely missing in the existing temporal extensions of OCL;
– A user-friendly syntax and formal *scenario-based semantics* of our OCL temporal extension (see Section 6);
– A *concrete implementation* conforming to the standard OCL [2]. In fact, all the works mentioned in Section 4 only address the way OCL has to be extended to deal with temporal constraints. The main purpose behind them was to use OCL in verification areas such as model checking. However, they did not reach this last step, at least not in practice, due to the absence of concrete implementations conforming to the standard OCL [2] of the proposed extensions.

5.1 Temporal Patterns

Formalisms such as linear temporal logic (LTL) and tree logic (CTL) have received a lot of attention in the formal methods community in order to describe temporal properties of systems. However, most engineers are unfamiliar with such formal languages. It requires a lot of effort to bridge the semantic gap between the formal definitions of temporal operators and practice. To shed light on this obstacle, let us consider the *safety_3* property, its equivalent LTL formula looks like:

$$\Box(load \wedge \neg run \Rightarrow ((\neg run \ \mathsf{U} \ (install \wedge \neg run)) \vee \neg \diamond run))$$

It means that each time ($\Box$) we have a load, this implies that there will be no run at least until (U) the install happens or there will be no run at all in the future ($\neg\diamond run$). To avoid such error-prone formulas, *Dwyer* et al. have proposed a pattern-based approach [3]. This approach uses specification patterns that, at a higher abstraction level, capture recurring temporal properties. The main idea is that a temporal property is a combination of one **pattern** and one **scope**. A scope is the part of the system execution path over which a pattern holds.

Patterns [3] proposes 8 patterns that are organized under a semantics classification (left side of Figure 2). One distinguishes occurrence (or non-occurrence) patterns from order patterns.

Occurrence patterns are: (*i*) Absence: an event never occurs, (*ii*) Existence: an event occurs at least once, (*iii*) BoundedExistence has 3 variants: an event occurs k times, at least k times or at most k times, and (*iv*) Universality: a state is permanent.

Order patterns are: (*i*) Precedence: an event P is always preceded by an event Q, (*ii*) Response: an event P is always followed by an event Q, (*iii*) ChainPrecedence: a sequence of events $P_1, \ldots, P_n$ is always preceded by a sequence $Q_1, \ldots, Q_n$ (it is a generalization of the Precedence pattern), and (*iv*) ChainResponse: a sequence of events $P_1, \ldots, P_n$ is always followed by a sequence $Q_1, \ldots, Q_n$ (it is a generalization of the Response pattern as well).

Scopes [3] proposes 5 kinds of scopes (right side of Figure 2): (*i*) Globally covers the entire execution, (*ii*) Before Q covers the system execution up to the first occurrence of Q, (*iii*) After Q covers the system execution after the first occurrence of Q, (*iv*) Between Q and R covers time intervals of the system execution from an occurrence of Q to the next occurrence of R, and (*v*) After Q until R is same as the Between scope in which R may not occur.

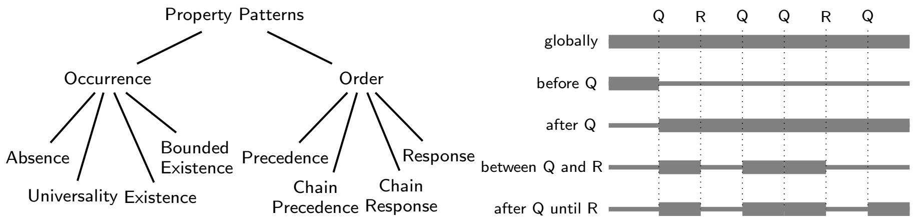


Fig. 2. Dwyer's patterns and scopes

Back to our temporal property *safety_3* : *there is an install between an application loading and its run.* It simply corresponds to the Existence pattern (exists *install*) combined with the Between scope (between load and run). It is clear that the patterns of *Dwyer* et al. dramatically simplify the specification of temporal properties, with a fairly complete coverage. Indeed, they collected hundreds of specifications and they observed that 92% of them fall into this small set of patterns/scopes [3]. Furthermore, a complete library is provided [20], mapping each pattern/scope combination to the corresponding formula in many formalisms (e.g. LTL, CTL, QREs, μ-calculus).

For these reasons, we adopt this pattern-based approach for the temporal part of our OCL extension and we bring enhancements to improve the expressiveness:

– *Dwyer* et al. have chosen to define scopes as right-open intervals that include the event marking the beginning of the scope, but do not include the event marking the end of the scope. We extend scopes with support to open the scope on the left or close it on the right. This adds one variant for both the Before and After scopes and three supplementary variants for the Between and After . . . until scopes. We have chosen open intervals as default semantics.
– In *Dwyer* et al. work, Between and After . . . until scopes are interpreted relative to the first occurrence of the designated event marking the beginning

of the scope (Figure 2). We kept this as default semantics and we provide an option to select the last occurrence semantics.

– To respect the classical semantical conventions of temporal logics, we renamed the After . . . until scope as After . . . **unless**. Then to improve the usability, we added the scope When that has an OCL boolean expression as a parameter and that covers the execution intervals in which this OCL expression is evaluated to true. The When scope is derived from the After . . . unless scope as follows:

$$\textbf{When } P \equiv \textbf{After } \mathsf{becomesTrue}(P) \textbf{ unless } \mathsf{becomesTrue}(\mathit{not}\ P)$$

The becomesTrue event is introduced in Subsection 5.2.

– Order patterns describe sequencing relationships between events and/or chains of events. The *Dwyer* et al. semantics adopt non strict sequencing. For example, A, B (is) preceding B, C in both A, B, C and A, B, B, C executions. We add features to specify strict sequencing for an order pattern. For example, A, B (is) preceding **strictly** B, C only in the A, B, B, C execution. We provide same constructs to have strict sequencing within one chain of events, A, B to denote a non strict sequencing and $A; B$ for a strict one.

– In *Dwyer* et al. work, there is no construct equivalent to the temporal operator Next. For example, A (is) preceding C in both $A; C$ and $A; B; C$ executions. We add features to specify the Next temporal operator for an order pattern. For example, A (is) preceding **directly** C only in the $A; C$ execution. The directly feature is a particular case of strict sequencing.

These enhancements are inspired by our needs within the TASCCC project [1] and the *Dwyer*'s notes about the temporal properties that were not supported [20]. It is obvious that these enhancements improve the requirement coverage (i.e. 92%) shown by *Dwyer*, but we did not measure it precisely.

5.2 Events

Events are predicates to specify sets of instants within the time line. We discussed in Section 3 the different types of events in the object-oriented approach. There are operation (call/start/end) events, time-triggered events and state change events. We have seen that when integrating the clock into the system, time-triggered events are particular state change events. Hence, we only need to extend OCL with the necessary construct for both operation and state change events.

We aim to connect our OCL temporal extension to formal methods such as model-checking and test scenario generation. Formal methods are mainly based on the synchronous paradigm that has well-founded mathematical semantics and that allows formal verification of the programs and automatic code generation. The essence of the synchronous paradigm is the atomicity of reactions (operation calls) where all the occurring events during such a reaction are considered simultaneous. In our work, we will adopt the synchronous paradigm, and we then merge the operation (call/start/end) events into one call event, named isCalled, that leads the system from a pre-state to a post-state without considering neither observing intermediate change states.

isCalled: is a generic event construct that unifies both operation events and state change events. It has three optional parameters:

- op: is the called operation. The keyword *anyOp* is used if no operation is specified
- pre: is an OCL expression that is a guard over the system pre-state and/or the operation parameters. The operation invocation will lead to a call event only if this guard is satisfied by the pre-state of the call. If it is not satisfied, the event will not occur even if the operation is invoked.
- post: is an OCL expression that is a guard over the system post-state and/or the return value. The operation invocation will lead to a call event only if this guard is satisfied by the post-state of the call.

becomesTrue: is a state change event that is parameterized by an OCL boolean expression P, and designates a step in which P becomes true, i.e. P was evaluated to false in the previous state. In the object-oriented paradigm, a state change is necessarily a consequence of some operation call, therefore the **becomesTrue** construct is a syntactic sugar and stands for any operation call switching P to true (see Figure 3):

$$\mathsf{becomesTrue}(P) \;\equiv\; \mathsf{isCalled}(op:\; anyOp,\; pre:\; not\ P,\; post: P)$$

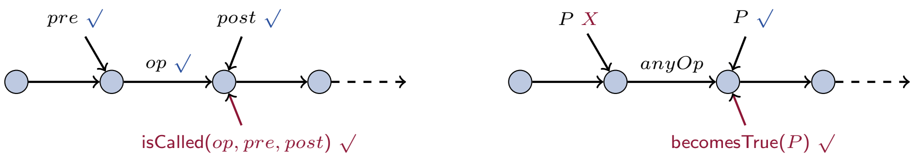


Fig. 3. Events

We also define two generic operators/constructors over events:

Disjunction: $ev1 \mid ev2$ occurs when $ev1$ occurs **or** $ev2$ occurs

Exclusion: $ev1 \ \setminus\ ev2$ occurs when $ev1$ occurs **and** $ev2$ does **not**

Other operators (Negation, Conjunction, ...) can be easily derived:

$$\begin{aligned}
\mathbf{not}(event) &\equiv \mathsf{isCalled}(anyOp, true, true) \ \setminus\ event \\
\mathsf{becomesTrue}(P_1) \wedge \mathsf{becomesTrue}(P_2) &\equiv \mathsf{isCalled}(anyOp, \neg P_1 \wedge \neg P_2, P_1 \wedge P_2) \\
&\neq \mathsf{becomesTrue}(P_1 \wedge P_2)
\end{aligned}$$

5.3 Quantification

Our OCL extension supports universal quantification over objects prior to quantification over time. The OCL feature let $Variables$ in can be used within our OCL extension on the top of temporal expressions (see Section 7.3).

6 Semantics

Several formal semantics have been provided to describe the OCL language. These are not given in this paper, only the semantics of our OCL temporal extension are defined. Interested readers may refer to [2,9].

A test case is a scenario/sequence of operations calls. Since we are interested in test cases generation, we adopt a scenario-based semantics over the synchronous paradigm to formally describe our temporal extension. The essence of that paradigm is the atomicity of reactions (operation calls) where all the events occurring during such a reaction are considered as simultaneous. A reaction is one atomic call event, that leads the system directly from a pre-state to a post-state without going through intermediate states.

6.1 Events

We define the set of all atomic events of a given object model as follows:

Definition 1 (Alphabet of Atomic Events). *Let $\mathbb{O}$ be the set of all operations and $\mathbb{E}$ be the set of all OCL expressions of an object model $\mathcal{M}$. The* ***alphabet Σ of atomic events*** *is defined by the set $\mathbb{O} \times \mathbb{E} \times \mathbb{E}$.*

An atomic event $e \in \Sigma$ then takes the form: $e = (op, pre, post)$. It stands for a call of the operation *op* in a context where *pre* stands for the precondition satisfied in the pre-state and *post* for the postcondition satisfied in the post-state.

We now give the formal meaning of the notion of events introduced in the grammar presented in Figure 5.

Definition 2 (Events). *Let Σ be the alphabet of atomic events, $\mathbb{O}$ be the set of all operations and $\mathbb{E}$ the set of all OCL expressions. An* ***event*** *is either an* $\mathsf{isCalled}(op, pre, post)$ *or* $\mathsf{becomesTrue}(\mathsf{P})$ *where:*

$$\begin{aligned}\mathsf{isCalled}(op, pre, post) &= \{(op, pre', post') \in \Sigma \mid pre' \Longrightarrow pre,\ post' \Longrightarrow post\} \text{ and}\\ \mathsf{becomesTrue}(P) &= \{(op, pre, post) \in \Sigma \mid op \in \mathbb{O},\ pre \Longrightarrow \neg P,\ post \Longrightarrow P\}\end{aligned}$$

Definition 2 calls for the following three comments:

- In our language, the operation *op* can be replaced by *anyOp* the set of all operations as follows:
  $$\mathsf{isCalled}(anyOp, pre, post) = \bigcup_{op \in \mathbb{O}} \{(op, p, q) \in \Sigma \mid\ p \Rightarrow pre,\ q \Rightarrow post\}$$
- $\mathsf{becomesTrue}(P)$ is equivalent to $\mathsf{isCalled}(anyOp, \neg P, P)$. We keep this primitive to make our language easier to use.
- An event does not represent a single atomic event, but a specific subset of atomic events. It is intuitively the set of all atomic events in which the operation *op* is invoked, in a pre-state which implies the expression *pre* and leading to a post-state which implies the expression *post*. The set of all events is then defined[3] as the set 2^{Σ}.

[3] 2^X denotes the set of all subsets of X.

A disjunction (resp. exclusion) of events is an event. By considering events as subsets of Σ, the semantics of the disjunction (resp. exclusion) constructor $|$ (resp. $\backslash$) over events is given as a simple union (resp. minus) over sets.

Definition 3 (Operators over Events). *Let Σ be the alphabet of atomic events. The* ***disjunction operator*** $|$ *and the* ***exclusion operator*** $\backslash$ *over Σ are defined as follows:*

$$\begin{array}{ll} | \ : 2^{\Sigma} \times 2^{\Sigma} \to 2^{\Sigma} & \qquad \backslash \ : 2^{\Sigma} \times 2^{\Sigma} \to 2^{\Sigma} \\ \quad (E_1, E_2) \mapsto E_1 \cup E_2 & \qquad \quad (E_1, E_2) \mapsto E_1 - E_2 \end{array}$$

6.2 Scenarios

We introduce the notion of a scenario, which allows us to interpret our OCL temporal expressions. A *scenario* σ in a model $\mathcal{M}$ is a finite sequence $(e_0, \ldots, e_n) \in \Sigma^*$ of atomic events. Such a scenario embodies the temporal order between atomic event triggering, where the notion of time is implicitly specified. In a scenario $(e_0, \ldots, e_n)$, there is a logical time associated to the atomic event e_0 which precedes the logical time associated to the atomic event e_1, and so on.

In the following, for every scenario $\sigma \in \Sigma^*$ of length n, we write $\sigma = (\sigma(0), \ldots, \sigma(n-1))$. Thus, $\sigma(i)$ denotes the atomic event at index i and $\sigma(i:j)$ the part of σ containing the sequence of atomic events between i and j.

6.3 Temporal Expressions

We define here the semantics for our temporal expressions that are evaluated over event-based scenarios.

Definition 4 (Scopes). *Let $\mathbb{S}$ be the set of scopes defined in the grammar presented in Figure 5. The* ***semantics of a scope*** *$s \in \mathbb{S}$ is given by the function $[[s]]^s : \Sigma^* \longrightarrow 2^{\Sigma^*}$ defined for every $\sigma \in \Sigma^*$ of length n as follows:*

- $[[\mathsf{globally}]]^s(\sigma) = \{\sigma\}$
- $[[\mathsf{before}\ E]]^s(\sigma) = \{\sigma(0:i-1) \ \mid \ \sigma(i) \in E \textit{ and } \forall k, 0 \leq k < i, \sigma(k) \notin E\}$
- $[[\mathsf{After}\ E]]^s(\sigma) = \{\sigma(i+1:n-1) \ \mid \ \sigma(i) \in E \textit{ and } \forall k, 0 \leq k < i, \sigma(k) \notin E\}$
- $[[\mathsf{between}\ E_1 \mathsf{\ and\ } E_2]]^s(\sigma) = \{\sigma(i_k+1:j_k-1) \mid$
  $\forall k \geq 0, i_k < j_k < i_{k+1}, \sigma(i_k) \in E_1, \sigma(j_k) \in E_2,$
  $\forall m, i_k \leq m < j_k, \sigma(m) \notin E_2 \textit{ and } \forall l, j_k < l < i_{k+1}, \sigma(l) \notin E_1\}$
- $[[\mathsf{after}\ E_1 \mathsf{\ unless\ } E_2]]^s(\sigma) =$
  $\{\sigma(i_k+1:j_k-1) \mid \forall k \geq 0, i_k < j_k < i_{k+1}, \sigma(i_k) \in E_1, \sigma(j_k) \in E_2,$
  $\forall m, i_k \leq m < j_k, \sigma(m) \notin E_2 \textit{ and } \forall l, j_k < l < i_{k+1}, \sigma(l) \notin E_1\}$
  $\cup \{\sigma(i:n-1) \mid \sigma(i) \in E_1, \forall m \geq i, \sigma(m) \notin E_2\}$

Definition 5 (Patterns). *Let $\mathbb{P}$ be the set of patterns defined in the grammar presented in Figure 5. The* ***semantics of a pattern*** *$p \in \mathbb{P}$ is given by the function $[[p]]^p : \Sigma^* \longrightarrow \{true, false\}$ defined for every $\sigma \in \Sigma^*$ as follows:*

- $[[\textsf{never } E]]^p(\sigma) \Leftrightarrow \forall i \geq 0, \sigma(i) \notin E$
- $[[\textsf{always } P]]^p(\sigma) \Leftrightarrow [[\textsf{never}(\textsf{isCalled}(anyOp, \_, \neg P))]]^p(\sigma)$
- $[[E_1 \textsf{ preceding } E_2]]^p(\sigma) \Leftrightarrow \forall i \geq 0, (\sigma(i) \in E_2 \Rightarrow \exists k \leq i, \sigma(k) \in E_1)$
- $[[E_1 \textsf{ following } E_2]]^p(\sigma) \Leftrightarrow \forall i \geq 0, (\sigma(i) \in E_2 \Rightarrow \exists k \geq i, \sigma(k) \in E_1)$
- $[[\textsf{eventually } E\ \alpha \textsf{ times}]]^p(\sigma) \Leftrightarrow \textsf{card}\big(\{i \mid \sigma(i) \in E\}\big) \begin{cases} = k \ \mathit{if}\ \alpha = k \\ \geq k \ \mathit{if}\ \alpha = \textsf{at least } k \\ \leq k \ \mathit{if}\ \alpha = \textsf{at most } k \end{cases}$

Definition 6 (OCL temporal expressions). *The **semantics of an OCL temporal expression*** (pattern, scope) $\in \mathbb{P} \times \mathbb{S}$ *over a scenario* $\sigma \in \Sigma^*$*, denoted by* $\sigma \vDash$ (pattern, scope)*, is defined by:*

$$\sigma \vDash (\textsf{pattern}, \textsf{scope}) \Longleftrightarrow \forall \sigma' \in [[\textsf{scope}]]^s(\sigma),\ [[\textsf{pattern}]]^p(\sigma')$$

Due to the lack of space in this paper, we do not provide the semantics of all variants of patterns and scopes that we defined in our temporal extension, interested readers may refer to [21].

7 Integration within the Eclipse/MDT Tool-Chain

7.1 Structure of Eclipse's OCL Plug-In

The Eclipse/MDT OCL Plug-in [22] provides an implementation of the OCL OMG standard for EMF-based models. It provides a complete support for OCL, but we will only focus on some capabilities that are represented and highlighted in red within Figure 4.

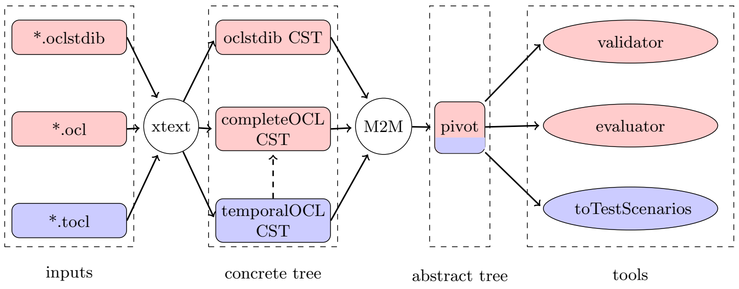


Fig. 4. Eclipse MDT/OCL 4.× with Temporal extension

On the left of Figure 4, there are two Xtext editors that support different aspects of OCL usage. The completeOCL editor for **.ocl* documents that contain OCL constraints, and the OCLstdlib editor for **.oclstdlib* documents that facilitates development of the OCL standard library. This latter is primarily intended for specifying new functions and predicates to use within OCL expressions.

In the middle of Figure 4, the architecture of the OCL plug-in is based around a pivot model. The pivot model isolates OCL from the details of any particular UML or Ecore (or EMOF or CMOF or etc.) meta-model representation. OCL expressions can therefore be defined, analyzed and evaluated for any EMF-based meta-model. Notice that most object-oriented meta-models (e.g. UML) are already specified within EMF.

From left to right, the Xtext framework [23] is used to transform the OCL constraints document to a corresponding Concrete Syntax Tree (CST). Then, using a Model to Model transformation (M2M), it generates the pivot model which corresponds to the Abstract Syntax Tree (AST). Notice that the CST and the AST are both defined within the OMG standard [2]. Finally on the right of Figure 4, the OCL plug-in provides interactive support to validate OCL expressions through their pivot model and evaluate them on model instances.

As highlighted in blue in Figure 4, we integrated our temporal extension within the Eclipse/MDT OCL tool-chain with respect to its architecture. We first extended the OCL concrete grammar to parse **.tocl* documents that contain temporal OCL properties. After that, we extended in Ecore both completeOCLCST and pivot meta-models with all the temporal constructs we defined. We kept both Xtext and M2M frameworks. Finally, in a join work with our partner *LIFC* within the TASCCC project, we developed a tool to transform temporal properties to test scenarios [1,6] (see Section 8).

Due to the lack of space in this paper, we do not give the implementation details on the temporalOCLCST structure and the pivot extension, but the temporal OCL plug-in is published with documentation under a free/open-source license [21].

7.2 Concrete Syntax

We extended the OCL concrete grammar defined within the OMG standard [2] and implemented it within the Eclipse/MDT plug-in. The syntax of our language for **.tocl* documents is summarized in Figure 5.

```
   TempOCL ::= temp (name)? ':' TempSpec
  TempSpec ::= Quantif? Pattern Scope
   Quantif ::= let Variable (',' Variable)* in

   Pattern ::= always OclExpression
             | never Event
             | eventually Event ((at least | at most)? integer times)?
             | EventChain preceding(directly | strictly)? EventChain
             | EventChain following (directly | strictly)? EventChain

     Scope ::= globally
             | before Event ('[' | ']')?
             | after ('[' | ']')? Event
             | between ('[' | ']')? last? Event and Event ('[' | ']')?
             | after ('[' | ']')? last? Event unless Event ('[' | ']')?
             | when OclExpression

     Event ::= CallEvent ( '|' Event)?
             | ChangeEvent ( '|' Event)?
EventChain ::= Event (',' Event)*
             | Event (';' Event)*

   CallEvent ::= isCalled '(' (anyOp | op : Operation)
                 (',' pre : OclExpression)?
                 (',' post : OclExpression)? ')'
 ChangeEvent ::= becomesTrue '(' OclExpression ')'
```

Fig. 5. Grammar of the OCL temporal extension

In this figure, non-terminals are designated in *italics* and terminals in **bold**. (. . .)? designates an optional part and (. . .)* a repetitive part. Finally, the non-terminals imported from the standard OCL grammar (e.g. OclExpression) are underlined. This grammar represents the temporal layer we added to OCL expressions (temporal patterns, events constructs and support of quantification). Taking advantage of the integration within the Eclipse/MDT OCL, we developed, with the help of the Xtext framework, a temporal OCL editor which provides syntax coloring, code formatting, code completion, static validation (well formedness, type conformance. . .) and custom quick fixes, etc. Furthermore, there is an outline view that shows the concrete syntax tree of the temporal OCL property on-the-fly (while typing). Figure 6 illustrates a snapshot of the outline view.

7.3 Examples of Temporal Properties

In Exp 3, the temporal properties we identified in Section 3 are written using our OCL temporal extension. Due to our grammar, the temporal properties seem to be written in natural language. They are ruling call event occurrences with different patterns: following (strict), preceding (non-strict), existence and boundedexistence that are combined with globally and between scopes. Both *safety_2* and *safety_3* properties require quantification over objects prior to temporal operators to specify relations between events. For instance, in *safety_2* we need to specify that the load of an application *app* must precede the run of the same application *app*, and not any other. To do so, we introduced the variable *apptoInstall* which allows us to set the same parameter *apptoInstall* for both *load* and *run* operations.

```
context System
temp safety_1:
        eventually isCalled(load(app:Application)) at most 1 times
        globally

temp safety_2: let apptoInstall : Application in
                isCalled(load(app:Application), pre: app =
                    apptoInstall)
        preceding isCalled(run(app:Application), pre: app =
            apptoInstall)
        globally

temp safety_3: let apptoInstall : Application in
        eventually isCalled(install())
        between    isCalled(load(app:Application), pre: app =
            apptoInstall)
        and        isCalled(run(app:Application), pre: app =
            apptoInstall)

temp liveness:              isCalled(install())
        following strictly isCalled(load(app:Application))
        globally
```

Exp. 3. Temporal OCL constraints

The *safety_3* property is not relevant because having an install call between the load and the run does not ensure that the application will be really installed.

This will not happen if some dependencies are not loaded. To overcome this, we propose in Exp 4 two variants of the *safety_3* property. The *safety_3_v1* property ensures that there is a particular install call, leading to a post-state where the application is installed. The *safety_3_v2* property only specifies that the application becomes installed independently of any operation call (see the becomesTrue semantics in Subsection 5.2). It requires any operation call from a pre-state where the application was not installed to a post-state where it is installed.

```
temp safety_3_v1: let apptoInstall : Application in
        eventually isCalled(install(),
                            post: self.installed_apps->includes(apptoInstall))
        between    ...

temp safety_3_v2: let apptoInstall : Application in
        eventually becomesTrue(self.installed_apps->includes(apptoInstall))
        between    ...
```

Exp. 4. Variants of *Safety_3* property

8 Application: Test Purpose Framework

Testing nowadays programs leads naturally to an exponential state space. When reducing the state space, the testing process derivation may miss test cases of interest and yield irrelevant ones. Test purposes (test intentions) are viewed as one of the most promising directions to cope with this limit [4,5]. They are commonly used to guide the test generation techniques. A test purpose is a description of the part of the specification that we want to test and for which test cases are later generated.

Thinking of functional and security properties when writing test purposes is a common practice, but it has not been automated. We propose to automatically handle test purposes. We first specify the test purposes as OCL temporal properties using our extension. Then, we transform them automatically into regular expressions. This phase was achieved in a join work with our partner *LIFC* who generates automatically regular expressions from properties written in our OCL extension and measures the coverage of the properties [6]. Considering scenario-based semantics (see Section 6), the regular expressions generated are equivalent to the OCL temporal expressions from which they are derived. This automatic transformation is done based on the complete library given by *Dwyer* et *al.* [24,3], mapping each pattern/scope combination to the corresponding formula in many formalisms such as LTL, CTL, QRegExps and μ-calculus.

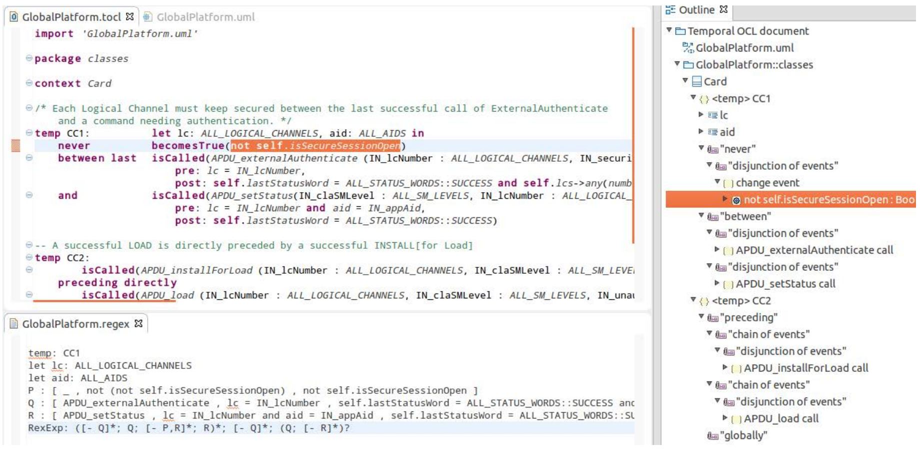
We choose regular expressions as an output language because they are generic enough to be used (with some adaptation) in large family of test generation techniques that are guided by test purposes. For instance, our framework connects naturally to the combinatorial test generation tool *Tobias* [25], that unfolds, in a combinatorial way, tests expressed as regular expressions. Furthermore, approaches such as [4,5,26] that describe their test purposes manually in the form of Labeled Transition systems (LTS) or Input-Labeled Transition systems

(IOLTS), could easily be targets of our framework. We only need to translate the regular expressions produced from the OCL temporal expressions into these two formalisms, which requires a little technical effort.

The first use of this test purpose framework is within the TASCCC project [1] which aims to automatize testing security properties on smart card products and experiment it on *GlobalPlatform*, a last generation smart card operating system. The process of test generation used in this project consists mainly of five phases:

1. Identifying security properties from the Common Criteria standard[4];
2. Writing these security properties using our OCL temporal extension and based on the *GlobalPlatform* UML model distributed by *Smartesting*;
3. Translating the OCL temporal properties into equivalent test scenarios that are regular expressions over an alphabet of API calls;
4. Transforming the test scenarios into test cases by means of *Tobias* [25];
5. Animating the generated test cases on the *GlobalPlatform*. This is performed by the *CertifyIt* tool of *Smartesting*.

Figure 6 is a snapshot of the Temporal OCL editor in which the *GlobalPlatform* security properties (extracted from Common Criteria) were entered and the corresponding regular expressions were generated. The visible property specifies that each logical channel must keep secured between the last successful call of ExternalAuthenticate and a command needing authentication.



```
import 'GlobalPlatform.uml'

package classes

context Card

/* Each Logical Channel must keep secured between the last successful call of ExternalAuthenticate
    and a command needing authentication. */
temp CC1:           let lc: ALL_LOGICAL_CHANNELS, aid: ALL_AIDS in
    never           becomesTrue(not self.isSecureSessionOpen)
    between last    isCalled(APDU_externalAuthenticate (IN_lcNumber : ALL_LOGICAL_CHANNELS, IN_securi
                        pre: lc = IN_lcNumber,
                        post: self.lastStatusWord = ALL_STATUS_WORDS::SUCCESS and self.lcs->any(numb
    and             isCalled(APDU_setStatus(IN_claSMLevel : ALL_SM_LEVELS, IN_lcNumber : ALL_LOGICAL_
                        pre: lc = IN_lcNumber and aid = IN_appAid,
                        post: self.lastStatusWord = ALL_STATUS_WORDS::SUCCESS)

-- A successful LOAD is directly preceded by a successful INSTALL[for Load]
temp CC2:
        isCalled(APDU_installForLoad (IN_lcNumber : ALL_LOGICAL_CHANNELS, IN_claSMLevel : ALL_SM_LEVE
    preceding directly
        isCalled(APDU_load (IN_lcNumber : ALL_LOGICAL_CHANNELS, IN_claSMLevel : ALL_SM_LEVELS, IN_una
```

```
temp: CC1
let lc: ALL_LOGICAL_CHANNELS
let aid: ALL_AIDS
P : [ _ , not (not self.isSecureSessionOpen) , not self.isSecureSessionOpen ]
Q : [ APDU_externalAuthenticate , lc = IN_lcNumber , self.lastStatusWord = ALL_STATUS_WORDS::SUCCESS anc
R : [ APDU_setStatus , lc = IN_lcNumber and aid = IN_appAid , self.lastStatusWord = ALL_STATUS_WORDS::SU
RexExp: ([- Q]*; Q; [- P,R]*; R)*; [- Q]*; (Q; [- R]*)?
```

```
▼ Temporal OCL document
    GlobalPlatform.uml
  ▼ GlobalPlatform::classes
    ▼ Card
      ▼ <temp> CC1
        ▸ lc
        ▸ aid
        ▼ "never"
          ▼ "disjunction of events"
            ▼ change event
              ▸ not self.isSecureSessionOpen : Boo
        ▼ "between"
          ▼ "disjunction of events"
            ▸ APDU_externalAuthenticate call
          ▼ "disjunction of events"
            ▸ APDU_setStatus call
      ▼ <temp> CC2
        ▼ "preceding"
          ▼ "chain of events"
            ▼ "disjunction of events"
              ▸ APDU_installForLoad call
          ▼ "chain of events"
            ▼ "disjunction of events"
              ▸ APDU_load call
        "globally"
```

Fig. 6. The Temporal OCL editor (*GlobalPlatform* properties)

9 Conclusion

Although many temporal extensions of OCL exist, they have never been used convincingly in practice. To cope with this, we have presented a pattern-based extension of the OCL language to express temporal constraints on object-oriented

[4] www.commoncriteriaportal.org

systems. We defined our language with a formal scenario-based semantics to support the specification of test purposes and their automatic translation into regular expressions. We also developed this extension and integrated it into the Eclipse's OCL plug-in (version 4.×). As regards practical applications, our OCL extension is used in a test purpose framework dedicated to UML/OCL models in order to develop strategies to support the automatic testing of security properties on the smart card operating system *GlobalPlatform*.

Future work. As previously stated, adding temporal aspects to the OCL language could be a promising direction to explore model checking techniques. We intend to connect our language to usual model checking tools inspired by the work proposed by *Distefano* et al. in [12].

References

1. Projet TASCCC, Test Automatique basé sur des SCénarios et évaluation Critères Communs, `http://lifc.univ-fcomte.fr/TASCCC/`
2. Object Management Group. Object Constraint Language (February 2010), `http://www.omg.org/spec/OCL/2.2`
3. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st International Conference on Software Programming, pp. 411–420 (1999)
4. Jard, C., Jéron, T.: TGV: theory, principles and algorithms. In: World Conference on Integrated Design and Process Technology, IDPT 2002, California, USA (2002)
5. Gaston, C., Le Gall, P., Rapin, N., Touil, A.: Symbolic Execution Techniques for Test Purpose Definition. In: Uyar, M.Ü., Duale, A.Y., Fecko, M.A. (eds.) TestCom 2006. LNCS, vol. 3964, pp. 1–18. Springer, Heidelberg (2006)
6. Cabrera Castillos, K., Dadeau, F., Julliand, J., Taha, S.: Measuring Test Properties Coverage for Evaluating UML/OCL Model-Based Tests. In: Wolff, B., Zaïdi, F. (eds.) ICTSS 2011. LNCS, vol. 7019, pp. 32–47. Springer, Heidelberg (2011)
7. Baier, C., Katoen, J.P.: Principles of Model Checking. Representation and Mind Series. The MIT Press (2008)
8. Ziemann, P., Gogolla, M.: OCL Extended with Temporal Logic. In: Broy, M., Zamulin, A.V. (eds.) PSI 2003. LNCS, vol. 2890, pp. 351–357. Springer, Heidelberg (2004)
9. Richters, M., Gogolla, M.: OCL: Syntax, Semantics, and Tools. In: Clark, A., Warmer, J. (eds.) Object Modeling with the OCL. LNCS, vol. 2263, pp. 42–68. Springer, Heidelberg (2002)
10. Cengarle, M.V., Knapp, A.: Towards OCL/RT. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 390–409. Springer, Heidelberg (2002)
11. Calegari, D., Cengarle, M.V., Szasz, N.: UML 2.0 interactions with OCL/RT constraints. In: FDL, pp. 167–172 (2008)
12. Distefano, D., Katoen, J.P., Rensink, A.: On a temporal logic for object-based systems. In: Fourth International Conference on Formal Methods for Open Object-Based Distributed Systems IV, Norwell, MA, USA, pp. 305–325 (2000)
13. Mullins, J., Oarga, R.: Model Checking of Extended OCL Constraints on UML Models in SOCLe. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 59–75. Springer, Heidelberg (2007)
14. SOCLe Project, `http://www.polymtl.ca/crac/socle/index.html`

15. Flake, S., Mueller, W.: Formal semantics of static and temporal state-oriented OCL constraints. Software and Systems Modeling (SoSyM) 2, 186 (2003)
16. Bradfield, J., Filipe, J.K., Stevens, P.: Enriching OCL Using Observational Mu-Calculus. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 203–217. Springer, Heidelberg (2002)
17. Ramakrishnan, S., Mcgregor, J.: Extending OCL to support temporal operators. In: 21st International Conference on Software Engineering (ICSE 1999) Workshop on Testing Distributed Component-Based Systems, LA, May 16-22 (1999)
18. Conrad, S., Turowski, K.: Temporal OCL: Meeting specifications demands for business components. In: Unified Modeling Language: Systems Analysis, Design, and Development Issues, pp. 151–166. Idea Publishing Group (2001)
19. Object Managment Group. UML profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) (November 2009)
20. Specification patterns, `http://patterns.projects.cis.ksu.edu`
21. OCL temporal extension (2012), `http://wwwdi.supelec.fr/taha/temporalocl/`
22. OCL (MDT), `http://www.eclipse.org/modeling/mdt/?project=ocl`
23. Xtext 2.1, `http://www.eclipse.org/Xtext/`
24. Spec Patterns, `http://patterns.projects.cis.ksu.edu/`
25. Ledru, Y., du Bousquet, L., Maury, O., Bontron, P.: Filtering TOBIAS Combinatorial Test Suites. In: Wermelinger, M., Margaria-Steffen, T. (eds.) FASE 2004. LNCS, vol. 2984, pp. 281–294. Springer, Heidelberg (2004)
26. Tretmans, J.: Conformance testing with labelled transition systems: Implementation relations and test generation. Computer Networks and ISDN Systems 29(1), 49–79 (1996)

The Program Is the Model: Enabling Transformations@run.time

Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara

Universidad Autónoma de Madrid, Spain
{Jesus.Sanchez.Cuadrado,Esther.Guerra,Juan.deLara}@uam.es

Abstract. The increasing application of Model-Driven Engineering in a wide range of domains, in addition to pure code generation, raises the need to manipulate models at runtime, as part of regular programs. Moreover, certain kinds of programming tasks can be seen as model transformation tasks, and thus we could take advantage of model transformation technology in order to facilitate them.

In this paper we report on our works to bridge the gap between regular programming and model transformation by enabling the manipulation of Java APIs as models. Our approach is based on the specification of a mapping between a Java API (e.g., Swing) and a meta-model describing it. A model transformation definition is written against the API meta-model and we have built a compiler that generates the corresponding Java bytecode according to the mapping. We present several application scenarios and discuss the mapping between object-oriented meta-modelling and the Java object system. Our proposal has been validated by a prototype implementation which is also contributed.

Keywords: Model-Driven Engineering, Model Transformations, Transformations at Runtime, APIs, Java Virtual Machine.

1 Introduction

Model-Driven Engineering (MDE) is becoming a popular software development paradigm to automate development tasks via domain-specific languages (DSLs) and code generation. The application of MDE to more advanced scenarios is leading to a trend to use models at runtime as part of running systems [4]. The use of models at runtime requires interacting with functionality written in general-purpose languages (GPL), in particular made available in the form of APIs. However, there is a gap between model management frameworks and GPLs that overcomplicate this application scenario. Additionally, we have observed that certain kinds of programming tasks can be more naturally expressed using model transformation languages, rather than using GPLs like Java. Hence, we could take advantage of model transformation technology to facilitate them, but the lack of proper integration mechanisms hinders this possibility.

The most common approach to bridge the modeling technical space (also known as modelware) and existing programming APIs is writing ad-hoc

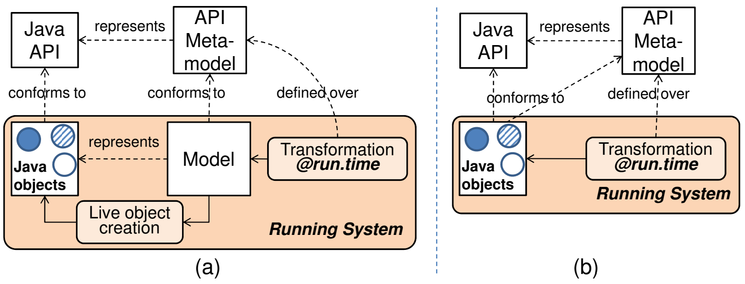


Fig. 1. (a) Transformation at runtime with an intermediate model representing the “live” objects. (b) Our approach, where runtime objects are seamlessly seen as models.

programs that map a given model to some API, using the facilities of the underlying meta-modeling framework (e.g., generated classes or reflective interfaces in EMF). Applications of this approach can be found in [5,17], and some automatic mapping tools have been proposed to facilitate this task [11,14,15]. Given a meta-model that represents the API, models are injected from live objects or objects are created from a given model. The main issue with these approaches is that models that imitate the API object structure must be created at runtime as well, together with the machinery to create or read the runtime objects from the models, which is inefficient and adds unnecessary complexity. This approach is depicted in Fig. 1(a).

Instead, in this paper we propose a more direct approach which provides a better integration of model transformations with programs written using GPLs (Java in particular), as depicted in Fig. 1(b). In particular, our approach permits model transformations to use directly Java objects as if they were part of a model, conformant to a meta-model. This takes the benefits of MDE to the program level, realizing some aspects of Bertrand Meyer’s *Single Product Principle: “The program is the model. The model is the program”* [13], and enabling a better integration of modeling and programming tasks.

Our approach is based on the specification of a mapping between a Java API (e.g., Swing) and a meta-model describing it. A model transformation definition is written against the API meta-model and then compiled into the corresponding Java bytecode according to the mapping. In this way, the integration of Java programs and model transformation technology is seamless. In this paper, we present several application scenarios and discuss the challenges involved in mapping object-oriented meta-modeling to the Java object system. Our proposal has been validated by a prototype implementation, which is also contributed.

Paper Organization. Section 2 gives an overview of our approach. Section 3 presents some background and introduces a running example. Section 4 details our approach to describe APIs by means of meta-models. Section 5 presents more sophisticated mapping constructs to support different API styles. Section 7 evaluates the approach on further examples in different scenarios. Section 8 compares with related work and Section 9 ends with the conclusions and future work.

2 Overview

In this section, we identify the requirements and challenges posed by the integration of modelware and object-oriented programming languages, and introduce our approach. We will focus on the integration from the perspective of model manipulations typically performed with model-to-model transformation languages, and using Java as the target GPL. For simplicity, we will consider the manipulation of object-oriented APIs, although the approach can be generalised to any kind of object-oriented program.

As explained in the introduction, several approaches exist to bridge the modelware technical space and GPLs, most notably Java [1,3,9,11]. However, a practical bridge should fulfil in addition the following requirements:

– *Non-intrusive.* It must not require modifying existing meta-models or existing APIs (e.g., using manually written annotations).
– *Seamless integration.* Once transformations are defined, they should be easily and seamlessly invoked from programs, as black-boxes, using regular constructs of the GPL. For instance, it should be possible to invoke a transformation by creating a new transformation object, setting the involved models and calling a *run* method. This aspect includes integration at the IDE level as well. And the other way round: model transformation developers should be able to deal with runtime objects as if they were model elements, described by a meta-model, using the model transformation language normally.
– *Efficient.* The bridge between Java objects and models should be efficient from both performance and memory usage point of views. In the ideal case, it should not require intermediate data structures, so that the model-based manipulations of Java objects become as efficient as if they were written using Java code. Such an intermediate representation would hinder some uses of transformations at runtime, like streaming transformations.
– *API style coverage.* APIs may provide access to objects in different ways, being the use of *getter* and *setter* methods the simplest one. A practical bridge should consider the most common access mechanism to cover a wider range of APIs.

Although some researchers have proposed solutions to bridge models and Java objects (cf. Section 8), to the best of our knowledge, no existing tool satisfies all the previous requirements.

2.1 Architecture

The elements of our approach are depicted in Figure 2. In general, a transformation definition, written in some transformation language, manipulates models that conform to some meta-models. We extend this pattern to allow a transformation definition to manipulate objects of a given API as if they were model

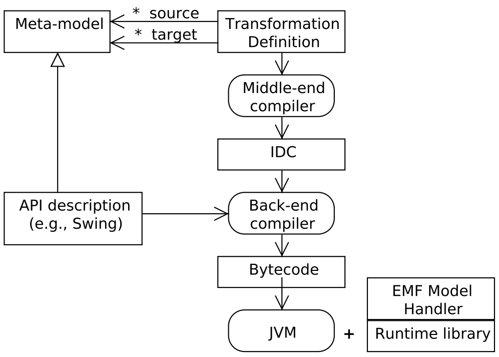


Fig. 2. Elements of our approach

elements. The underlying idea is to specify an API description model that, from the perspective of the transformation developer, acts as a meta-model for the API. This model establishes a mapping between the API and a set of meta-model elements that are used to write a transformation definition against them.

In our solution, the transformation definition is compiled to an intermediate language, called IDC (Intermediate Dependency Code), that provides primitive instructions for model manipulation (see next section for more details). This compilation step is performed without taking into account whether the transformation will deal with an API or a regular model. Then, the IDC intermediate representation is compiled to the Java Virtual Machine (JVM) bytecode format. At this step, the API description is used by the back-end compiler to generate bytecode to access Java objects directly (e.g., using method calls). Despite compiling a transformation definition to the JVM, our tool relies on the underlying meta-modeling framework when regular (meta-)models are used, with indirect access to model elements via a model handler (e.g., EMF Model Handler in the figure). Please note that this architecture is not specific to Java or the JVM, but the only requirements are that the underlying programming system be object oriented and that there is an interoperability mechanism so that the transformation language can manipulate the runtime objects. For instance, our approach could be adapted to other virtual machines (e.g., .NET) but also to a compiler pipeline, such as GCC, by generating compatible object code.

3 Background and Running Example

In this section we provide a running example that will be used throughout the paper. However, we first outline the technical context of our work, that is, the Eclectic transformation tool [7], the IDC intermediate language and the Java Virtual Machine (JVM).

Eclectic is a transformation tool based on the idea of a family of model transformation languages. Instead of having a large language with many constructs, we provide several small languages, each one of them focused on a specific kind

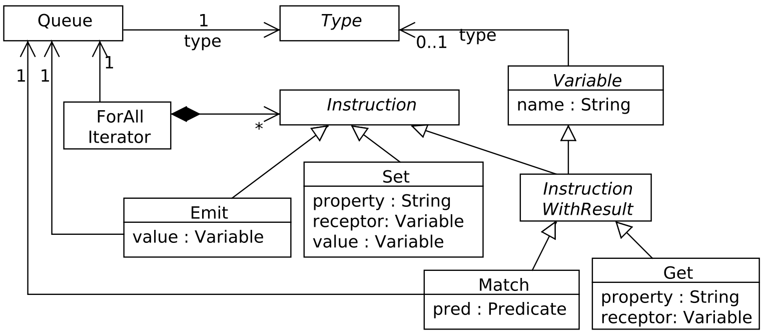


Fig. 3. Excerpt of the IDC meta-model

of transformation task. By now, the family is made of: (i) a unidirectional rule-based mapping language – in the style of the declarative part of ATL [12] – to specify correspondences between source and target model elements, (ii) a pattern language, (iii) a language for attribute computation, inspired by attribute grammars, (iv) a template-based, target-oriented language and (v) a language to orchestrate the execution of the different transformation tasks. Each language compiles down to the IDC intermediate language, which provides the composition and interoperability mechanisms. The translation from API descriptions to JVM bytecode is performed at the IDC level, so that every language of Eclectic can take advantage of the bridge. For the sake of simplicity, in this paper we will just use one language of Eclectic: the mapping language.

IDC is a simple, low-level language composed of a few instructions, some of them specialized for model manipulation. Figure 3 shows an excerpt of its meta-model. Every instruction inherits from the Instruction abstract metaclass. Since most instructions produce a result, they also inherit from Variable (via InstructionWithResult) so that the produced result can be referenced as a variable. Indeed, we use a simplified form of Static Single Assignment (SSA) to represent data dependencies between instructions [8], since every generated value is stored into a uniquely identified variable.

The IDC language provides instructions to create closures, invoke methods, create model elements and set and get properties (Set and Get in Figure 3), among others. In IDC, there is no notion of rule, but the language provides a more general mechanism based on queues. A Queue holds objects of some type, typically source model elements and trace links. The ForAllIterator receives notifications of new elements in a queue, and executes the corresponding instructions. There are two special instructions to deal with queues: Emit puts a new object into a queue, while Match retrieves an element of a queue that satisfies a given predicate. If such an element is not readily available, the execution of this piece of code is suspended into a *continuation* [6] until another part of the transformation provides the required value via an Emit.

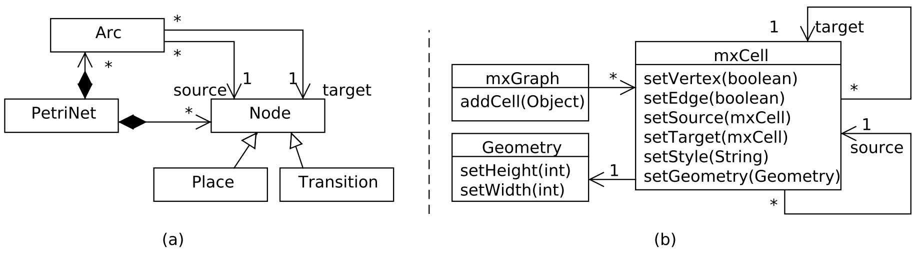


Fig. 4. (a) Petri nets meta-model. (b) A small excerpt of the jGraph API.

To some extent, IDC can be considered as an event-based approach to model transformation. If the transformation is executed in batch mode (i.e., all source model elements are readily available) then the transformation queues are just filled at the beginning and the transformation proceeds normally.

IDC transformations are compiled to the JVM. The JVM is a stack-based virtual machine based on instructions called bytecodes. Interestingly, the JVM instruction set is statically typed, since it requires detailed type information to perform operations over objects. For instance, calling a method requires specifying the name of the class or interface where the method was defined, the types of the parameters and the return type. We have taken this characteristic into account in the design and implementation of the bridge between meta-models and Java classes.

3.1 Running Example

As a motivating example, let us suppose we are using Petri net models conforming to the meta-model in Figure 4(a), and we are interested in visualizing such models for debugging purposes. A possible solution is to use an API like jGraph[1] which allows visualizing graph-like structures and provides automatic layout capabilities. However, if we address this task using plain Java and EMF, this would require writing an interpreter that imperatively traverses the model, keeps track of the cycles and instantiates the jGraph classes.

Instead, with our approach, we build a simple transformation that maps Petri net concepts (places, transitions and arcs) to jGraph API concepts. In particular, the excerpt of the jGraph API used in this transformation is shown in Figure 4. The mxCell class represents a visualizable element. Its setVertex and setEdge methods identify whether the cell acts as a vertex or as an edge. If it is an edge, the setSource and setTarget methods can be used to establish connections to other elements. A cell has an associated Geometry that establishes its size. This value is compulsory and can be set in the constructor or with the setGeometry method. For simplicity, we will not consider it until Section 4.3.

[1] http://jgraph.com

```
1  mapping petrinet2jgraph (in) −> (out)
2  in : 'platform:/resource/example/petrinet.ecore'
3  out : 'platform:/resource/example/jgraph.apidesc'
4
5  from p : in!PetriNet to g : out!Graph
6    g.cells <− p.nodes
7    g.cells <− p.arcs
8  end
9
10 from place : in!Place to cell : out!Cell
11   cell.vertex = true
12   cell.value = place.name
13 end
14
15 from t : in!Transition to cell : out!Cell
16   cell.vertex = true
17   cell.value = t.name
18 end
19
20 from arc : in!Arc to cell : out!Cell
21   cell.edge = true
22   cell.source <− arc.source
23   cell.target <− arc.target
24 end
```

Fig. 5. Transformation from Petri nets to jGraph with the Eclectic mapping language

Figure 5 shows the corresponding transformation, using the Eclectic mapping language. Rules establish a mapping between a source type and a target type. Line 1 declares the transformation name and parameters (input and output models), whose conforming meta-model is given in lines 2 and 3. Please note that for out we do not use a regular meta-model, but an API description that acts as a meta-model. Then, four transformation rules are defined translating each element of the Petri net into jGraph. The = operator assigns an attribute value and the ← operator resolves a reference.

This program declaratively specifies the transformation between two data structures: the Petri net model and the jGraph's representation of visualizable graphs. The transformation is defined as if there were models in the source and target domains; however, the transformation actually produces Java objects in the target. This is done by including an API description model in place of the target meta-model. The next section explains how to describe an API.

4 Mapping Meta-models to the Java Object Model

APIs in object-oriented languages are typically formed by a set of classes that represent the elements of some domain (e.g., widgets in a GUI toolkit). We propose to establish a mapping between API classes and a meta-model that describes the structure of the API. Several descriptions may be available for a given API, perhaps focussing on a different aspect. Describing an API as a meta-model permits the use of model transformation tools to manipulate the object graph of the API. An additional advantage is that the meta-model provides a compact description of the API that simplifies the access to its structure, since it does not contain behaviour methods (i.e., in contrast to get/set methods, which are related to the structure).

In this way, a key point of our approach is the mapping between meta-modeling concepts and object-oriented API concepts. To specify this mapping, we have built a meta-model called *API description*. API description models will drive the bytecode generation phase of our compiler.

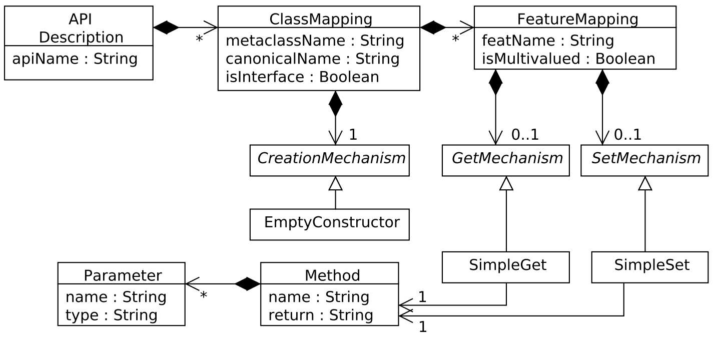


Fig. 6. Core elements of the API description meta-model

The rest of the section explains how APIs are described in our approach. First of all, we focus on the basic mappings that correspond to basic object-oriented constructs. Next, we introduce a DSL to specify these mappings. Finally, we deal with the mapping of constructors.

4.1 Basic Mapping

We have designed the mapping between a meta-model and an API from the perspective of the primitive operations over model elements that our model transformation engine requires. There are three kinds of operations: create elements, read features and write features (i.e., accessing features).

The simplest mapping is a one-to-one correspondence between meta-classes and API classes, and structural features (attributes for primitive types and references for classes) with getter and setter methods. The mapping for features must take into account whether the it is multivalued or not. Following the Java conventions, a mono-valued feature is mapped to a pair of accessor methods FeatureType get⟨⟨featureName⟩⟩() and void set⟨⟨featureName⟩⟩(FeatureType value). In the case of multi-valued features, the pair of accessor methods is Collection<FeatureType> get⟨⟨featureName⟩⟩() and void add⟨⟨featureName⟩⟩(FeatureType value).

However, an API may overlook these conventions, providing different access mechanisms. In [11], a fixed number of simple mappings is proposed. In our case, we do not restrict our API meta-model to some predefined mappings, but it has been designed with extensibility in mind so that new mappings can be added as they are discovered. Figure 6 shows the core meta-model.

An APIDescription is composed of ClassMapping elements, mapping a metaclass to a Java class (canonical name in the meta-model). Each feature of the metaclass must be mapped via a FeatureMapping. The mechanisms to access properties are abstracted by the GetMechanism and SetMechanism classes, which can be specialized with concrete mechanisms. One of such is the one-to-one correspondence explained above (through SimpleGet and SimpleSet).

The meta-model also includes detailed information about parameters and return types. This is needed in our case because the JVM specification requires the compiler to generate bytecode with this information. In any case, this information can be gathered via reflection to alleviate the user from this burden.

4.2 A DSL to Specify Mappings

We have built a textual DSL to facilitate the task of specifying mappings. The DSL allows specifying the meta-model that describes the API at the same time as the mapping to the API. There are four basic constructs:

1. metaclass maps a metaclass to the corresponding Java class or interface. The metaclass is automatically marked as abstract if it is mapped to an interface or an abstract Java class.
2. ref establishes a mapping for a reference (i.e., a feature whose type is a metaclass). Moreover, an access mechanism has to be specified. The simplest one is to associate a getter and a setter method. The * modifier indicates that the meta-model feature is multivalued. In the Java side, Array<JavaType> and CollectionType<JavaType> indicate that a method takes or returns several elements.
3. attr establishes a mapping for an attribute. It is similar to ref, but it deals with primitive types. We support the Java primitive types, and also java.lang.String as a primitive type. Automatic conversions between the meta-model type and the actual Java type are also supported (e.g., byte is converted to int).
4. constructor indicates how to create a new object. There is a straightforward mapping to the empty constructor if it is available. In the next subsection, we elaborate on how to establish mappings to non-empty constructors.

Figure 7 shows the mapping definition for the running example. As it can be seen, the Graph metaclass is mapped to the mxGraph Java class. The cells multivalued reference is mapped to the addCell method so that the generated transformation code will add cells one-by-one. This method takes a java.lang.Object as parameter, but from the transformation point of view, the object will always be a mxCell object (see next mapping from Cell to mxCell) . Finally, the feature is write-only because jGraph's API does not offer any getter method for cells. In Section 5 we will show an extension to overcome this limitation.

4.3 Mapping Constructors

In meta-modelling, metaclasses do not have constructors to ensure proper initialisation of objects, but the multiplicities assigned to features act as initialisation constraints. On the other hand, Java classes require at least one constructor, which does not necessarily need to be an empty constructor. In a model transformation, a model element (or a set of target elements that form a target pattern) is first created by a rule, and then initialised assigning values to features. Hence, there is a mismatch between model element instantiation and constructor-based instantiation of Java classes when an empty constructor has not been defined.

We have devised two extensions of the API description model to deal with this problem. The first extension permits associating literal values to the parameters

```
api jgraph described by "http://jgraph/api"
metaclass Graph to com.mxgraph.view.mxGraph {
  empty constructor
  ref cells∗ : Cell
      set method addCell(java.lang.Object) : java.lang.Object
      // no get method is defined −> readonly property
}

metaclass Cell to com.mxgraph.model.mxCell {
  empty constructor
  attr edge : Boolean
      get method getEdge() : boolean
      set method setEdge(boolean) : void
  attr vertex : Boolean
      get method getVertex() : boolean
      set method setVertex(boolean) : void
  ref source : Cell
      set method setSource(com.mxgraph.model.mxCell) : void
  ref target : Cell
      set method setTarget(com.mxgraph.model.mxCell) : void
}
```

Fig. 7. API description for jGraph

of a constructor. When the class is created, our engine uses the literal values as constructor parameters. This approach only works with constructors whose parameters can be primitive types or *null*.

The second extension provides greater flexibility by delaying instantiation until all parameter values are available. The underlying idea is to associate a constructor parameter to a feature of the meta-model, so that the corresponding object is not created until the value of all parameters are given by means of feature assignments (set instructions in IDC). Then, instead of setting the feature, the value is used as part of the constructor.

This process is transparent both to the transformation developer and to the transformation language developer, because the reordering of the instructions is made at the IDC level. As an example, even though jGraph's mxCell has an empty constructor, a better practice is to use another constructor that takes the cell value and a Geometry object as parameters. This ensures that cells are valid by construction. Figure 8 illustrates the rewriting process. The left-hand side shows the rule to transform Places extended to consider that a Geometry object has to be created as well (the linking construct establishes a link between both target elements). Below, the mapping from Cell to mxCell now includes a constructor statement that specifies which properties the constructor depends on.

Figure 8(b) shows the resulting IDC code. First, the target elements are created (instructions 1 and 2), and then, a new trace link is created and emitted to the trace (3-6). Afterwards, the features are set (we use intermediate variables v, lit1, lit2). It is important to note that feature assignments in the original transformation are translated to IDC Set instructions (9, 12, 13), despite the fact that they are actually constructor parameters.

Figure 8(c) shows the result of the rewriting. The Set instructions that correspond to constructor parameters are removed, but the assigned values will be used as constructor parameters. To this end, every instruction that is directly

```
from p : in!Place to
  to c : out!Cell, g : out!Geometry
  linking c.geometry = g

  c.value = p.value
  g.width = 20
  g.height = 20 [...]
end
________

metaclass Cell to mxCell
  constructor(value, geometry)
  ref geometry : Geometry
    constructor com[...].mxGeometry
    get method [...]
end
```

(a)

```
  forAll p : in!Place
1   c = new out!Cell
2   g = new out!Geometry
3   tlink = new trace!Link
4   tlink.s = p
5   tlink.t = c
6   emit tlink to Trace
7   c.geometry = g
8   v = p.value
9   c.value  = v
10  lit1 = 20
11  lit2 = 20
12  g.width  = lit1
13  g.height = lit2
  end
```

(b)

```
  forAll p : in!Place
10  lit1 = 20
11  lit2 = 20
 2  g = new out!Geometry(lit1, lit2)
 8  v = p.value
 1  c = new out!Cell(v, g)
 3  tlink = new trace!Link
 4  tlink.s = p
 5  tlink.t = c
 6  emit tlink to Trace
  end
```

(c)

Fig. 8. Translation and rewriting for constructors. (a) Transformation rule and API description. (b) Standard translation to IDC (in pseudocode). (c) Rewritten version that meets constructor dependencies.

or indirectly part of the computation of the value is moved before the creation instruction. If two or more target elements have mutually recursive constructors, it will result in a compiler error. Besides, it is worth noting that, at runtime, computing a value required by a constructor may require another transformation rule to produce it, so the scheduling of the transformation may become affected. In our case, we have the Match instruction to retrieve values from the transformation trace model. As explained in Section 3, this instruction is able to stop the computation of a rule if the value is not available, resuming the rule when another rule provides such a value. This mechanism allows performing this rewriting safely.

An important property of this strategy is that it is non-intrusive, in the sense that we do not need to change the transformation language adding constructors, but all the rewritings are performed at the IDC level.

5 Extending the Mapping

The presented mapping considers the basic elements needed to manipulate Java APIs with a model transformation language. However, it is limited to APIs that expose their structure via accessor methods. In this section, we present some extensions that we have added in order to cover a wider range of APIs. First, we will present an extension mechanism enabling flexible user-defined mappings. Then, we will present some extensions based on design patterns.

5.1 User-Defined Mappings

A simple extension is to consider mappings expressed using Java code. This permits specifying feature mappings that require accessing the API using some mechanism that is not supported by the API description language.

We have extended our meta-model and the DSL to consider get and set mechanisms implemented using Java code. In this extension, method calls are not performed over the original receptor object, but with an additional level of indirection. A "mapper class" is then implemented, that contains methods that are mapped to the meta-model features, in charge of getting or setting values for a given API object. Hence, the methods of this class contain API access logic that is not provided as simple method calls by the API, but must be manually written the API user.

Listing 1 shows a modified version of the API specification for the running example, which enables the retrieval of the cells of a graph. First, the mapper keyword selects the class implementing the methods that will perform the mapping (there is one mapper class per API description). Then, a syntax similar to the one for the getter methods is used, but replacing method by mapper. Listing 2 shows the piece of Java code that implements the mapping for the cells feature in the getCells method. A mapper class must implement IUserDefinedMapping and provide the setContext method. The latter is an initialization method to establish the transformation runtime information in case it is needed (e.g., access the source model to lookup some object).

```
api jgraph described by "http://jgraph/api"
mapper class example.JGraphMapping

metaclass Graph to com.mxgraph.view.mxGraph {
  empty constructor

  ref cells∗ : Cell
    get mapper getCells(com.mxgraph.view.mxGraph) :
      Array<com.mxgraph.model.mxCell>
    set method addCell(java.lang.Object) :
      java.lang.Object
}
```

Listing 1. User-defined mapping for "cells"

```
public class JGraphMapping implements
        IUserDefinedMapping {
  public mxCell[] getCells(mxGraph receptor) {
     mxCell root= ((mxCell) graph.getDefaultParent());
     mxCell[] cells= new mxCell[root.getChildCount()];
     for(int i = 0; i < root.getChildCount(); i++) {
       cells[i] = (mxCell) root.getChildAt(i);
     }
     return cells;
  }
  public void setContext(Context contex) { ... }
}
```

Listing 2. Mapper class for jGraph

5.2 Mappings Related to Design Patterns

In practice, object-oriented programs use a variety of techniques to provide greater flexibility to some aspects regarding construction of objects, structure or behaviour. Some of these techniques have been documented as design patterns [10]. We have studied which design patterns are relevant for our mapping, so that support for them can be provided extending the core meta-model.

We have identified six relevant patterns for our case: Abstract Factory, Singleton, Composite, Facade, Observer and Iterator. Our approach is to provide a description of how the pattern is instantiated in a given API, so that our compiler is able to generate the access code according to the description. Figure 9 shows the extension of the core mapping meta-model to describe the *Iterator* and *Observer* patterns. Currently, we have just given support to these two patterns, thus in the rest of the section we will focus on them. In any case, we expect that supporting the other four patterns will involve similar elements.

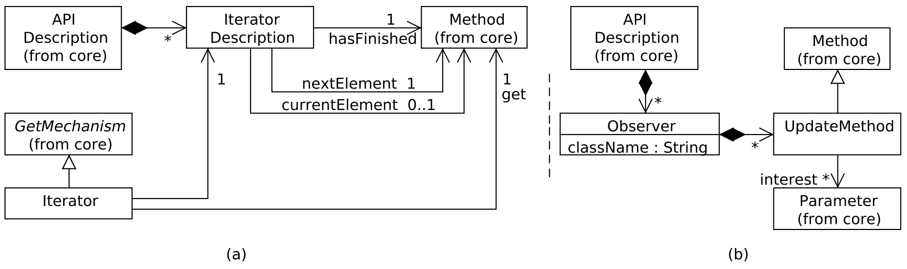


Fig. 9. Extensions to support design patterns. (a) Iterator. (b) Observer.

Iterator. An aggregated object provides access to its contents via another object that holds the iteration state. In our case, a reference may need to be filled from the elements of an iterator. Figure 9(a) shows how this pattern is represented in our meta-model. One or more IteratorDescription elements are declared, specifying the methods used to perform the iteration. If no currentElement is given, a Java-like iterator style is assumed (i.e., the nextElement method increments the iterator and returns the next element). Given this specification, the access to a multivalued feature can be mapped to an iterator. Our compiler generates the code to perform the iteration and retrieve the corresponding elements.

Observer. This pattern allows one or more subscribers to receive events of some type from a publisher. A transformation can be a subscriber (observer) so that a rule is triggered if its source pattern matches an incoming event. A transformation can behave as a publisher (observable) if it emits an event of some type each time a rule creates a new target element.

If a transformation uses an API that requires an observer, the transformation class automatically implements the required interfaces. The user of the transformation just needs to register the transformation in the corresponding observable. Figure 9(b) shows the extension to deal with observers. One or more Observer classes can be associated to an API. Each one of them contains a set of UpdateMethod elements that receive event objects as parameters. The event objects have to be described with the API description language as well, so that transformation rules can use them. The interest reference indicates which parameters of the update method must be passed to the transformation engine, discarding the rest. As explained in Section 3, at the IDC level, we use queues to feed the transformation engine. Thus, it is straightforward to fill IDC queues with the event objects received in the update methods. Effectively, by using this pattern we enable a form of streaming transformations.

6 Implementation and Integration

This section outlines some details of our implementation and the tool support.

6.1 Integration with Java Code

One distinctive aspect of our approach is that it seamlessly integrates with existing Java code. When a transformation definition is compiled, a class that acts as a front-end to configure and invoke the transformation is created. In this way, from the developer perspective, a transformation definition is just a Java class that performs a complex computation, taking some data structure and returning another one. For example, executing the running example will only involve writing a piece of code similar to the one shown in Listing 3. In constrast, other model transformation tools (e.g., ATL or ETL) provide dedicated launching mechanisms that are mainly intended to deal with EMF models.

```
public class Test {
  public static void main(String[] args) {
    petrinet2jgraph t = new petrinet2jgraph();
    EMFLoader loader = new EMFLoader();
    t.setInModel(loader.load("PetriNet.ecore", "model.xmi"));
    t.execute();

    mxGraph g = t.getOutModel().getRoot(mxGraph.class);
    // some code to launch the visualization
  }
}
```

Listing 3. Invoking a transformation compiled with Eclectic

6.2 Implementation and Tool Support

We have implemented the architecture shown in Figure 2. Transformation definitions are written with Eclectic, whose concrete syntax has been implemented using Xtext. The Eclectic compiler has been bootstrapped, so that internally it has the same architecture. Notably, we have used a subset of Eclectic to implement the middle-end compiler of Eclectic. The back-end compiler is written in Java, using the BCEL library[2] to generate bytecode. This step of the compilation deals with the translation of the IDC instructions, using the API description to generate the bytecode that access directly the Java objects.

Regarding tool support, we have built an Eclipse plugin that includes editors for the Eclectic languages and integrates the Eclectic compiler so that transformations are automatically re-compiled after a change. The *.class* files generated by the compilation process are added to the project classpath, so that they can readily be used in the Eclipse workspace. Finally, the binaries and the source code of our tool have been publicly released[3].

7 Case Studies and Assessment

We have evaluated our approach by implementing some case studies that exercise different kinds of APIs. Additionally, we have identified three main application scenarios for our approach, and we organise the case studies according to them.

[2] `http://jakarta.apache.org/bcel/`

[3] Eclectic web site: `http://sanchezcuadrado.es/projects/eclectic`

7.1 Application Scenarios

Model to Java. The first scenario consists of transforming a regular model, conforming to some meta-model, to Java objects that will be integrated with a running system. The running example of Section 3 belongs to this scenario. We have also implemented a simple transformation from UML class diagrams to Swing graphical user interfaces. Unlike a normal code generation approach, ours enables the dynamic generation of pieces of an application at runtime from a model, which ultimately permits runtime modifications just by changing the model and re-executing the transformation.

Java to model. The second scenario is the reverse situation. A set of Java live objects are processed by the transformation engine to yield a given model. We have identified two situations where this scenario could be particularly useful. The first one is reverse engineering at runtime, where the structure of a system at a given moment can be automatically analysed by means of a model transformation in order to discover certain information. This has the advantage that the source code of the application is not needed. We are implementing a case study where a transformation is used to reverse engineer a Swing GUI at runtime. A UML class model that represents the underlying design of the GUI is obtained, so that it can be compared against Swing best practices.

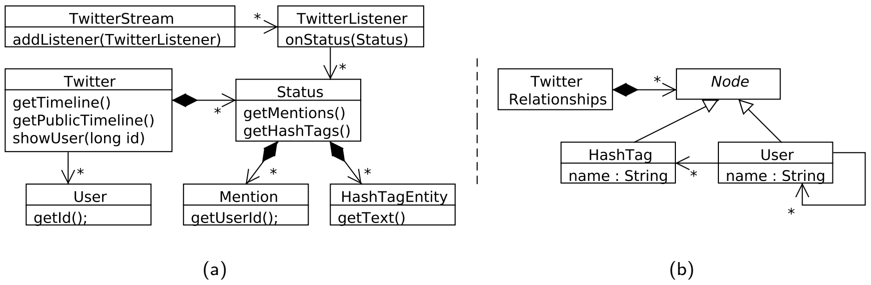
The second one is the possibility of taking advantage of existing APIs that perform some complex processing and generate an in-memory data structure that has to be further manipulated.

An important advantage of our approach is that there is no need to create a dedicated injector (i.e., a program that reads an artifact in the source technology and generates a model that can be manipulated using MDE technology). Instead, an existing API is directly used.

We have implemented two case studies of this kind. First, we have built a simplified version of the *Jar2Uml* case study[4]. In this application, the BCEL (Byte Code Engineering Library) APIis used to read the structure of a Jar file, which is then transformed into a UML class diagram.

The second case study involves using the *Twitter4J* API to read a stream of *tweets* from Twitter. A model-to-model transformation is in charge of discovering a graph of relationships between users emiting these tweets (and mentioned in them) and hashtags (i.e., keywords). Figure 10 shows (a) the Twitter4J API and (b) a simple meta-model to represent some relationships that arise in Twitter. The description of this API includes the Observer pattern to notify about events such as new tweets produced by users. Figure 10(c) shows the declaration of the TweetListener and the update method onStatus. The [0] modifier indicates that we are interested in the first parameter (as in general the *update* method could include more than one parameter). Our compiler automatically implements the update method and connects the received event to the rules "waiting" for values of this type. It is worth noting that this is a *streaming transformation*, which produces a target model, which gets continously updated as new events

[4] `http://ssel.vub.ac.be/ssel/research/mdd/jar2uml`

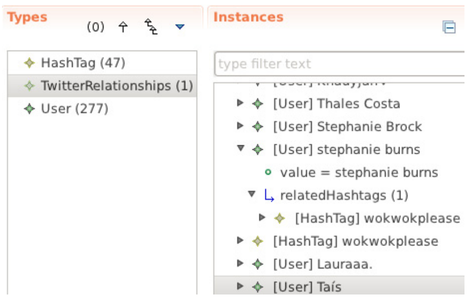


```
observer MyListener : twitter4j.TwitterListener {
  update [0] onStatus(twitter4j.Status) : void
  // other update methods ...
}

metaclass Tweet to twitter4j.Status {
  attr text : String get method getText() : String
  ref user: User get method getUser() : twitter4j.User
  ref hashtags*: HashTag
      get method getHashtagEntities() :
                 Array<twitter4j.HashTagEntity>
}

metaclass HashTag to twitter4j.HashtagEntity {
   attr text : String get method getText() : String
}
```

(c)


Types (0)
HashTag (47)
TwitterRelationships (1)
User (277)

Instances
type filter text
[User] Thales Costa
[User] Stephanie Brock
[User] stephanie burns
value = stephanie burns
relatedHashtags (1)
[HashTag] wokwokplease
[HashTag] wokwokplease
[User] Lauraaa.
[User] Taís

(d)

Fig. 10. (a) Twitter4J API. (b) Meta-model for relationships. (c) Excerpt of the JTwitter4J API description. (d) Resulting model.

arrive. This is possible because IDC features a scheduling mechanism based on continuations that allow us to stop a transformation rule until the required data is available, in this case associated to another event (a new tweet).

Pure Java transformations. In our experience there are several programming tasks that could be seen as a model transformation task. In many cases one needs to establish a mapping between semantically equivalent data structures. For instance, the Transfer Object J2EE pattern advocates creating POJOs (Plain Old Java Object) to transfer information between application tiers. The mappings between Business Objects and POJOs can be seen as a model transformation.

We have implemented two simple case studies of this scenario as a proof of concept. In the first one, the Java reflective API is used to access the information of classes of a given API, and we generate a PDF file (using the iText API[5]) that summarizes the methods and properties of each class.

In the second case study, we have implemented a transformation from ANT files to JGraph in order to visualize dependencies among build targets. Figure 11(a) shows an excerpt of the API description. The Iterator pattern is used in the ANT API to give access to the dependencies of a given target. We have specified the usage of this pattern in the API as explained in Section 5, by

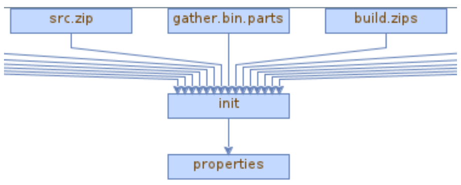
[5] http://itext.com

describing the iterator class and establishing which methods are used to perform the iteration. In this case, the API uses a java.util.Enumeration, and hence the model does not include a method to retrieve the current element, but only the method to retrieve the next one. Figure 11(b) shows the visualization obtained after executing the transformation for an ANT build file generated by Eclipse.

```
iterator Enumeration : java.util.Enumeration {
  finished hasMoreElements() : boolean
  next nextElement() : java.lang.Object
}

metaclass Target to org.apache.tools.ant.Target {
 attr name : String get method getName() : String
  ref dependencies∗ : Target
     get iterator Enumeration method
           getDependencies() : java.util.Enumeration
}
```

(a)



(b)

Fig. 11. (a) ANT API description (excerpt). (b) Dependencies in an Eclipse build file.

7.2 Assessment

The implementation of the case studies has shown that our approach provides a practical mechanism to integrate modelware and object-oriented programs.

The first scenario we tackle (*model to Java*) has been traditionally addressed by creating an ad-hoc processor that traverses a source model programatically an instantiates the API objects. Our approach simplifies this task in cases where there is a mapping between the source model and the API, since we can benefit from model-to-model transformation technology that is specialized for this task.

In the second scenario (*Java to model*), our approach permits leveraging existing APIs to facilitate obtaining an in-memory representation of complex artefacts, while model transformation technology is used to perform the analysis of such artefacts. Using the Observer pattern, we have shown that it is possible to apply this approach to construct *streaming transformations*, which listen to a stream of events and continuously update the target model.

The case studies for the third scenario (*pure Java transformations*) show that it is possible to apply model transformation technology to certain kinds of programming tasks that have a transformation nature, which otherwise would typically require writing a certain amount of boilerplate code. However, a model transformation language is a specialized language and therefore the programmer only needs to focus on the transformation task at hand, abstracting from accidental complexity. Hence, the development is facilitated and readability is improved. On the other hand, applying this approach may require from developers to learn a new language, and the cost-benefit of this trade-off has not been assessed yet.

With respect to performance, with our approach Java objects are in general dealt with as if they were manipulated with Java code, thus no overhead is expected. However, there are a few cases where our implementation still does not generate direct JVM calls, but reflection needs to be used. We aim at improving this part of the Eclectic compiler.

Finally, we found our API description DSL expressive enough in most cases. We added the user-defined mappings as a fallback, but we needed to use it in the case studies only three times. Our aim is to extend the DSL as we gain more insight about which idioms are used most frequently in APIs, in particular supporting the four design patterns mentioned in Section 5.

8 Related Work

Even though model-based development is increasingly used in software projects, there is scarce integration between model-based technologies and object-oriented programming. Next we review the few proposals we are aware of.

Api2MoL [11] is a tool to automate the process of bridging models and APIs. Like our approach, it provides a DSL to describe the mapping between a meta-model and an API. However, this DSL is limited to a small fixed number of basic mappings. It is implemented as an interpreter that acts as injector (to create a model from API objects) or extractor (to recreate the API objects from the model). Our API description model is more expresive than that of Api2MoL. In fact, it would be straightforward to implement Api2MoL with our tooling.

The Sm@rt project [14] aims at model-based runtime system management. A meta-model mirroring a system management API is created, and each meta-model element is associated a template defining the Java code that is in charge of performing the mapping. Then, a synchronization engine is automatically generated from this template-based specification. The Sm@rt project is specially tailored for management APIs, although other kinds of APIs can be supported by providing the Java code to perform the mapping.

CHART [9] is a graph transformation language that manipulates Java objects. It requires manual annotation of Java classes and methods to give them a graph-like structure. Besides, it enforces a particular style of the object-oriented programs since graph edges have to be represented with a Java interface.

Tom [3] is a term-rewriting language that has been piggybacked into Java. It uses algebraic terms as the underlying data structure, but is able to transform any data structure as long as a *formal anchor* is provided. Our API description model is indeed a formal anchor. Tom allows mapping algebraic terms to Java classes generated by EMF, but this requires providing some boilerplate code in the rewriting specification to take into account that we are dealing with a graph.

Table 1 summarizes the requirements that a practical bridge between models and objects should fulfil (cf. Section 2), and how they are handled (or not) by the aforementioned approaches. First, the approaches differ in the domain of application (synchronization engines, graph transformation, term rewriting or model transformation). Only Api2MoL, Sm@rt and Eclectic are non-intrusive. An important requirement is a seamless integration of the used transformation and programming languages, which is only fully achieved by Eclectic (CHART and Tom require generating the transformation as textual Java classes). At runtime, Api2MoL and Sm@rt use an intermediate model, which may affect the efficiency in certain scenarios. On the contrary, the other approaches use Java objects directly, with the constraint that Tom only supports tree-like structures. Finally, Tom and Eclectic provide flexible mechanisms to access objects, including also getter and setter methods.

Table 1. Comparison of approaches. SE : Synchronization Engines, GT : Graph Transformation (in-place), TR : Term Rewriting (for trees) and MT : Model Transformation.

	Domain	**Non-intrusive**	**IDE Integration**	**Runtime**	**API style**
Api2MoL	SE	Yes	No	Intermediate model	Getter/Setter
Sm@rt	SE	Yes	No	Intermediate model	Management
CHART	GT	No	Java text	Java objects	Getter/Setter
Tom	TR	Only for trees	Java text	Java objects (trees)	Flexible
Eclectic	MT	Yes	Bytecode	Java objects	Flexible

SiTra [1] is a simple approach to write model transformations in Java. It provides a Java interface that all implemented rules has to follow, and a *Transformer* class which executes the defined rules. Hence, this approach provides only a very light support for model transformations, which have to be encoded in Java, and there is no support to handle EMF models.

Other approaches based on Virtual Machines include the EMF Transformation Virtual Machine (EMFTVM) [16], or ATL [12], which provides dedicated Virtual Machines for model transformations. Our approach has the advantage of facilitating the integration of model transformation languages and GPLs based on the JVM. Also, the scheduling mechanism based on continuations of IDC is more flexible allowing e.g. streaming transformations.

Regarding API description languages, the approaches to discover API metamodels proposed in [11] and [15], as well the Framework Specific Modeling Languages (FSMLs) proposed in [2] are complementary to our work.

9 Conclusions and Future Work

In this paper, we have presented an approach for the seamless integration of model transformation and general-purpose programming languages, like Java. The approach enables the manipulation of Java objects as if they were modeling elements, in a transparent way. For this purpose, we use a mapping model, which describes both the meta-model against which the transformation is defined, and the mapping to the API to be used. The approach enables the use of transformations at runtime, and its seamless integration in programming projects.

In the future, we plan to extend Eclectic with further specialized languages. We also plan to provide a more complete support for streaming transformations and to provide means to model API protocols (method dependencies) which induce a certain scheduling of transformation rules, as well as looking into how to support API composition.

Acknowledgements. Work funded by the Spanish Ministry of Economy and Competitivity (TIN2011-24139), and the R&D programme of Madrid Region (S2009/TIC-1650). We thank the referees for their valuable comments.

References

1. Akehurst, D.H., Bordbar, B., Evans, M.J., Howells, W.G.J., McDonald-Maier, K.D.: SiTra: Simple Transformations in Java. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 351–364. Springer, Heidelberg (2006)
2. Antkiewicz, M., Czarnecki, K., Stephan, M.: Engineering of framework-specific modeling languages. IEEE Trans. Softw. Eng. 35(6), 795–824 (2009)
3. Bach, J.-C., Crégut, X., Moreau, P.-E., Pantel, M.: Model Transformations with Tom. In: LDTA 2012 (2012)
4. Blair, G., Bencomo, N., France, R.B.: Models@ run.time. IEEE Computer 42, 22–27 (2009)
5. Bruneliere, H., Cabot, J., Jouault, F., Madiot, F.: Modisco: a generic and extensible framework for model driven reverse engineering. In: ASE 2010, pp. 173–174. ACM, New York (2010)
6. Clinger, W.D., Hartheimer, A., Ost, E.: Implementation strategies for first-class continuations. Higher-Order and Symbolic Computation 12(1), 7–45 (1999)
7. Cuadrado, J.S.: Towards a Family of Model Transformation Languages. In: Hu, Z., de Lara, J. (eds.) ICMT 2012. LNCS, vol. 7307, pp. 176–191. Springer, Heidelberg (2012)
8. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. 13, 451–490 (1991)
9. de Mol, M., Rensink, A., Hunt, J.J.: Graph Transforming Java Data. In: de Lara, J., Zisman, A. (eds.) FASE 2012. LNCS, vol. 7212, pp. 209–223. Springer, Heidelberg (2012)
10. Gamma, E., Helm, R., Johnson, R., Vlissides, J.M.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley (1994)
11. Izquierdo, J.L.C., Jouault, F., Cabot, J., Molina, J.G.: Api2mol: Automating the building of bridges between apis and model-driven engineering. Information & Software Technology 54(3), 257–273 (2012)
12. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming 72(1-2), 31–39 (2008)
13. Meyer, B.: The Triumph of Objects (Invited Talk). In: TOOLS 2012 (2012)
14. Song, H., Huang, G., Chauvel, F., Xiong, Y., Hu, Z., Sun, Y., Mei, H.: Supporting runtime software architecture: A bidirectional-transformation-based approach. Journal of Systems and Software 84(5), 711–723 (2011)
15. Song, H., Huang, G., Xiong, Y., Chauvel, F., Sun, Y., Mei, H.: Inferring Metamodels for Runtime System Data from the Clients of Management APIs. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part II. LNCS, vol. 6395, pp. 168–182. Springer, Heidelberg (2010)
16. Wagelaar, D., Tisi, M., Cabot, J., Jouault, F.: Towards a General Composition Semantics for Rule-Based Model Transformation. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 623–637. Springer, Heidelberg (2011)
17. Wazaabi: An open source EMF based dynamic declarative UI framework, http://wazaabi.org/

A Framework for Bidirectional Model-to-Platform Transformations

Anthony Anjorin*, Karsten Saller, Sebastian Rose, and Andy Schürr

Technische Universität Darmstadt,
Real-Time Systems Lab,
D-64283 Merckstraße 25, Darmstadt, Germany
{anjorin,saller,rose,schuerr}@es.tu-darmstadt.de

Abstract. Model-Driven Engineering (MDE) has established itself as a viable means of coping with the increasing complexity of software systems. *Model-to-platform* transformations support the required abstraction process that is crucial for a model-driven approach and are, therefore, a central component in any MDE solution. Although there exist numerous strategies and mature tools for certain isolated subtasks or specific applications, a general *framework* for designing and structuring model-to-platform transformations, which consolidates different technologies in a flexible manner, is still missing, especially when *bidirectionality* is a requirement.

In this paper, we present: (1) An abstract, conceptual framework for designing and structuring bidirectional model-to-platform transformations, (2) a concrete instantiation of this framework using string grammars, tree grammars, and triple graph grammars, (3) a discussion of our framework based on a set of core requirements, and (4) a classification and detailed survey of alternative approaches.

Keywords: bidirectional model-to-platform transformations, string grammars, tree grammars, triple graph grammars.

1 Introduction

Model-Driven Engineering (MDE) has established itself as a viable means of coping with the increasing complexity of modern software systems by increasing productivity, supporting platform independence and interoperability, and reducing the gap between problem and solution domains [2].

Model transformations, in general, play a central role in any model-driven solution [2] and *model-to-platform* transformations, in particular, enable an abstraction from platform-specific details, which is usually an important first step in the Model-Driven Architecture (MDA) approach [2]. In this paper, a *platform* is defined as the final step in a given transformation chain and, in this context, includes textual files (XML files, configuration files, property files and code in a

* Supported by the 'Excellence Initiative' of the German Federal and State Governments and the Graduate School of Computational Engineering at TU Darmstadt.

programming language), folder and file hierarchies (structures in a filesystem), engineering tools with internal data structures that can be manipulated via an API, and very simple and typically generic tree-like structures. A *model* is an abstraction that is suitable for a particular purpose. We regard models as being conform to a *metamodel*, which is a representation of relevant concepts and relations in a domain usually specified with a standard modelling language such as UML[1], MOF[2] or Ecore [20].
Application areas of model-to-platform transformations include:

1. Round trip engineering involving code generation (forward engineering), and system comprehension (reverse engineering).
2. The development and evolution of Domain Specific Languages (DSLs).
3. The integration of different tools with tool-specific import/export formats.

As these applications are and always have been crucial tasks in software engineering, various approaches and tools already exist [8,9,19]. Current approaches are, however, either application specific (e.g., fixed metamodel), only handle an isolated subtask (e.g., only code generation), or are strongly tied to a certain technology or standard (e.g., Ecore). A general *framework* that can be used to design and structure model-to-platform transformations in a flexible manner is, therefore, still missing, especially when *bidirectionality*, crucial in many applications [5,12], is an important requirement. Such a framework must combine and consolidate state-of-the-art technologies in such a way that the strengths of individual components are emphasized and weaknesses are compensated, but still be general enough to allow a free choice and replacement of concrete standards or technologies. In this paper, we present:

1. An abstract, conceptual framework for structuring the required components of a bidirectional model-to-platform transformation.
2. A concrete instantiation of this framework based on string grammars and tree grammars using ANTLR [18], and Triple Graph Grammars (TGGs) [14] using eMoflon [1] and the Eclipse Modeling Framework (EMF) [20].
3. A discussion of our framework based on a set of core requirements.
4. A classification and detailed survey of alternative approaches.

The paper is structured as follows: Section 2 presents our running example, discusses further application domains for bidirectional model-to-platform transformations, and identifies a set of core requirements. Section 3 introduces an abstract conceptual framework, independent of any concrete technologies, together with a concrete instantiation thereof as a proof-of-concept. Section 4 classifies related approaches and compares them with our framework based on our requirements, while Sect. 5 concludes the paper with a summary and a brief overview of future work.

[1] Unified Modeling Language.
[2] Meta-Object Facility.

2 Application Domains and Core Requirements

Model-to-platform transformations are relevant in a multitude of application domains. In this section, we focus on application areas that additionally involve bidirectionality and derive a minimal set of *core* requirements.

Our running example is taken from the application domain *Round Trip Engineering* and is used consequently in the rest of the paper to introduce and explain all relevant concepts.

2.1 Running Example: Round Trip Engineering

Inspired by a real-world system modernization and re-engineering application scenario[3], our running example involves a software developer (Fig. 1::1[4]) who is trying to improve a software system that consists of a substantial number of source code files (Fig. 1::2). The developer has a set of *refactoring rules* (Fig. 1::3) which are to be automatically applied to the software system to result in an improved version (Fig. 1::4). For our concrete example, each source code file specifies a number of *components*, each of which can *require* other components.

To support system comprehension, e.g., in preparation of a re-engineering of the system, a *dependency analysis* of the complete system (possibly comprising thousands of components) is required. A suitable metamodel that captures the relevant concepts needed to express the dependencies in the system (Fig. 1::5) is established and a *platform-to-model* transformation (Fig. 1::6) is used to extract a dependency graph (Fig. 1::7) from the software system. The refactoring rules can now be expressed as a model transformation (Fig. 1::8) that can be applied to yield an improved dependency graph (Fig. 1::9). The final step is to update/re-generate the system with a *model-to-platform* transformation (Fig. 1::10).

To keep things simple, we restrict the analysis to only the component dependency graph and ignore the internal specification of each component. Although components can require components in different source files, we restrict the analysis to a single file for presentation purposes. The sample file depicted in Fig. 2 consists of four components `T`, `L`, `R` and `B`. The components form a dependency *diamond* as `B` requires `T` indirectly via `L` and `R`. Due to certain domain specific reasons (e.g, redundant memory allocation for the topmost component), our client wishes to avoid such diamond dependency subgraphs. The refactoring rule for our running example is as follows: If the component at the base of the diamond (`B`) is not required by any other components, it should be copied[5] and one of the dependencies (`R`) must be transferred to the copy (`B_Copy`) to break up the diamond. The refactored file according to this rule is depicted in Fig. 3.

A model-driven approach is advantageous as the metamodel is an abstraction, which can be chosen to be exactly suitable for the task, i.e., the refactoring

[3] Part of an industrial cooperation with Eckelmann AG (www.eckelmann.de)

[4] The notation Fig. n::m refers to label m in Fig. n.

[5] A further simplification as, in reality, an analysis of the *content* of the component is required to determine how it can be appropriately split into two independent parts.

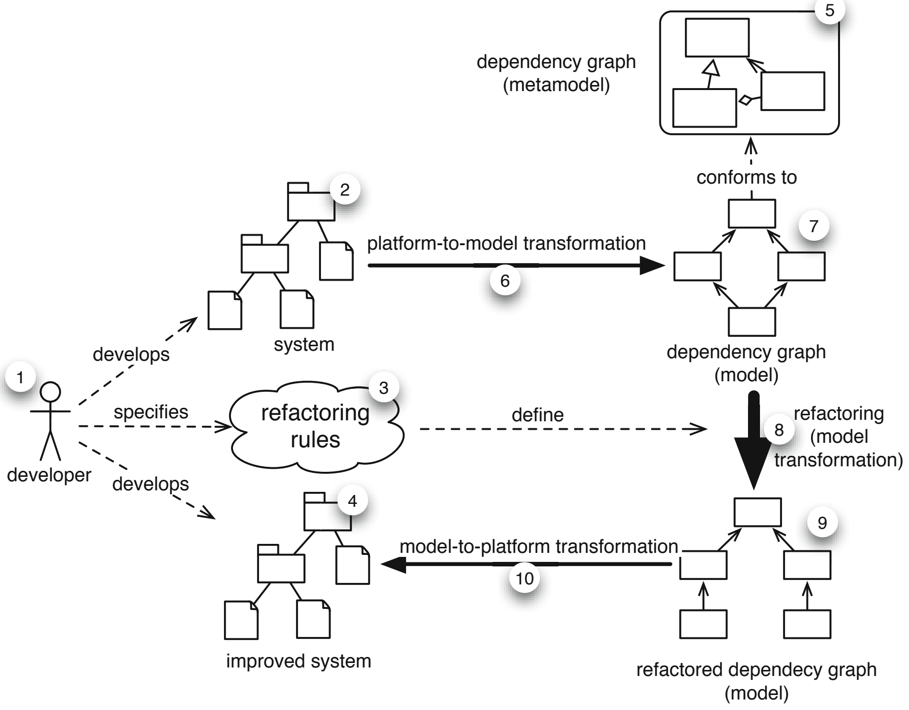

dependency graph (metamodel)
5
conforms to
2
platform-to-model transformation
6
7
system
dependency graph (model)
develops
1
specifies
refactoring rules
3
define
refactoring (model transformation)
8
developer
develops
4
9
model-to-platform transformation
10
improved system
refactored dependecy graph (model)

Fig. 1. Overview of the running example

```
Component T {
  // ...
}
Component L requires T {
  // ...
}
Component R requires T {
  // ...
}
Component B requires L R {
  // ...
}
```

Fig. 2. Before refactoring

```
Component T {
  // ...
}
Component L requires T {
  // ...
}
Component R requires T {
  // ...
}
Component B requires L {
  // ...
}
Component B_Copy requires R {
  // ...
}
```

Fig. 3. After refactoring

rules can be expressed as model transformations in a very concise, readable and maintainable manner. Furthermore, *bidirectionality* is a crucial requirement as the results of the model transformations must be reflected in the source code.

Our running example from the domain of round-trip engineering shows that bidirectional model-to-platform transformations are crucial to support the automation of repetitive, boring tasks (manually refactoring the source code files). In the following, we give a brief overview of other application domains which we use to derive a set of core requirements.

2.2 Further Application Domains

Round-trip engineering is not the only application domain where bidirectional model-to-platform transformations are necessary. From numerous examples we choose two further domains and give a schematic representation of the workflow and requirements for bidirectionality in each case.

DSL Development and Evolution: Domain Specific Languages (DSLs) are programming languages of limited expressiveness, which are designed to be maximally suitable for a particular task or domain [8]. The systematic development of such languages to increase productivity and improve communication between domain experts and professional software developers is a major application of MDE technologies in general, and model transformations in particular.

Models that conform to a certain metamodel (Fig. 4:1) can be specified in *abstract syntax*, i.e., as typed attributed graphs with respect to the metamodel (the type graph). For domain experts who wish to create models in the DSL, a *textual concrete syntax* is a more suitable means of specifying models, and supporting this requires at least a unidirectional text-to-model transformation (Fig. 4:2).

There are two reasons why bidirectionality is an important requirement in this context: Firstly, there might be a different group of domain experts who prefer to use some other kind of concrete syntax (possibly visual) to specify models of the same DSL (Fig. 4:3). To exchange models freely between the different groups of experts, it must be possible to transform back and forth, which requires a bidirectional model-to-platform transformation for each supported concrete syntax of the DSL. Secondly, a DSL *will* evolve over time to accommodate new or changed requirements and this can be supported via a corresponding model transformation from the old version of the metamodel to a new one (Fig. 4:4). In a real-world scenario, all models specified in the old version of the DSL (Fig. 4:2), *must* be transformed to be conform to the new metamodel and a possibly new version of the textual concrete syntax (Fig. 4:5). Such DSL evolution support clearly requires a bidirectional model-to-platform transformation for each version of the DSL.

Tool Integration: Engineering processes typically involve different stakeholders who work together to build or maintain a system. Each stakeholder has a specialized viewpoint or specific needs regarding the complete engineering process and uses established engineering tools in the corresponding (sub)domain.

An efficient exchange of data and information between the tools in use, i.e., *tool integration* [21], avoids redundancy and ensures consistency across tool borders.

A schematic tool integration setup is depicted in Fig. 5. An engineer in a certain domain works with his preferred engineering tool A (Fig. 5::1). To exchange data with another engineer using a different tool B (Fig. 5::7), the relevant data for the integration must be extracted from the tools. Engineering tools are very often *commercial off-the-shelf* tools and might only offer a tool-specific import/-export format (Fig. 5::2), very often XML [15]. In a tool integration scenario, a *tool adapter* (Fig. 5::3) is required to extract a suitable model (Fig. 5::4) from the available textual exchange format via a bidirectional model-to-platform transformation. The extracted model can then be synchronized (Fig. 5::5) with different but related models from other tools (Fig. 5::6), updating the data in the tools via respective tool adapters. Bidirectionality is thus an important requirement.

2.3 Core Requirements

After highlighting important application domains for bidirectional model-to-platform transformations, we derive a core set of requirements in this section, divided into two main groups: (1) Requirements concerning the resulting bidirectional model-to-platform transformation that is to be implemented, and (2) requirements concerning the *process* of establishing such a transformation.

Requirements Concerning the Resulting System: All requirements are formulated for a "correct" system, i.e., we assume that the transformation must perform as specified, e.g., by a testsuite, before requirements are considered.

(R1) Maintainability: The most important requirement is that the complete transformation be maintainable. This means that it should be relatively easy to extend, improve or otherwise adapt the transformation for all participants and stakeholders. This implies a number of sub-requirements including:

- *Support for bidirectionality* to ensure that changes in requirements can be reflected in the system without introducing inconsistencies,
- *Readability* to support communication and knowledge transfer amongst developers and to allow for a validation by domain experts who might not be professional software engineers and must at least understand parts of the transformation,
- *Stability* allowing subsystems to be exchanged as required without causing ripple-effects in others.

(R2) Scalability: Depending on the exact limits posed by the application domain and concrete scenario, the transformation must scale with respect to memory consumption and runtime complexity. If possible, this should be *guaranteed* by the applied approach, e.g., polynomial runtime.

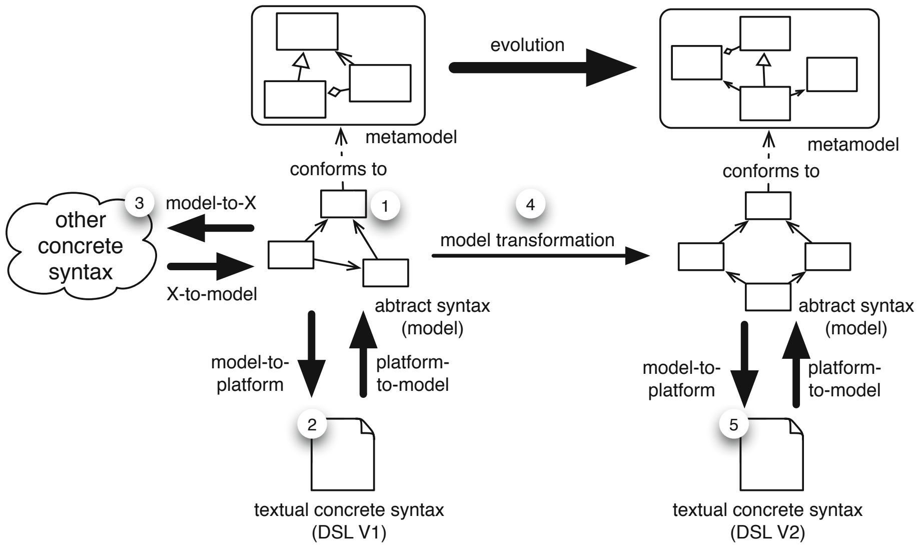


Fig. 4. DSL Development and Evolution

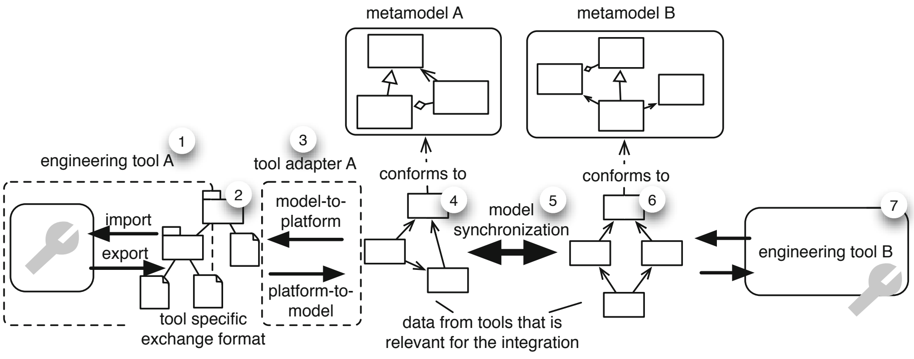


Fig. 5. Tool Integration

Requirements Concerning the Process: For the actual process involved in establishing a bidirectional model-to-platform transformation, we consider the following two requirements.

(R3) Productivity: The most important requirement concerning the development process is productivity, i.e., the speed of development which implies *usability*, adequate *tool support*, the possibility to *iteratively* develop and improve the system, and support for testing (static analysis for validation, debugging).

(R4) Generality: To handle real-world applications, the approach must be *general* enough. This means that the restrictions posed by the approach should not be too

limiting, i.e., should not restrict the class of possible applications to such an extent that the approach becomes useless for most practical purposes. This requirement has two implications: The approach must offer (i) a flexible and well-defined *fallback* to a turing complete language for situations where restrictions cannot be met, and (ii) a *uniform* treatment of a wide range of platforms including XML and other textual formats, directory structures, and generic data structures.

3 The Moflon Code Adapter (MOCA) Framework

Figure 6 depicts a framework for organizing the components necessary for a bidirectional model-to-platform transformation. This framework is abstract in the sense that it does not prescribe any concrete technologies or modelling standards. The main idea of our approach is to separate the transformation into two distinct parts: (i) A platform-to-tree transformation and (ii) a tree-to-model transformation.

The *platform* (Fig. 6::1) is transformed via a *parser* (Fig. 6::2) to a simple tree structure (Fig. 6::3). This tree structure should be a minimal abstraction of the platform which is nonetheless accessible to the chosen bidirectional transformation language (Fig. 6::4). Trees are transformed back to the platform via an *unparser* (Fig. 6::2) which typically linearizes the tree structure and adds platform details which were abstracted away by the parser if necessary. A crucial point is to keep the parser and unparser *as simple as possible*. In our opinion, this first step is often not worth supporting with a bidirectional language[6], and,

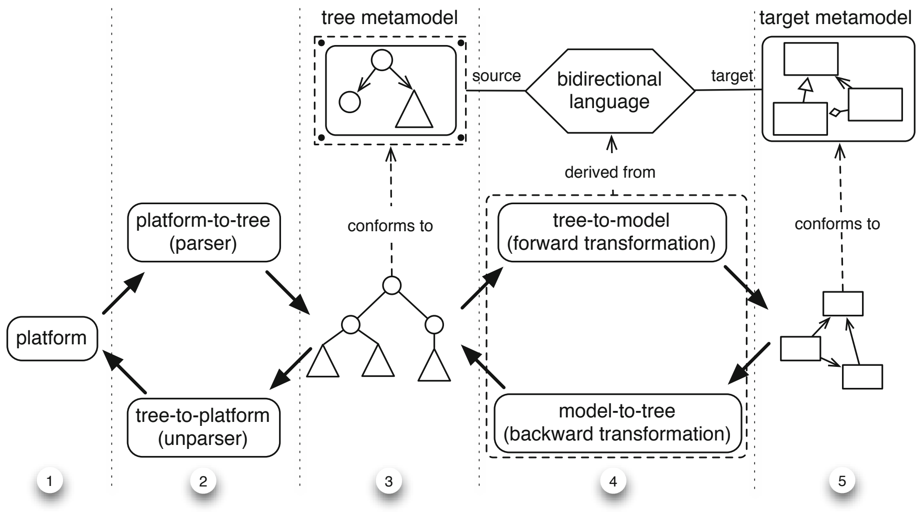

tree metamodel
target metamodel
source
bidirectional language
target
derived from
conforms to
conforms to
platform-to-tree (parser)
tree-to-model (forward transformation)
platform
tree-to-platform (unparser)
model-to-tree (backward transformation)
1
2
3
4
5

Fig. 6. The Moflon Code Adapter (MOCA) Framework

[6] This is not always true, e.g., if the application scenario requires layout preservation.

if it is kept to a bare minimum, almost all complexity can be shifted to the model-to-tree transformation, which can be appropriately handled with a suitable bidirectional transformation language (Fig. 6:4). In an MDE context, the tree should be a very simple structure which is nonetheless already conform to the modelling standard as required by the bidirectional transformation language and the target model (Fig. 6:5). As almost all standard parsers are context-free, "very simple" usually means acyclic and homogeneous with respect to typing (i.e., very few or even only a single "node" type is used).

The process prescribed by the framework already has a number of advantages:

Separation of Concerns: A strict separation of platform *comprehension* and *generation* (Fig. 6:2) from the actual *transformation* (Fig. 6:4) positively affects maintainability (R1) and productivity (R3) as the (un)parser can be kept very simple and be replaced without having to change the transformation. Furthermore, the bidirectional language can operate on a tree structure without irrelevant details of the textual representation leading to a simpler transformation with the clear task of (i) adding appropriate typing information and (ii) deducing context-sensitive relations to obtain the target model (Fig. 6:5).

A Clear Interface to Different (un)parser Technologies: Establishing a simple tree structure for the bidirectional transformation consolidates XML and different abstract syntax trees produced by parsers. Even the directory and file structure can be embedded in the tree structure if it is relevant for the transformation. This positively affects the generality (R4) of the approach.

Demanding only a semi-structured, i.e, hierarchical structure, greatly simplifies the task of parsing and unparsing and clearly places most of the complexity in the transformation, which can be supported with a bidirectional language. This applies the right tool for the right job and also allows for using standard parser and unparser technology via simple adapters, which can be easily replaced. This positively affects maintainability (R1) and productivity (R3).

Modularity: In general, the modular structure of the framework enables a high level of reuse and exchangeability of the platform, parser and/or unparser, the model-to-tree transformation, the target metamodel and the modelling standard without affecting all other components. This positively affects maintainability (R1) as components are stable, productivity (R3) due to possible reuse, e.g., of existing (un)parsers, and generality (R4), as at least parts of the system can be ported to a different platform or standard.

3.1 An Implementation of MOCA in eMoflon

As a proof-of-concept, the MOCA framework has been realized as part of our metamodelling tool eMoflon [1] and can be downloaded and used as described in our detailed tutorial[7]. Our MOCA implementation is currently in use for

[7] Available from `www.moflon.org`

a lecture at our university, and in two ongoing projects handling real world applications[8] from the industry.

Figure 7 depicts the concrete instantiation of the abstract MOCA framework as realized in eMoflon. The supported platform (Fig. 7::1) is a directory structure, which can contain files with different textual content as indicated by the shading in the diagram. To complement a built-in directory "parser", the user must provide a parser (Fig. 6::2) for each type of file, to produce an abstract syntax tree which is inserted as a shaded subtree into the resulting tree (Fig. 6::3). Our MOCA implementation provides dedicated support for XML via an adapter layer, i.e., arbitrary XML files can be automatically transformed to instances of our tree metamodel (`MocaTree`). To handle arbitrary textual formats, we provide support for parsers and unparsers generated with ANTLR [18] via *string grammars* and *tree grammars* with templates [19], respectively.

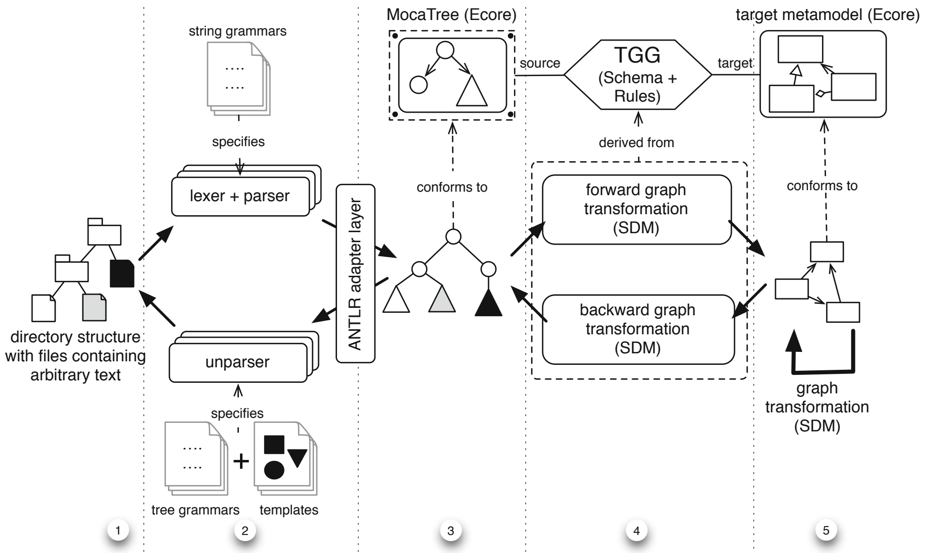


Fig. 7. A realization of the MOCA Framework as part of eMoflon

The textual format for our running example (Fig. 2 and 3) is non-standard and requires a parser. Using ANTLR, this involves specifying a lexer and parser as depicted for the example in Fig. 8. Please note how the parser builds up an abstract syntax tree indicated in the textual syntax as `^(ROOT CHILDREN)`. Using our ANTLR MOCA adapter, the abstract syntax tree produced by the parser can be directly loaded as an EMF model without any further effort. Please note that the lexer and parser *do absolutely nothing else* apart from recognizing the textual content and building a simple, homogeneous hierarchical structure.

[8] A bidirectional RTF to HTML transformation and a common DSL for consolidating iOS and Android app development.

```
COMPONENT:  'Component';
SPEC:       'SPEC';
BODY :      '{' .* '}'
REQUIRES:   'requires';
ID:         ('a'..'z'| 'A'..'Z')+;
WHITESPACE: ('\t' | ' ' | '\r' | '\n')+
```

(a) Lexer Grammar

```
main: componentSpec+
  -> ^(SPEC componentSpec+);
componentSpec: COMPONENT ID dep? BODY
  -> ^(ID ^(REQUIRES dep?) BODY);
dep: REQUIRES reqs+=ID+
  -> $reqs+;
```

(b) Parser Grammar

Fig. 8. Lexer and Parser Grammars

For code generation, the context-free nature of the tree is exploited using a *tree grammar*, which traverses the structure of the tree in a depth-first manner and evaluates a set of templates to produce text (Fig. 9). We use *StringTemplate* [19] as a template language, which is a very restricted, extremely simple template language with a minimal set of commands. Enforcing such simple templates leads to a strict model-view separation with various advantages [17]. These four simple rule-based specifications (Fig. 8, Fig. 9) implement the first step in the framework, the platform-to-tree transformation (Fig. 7::2), for our example.

```
main:  ^('SPEC'
         content+=component*)
  -> file(content={$content});

component: ^(name=STRING
             r+=reqs
             body=STRING)
  -> component(name={$name},
               r={$r},
               body={$body});

reqs: ^('REQUIRES'
        l+=STRING*)
  -> reqs(l={$l});
```

(a) Tree Grammar

```
file(content) ::= <<
<content; separator="\n\n">
>>

component(name, r, body) ::= <<
Component <name> <r>{<body>}
>>

reqs(l) ::= <<
<if(l)>
requires <l; separator=" ">
<endif>
>>
```

(b) Templates

Fig. 9. Tree grammar and templates

In our MOCA implementation, we use *Triple Graph Grammars* (TGGs)[14] as the bidirectional language to transform the context-free, homogeneous "tree" (Fig. 7::3) to the target model (Fig. 7::5). eMoflon is EMF/Ecore based and thus, the modelling standard used is EMF. Figure 10 depicts the *TGG Schema* for the running example, which is a triple of the metamodels involved in the transformation. To the left, the tree consists of files and nodes, while our target model, to the right, consists of "components" which can require other components and are all contained in a specification. Our complete tree metamodel

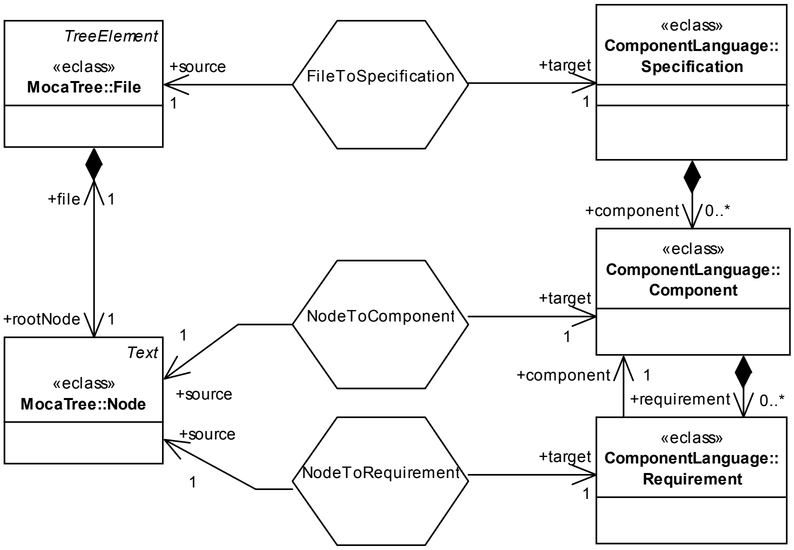


Fig. 10. TGG Schema (Triple of involved metamodels)

is just complex enough to represent XML files, directory structures and parse trees without losing information, i.e., we have basic concepts of folders, files, and nodes with attributes and labels. In the middle, *correspondence* types are defined, which are used for traceability. Already from the schema, it is clear that each file corresponds to a specification, and that both components and requirements correspond to nodes in the tree.

A TGG specification consists of a schema and a set of rules that describe the simultaneous evolution of triples of connected source, correspondence and target models. The advantage of using TGGs is that both forward and backward transformations can be automatically derived from this single specification and are guaranteed to be compatible with the described simultaneous evolution.

In sum, the required transformation for our running example consists of three TGG rules, one to transform files and specifications, one to handle components and one to create the requirement relation between components. The latter is depicted in Fig. 11. Please note that the hexagonal shape of correspondence types in the TGG schema and correspondences in the TGG rule is just syntactic sugar to indicate at a glance that these objects belong to the correspondence domain, i.e., can be interpreted as traceability types and links, respectively.

A TGG rule consists of context elements, depicted in black without any stereotype, and create elements depicted in green with an additional `create` stereotype. Context elements must be created by other rules and are used to induce an implicit dependency between rules, e.g., the TGG rule `NodeToRe-quirement` can only be applied if the two components involved (`component` and `reqComponent`) have already been created and identified with nodes in the tree by other rules (Fig. 11). The rule creates a requirement between `component` and

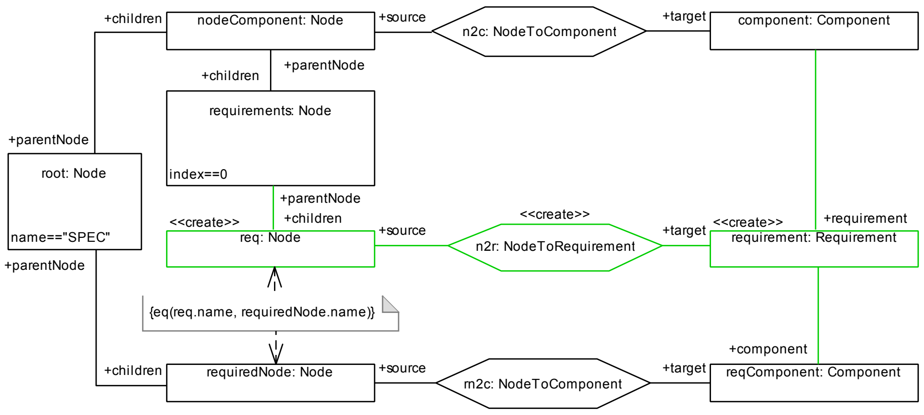


Fig. 11. TGG Rule NodeToRequirement

`reqComponent`, reflecting this in the tree by creating a new requirements node `req` for `nodeComponent`, with its name equal to that of `requiredNode`. This condition is expressed using the *attribute constraint* `eq(req.name, requiredNode.name)`. The TGG rule `NodeToRequirement` showcases the two main tasks of the bidirectional transformation: (i) Introducing appropriate typing, e.g., a `Requirement` instead of just a `Node`, and (ii) replacing the context-free acyclic tree with a context-sensitive graph structure, e.g., connecting two components via a requirement directly instead of having separate nodes with the same name.

To complete our running example, we can now specify the refactoring rules using an appropriate transformation language that can operate on the target model. Figure 12 depicts a graph transformation rule using *Story Driven Modelling* (SDM)[7], our graph transformation language for unidirectional model transformations in eMoflon. The rule is declarative and concise, and the diamond structure to be found can actually be "seen". According to the refactoring rule (Fig. 12), the dependency diamond should only be resolved if `component` is not required by any other components. This is enforced using a *negative application condition* (NAC) depicted by the crossed out element `otherReq`.

If the diamond structure is found and the NAC is not violated, the requirement `req2` is relocated to a new component `newComponent`, created as a copy of `component`. Please note the stereotype `destroy/create` used to indicate elements that should be deleted/created[9], the *attribute assignments* used to initialize the attribute values in `newComponent`, and the fixed (bound) elements starting from which the other elements must be found, indicated with a thicker border (`component` and `this`).

After applying the rule to all components, the refactored model can be transformed back to a tree and used to generate the refactored textual file as required (Fig. 3). After a backward or forward transformation, the created triple

[9] Additionally indicated via the red/green colour of the elements.

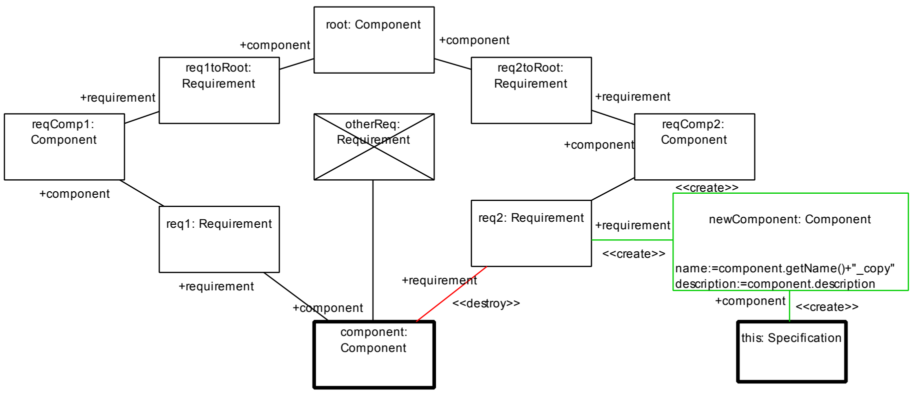


Fig. 12. SDM refactoring rule

of source, correspondence and target models can be visualized in eMoflon using our *integrator* which represents the correspondence model visually as links.[10]

In addition to the advantages of the abstract MOCA framework, our concrete choice of languages and standards has the following advantages:

Homogeneity via Complementary Languages: Our choice of languages, i.e., string grammars, tree grammars combined with templates with minimal logic, triple graph grammars and SDMs are all *rule-based* and *declarative*[11]. Based on our experience of working with this mix of languages, we believe that a high level of homogeneity is attained by supporting a common rule-based thinking in *patterns*. This positively affects maintainability (R1) and productivity (R3) as there is no disturbing shift in paradigm and trained skills in one language can be transferred to all others.

Formal Properties and Guarantees: By separating the transformation into different steps, the formal properties of the different individual languages still hold for the corresponding step. For example, the runtime complexity for *LL(*)* parsing (worstcase $O(n^2)$, in practice actually much less [18]) holds for tree generation, while TGGs guarantee polynomial runtime [14] for the tree-to-model transformation. Depending on the application scenario, such runtime efficiency can be crucial for scalability (R2). Furthermore, TGGs also guarantee that the derived transformations are *correct* with respect to the specified TGG, i.e., only a single specification is used, which positively affects maintainability (R1).

Flexible Fallback to Java: In our experience, practical problems can almost never be *completely* solved with a DSL. For example, even if a large part of a transformation can be specified with TGGs, certain parts, especially low-level

[10] Please refer to our tutorial (`www.moflon.org`) for screenshots and further details.

[11] SDMs are *programmed* graph transformations and, therefore, also have usual imperative language constructs such as if-else and loops.

attribute manipulation, must be specified using SDMs or directly in Java. All languages used in our MOCA implementation support a *fallback* to a more general language when necessary: ANTLR offers *syntactic* and *semantic predicates* in string and tree grammars [18], which can be used to embed Java statements to support the lexer/parser, SDMs offer *MethodCallExpressions* [1] to mix Java code in graph transformations in a type safe manner, and TGGs offer *Attribute-Constraints*, which are implemented in Java. ANTLR and eMoflon are both completely generative, i.e, map specifications to standard Java code which also simplifies mixing in hand-written code. This positively affects generality (R4), as basically any problem can be tackled that could also have been solved directly in a general purpose language, in our case Java.

Iterative Workflow: Last but not least, an iterative workflow is possible as the target metamodel can be iteratively refined. In each step, more parts of the tree can be handled by TGG rules until the transformation is complete. The platform can also be handled in an iterative manner, e.g., by using regular expressions to “filter” the textual files in the first iterations instead of a parser. ANTLR also supports this with a “fuzzy” parsing mode that ignores all content that cannot be parsed, i.e., parses these parts as a string block without further processing/structuring. An iterative workflow improves productivity (R3) as most mistakes can be found early enough in the development process.

3.2 Limitations and Drawbacks

Every approach has limitations and in the following, we discuss the most important drawbacks of our framework and its concrete implementation in eMoflon:

A Steep Learning Curve: Separating the transformation into different steps that are implemented with different languages has the potential of increasing complexity in general. Although we have tried to choose complementary languages with a common paradigm, it is still challenging to master all the different languages, especially without prior experience with rule-based languages.

Requires a Model-to-Model Transformation: Requiring a model-to-tree transformation as a separate step introduces an extra transformation language (TGGs) and tool dependency in the transformation chain.

Incrementality: The separation in different parts advocated by our framework makes the task of supporting incrementality for the *complete* transformation chain more challenging than if all steps were merged in a single specification.

4 Related Work

In this section, we discuss the main groups of alternative approaches to our generalized tree-based approach and highlight main strengths and weaknesses. We do not try to give a complete list of concrete tools but rather focus on *groups* of approaches, mentioning a few concrete representatives in each case.

4.1 Combination of Unidirectional Approaches

Although there is an increasing number of bidirectional languages available, the standard way of implementing bidirectional model-to-platform transformations is still to use two unidirectional transformation languages, one for each direction. Typical combinations include *Xtext* [6] for platform-to-model and *Xpand*[12] for model-to-platform, or *ANTLR* for platform-to-model and *Velocity*[13] for model-to-platform. The main advantage is clear; A combination of standard, mature unidirectional approaches is very general (R4) and "gets the job done" while existing bidirectional approaches are mostly still in development and are often not usable for real-world application scenarios, although they might work very well for a restrained class of problems. The flexible combination and the possibility of implementing parts of the transformation in standard languages offered by our framework is, in this respect, a pragmatic approach to having the best of both worlds, i.e., still profiting from the advantages of a bidirectional language. Similarly, standard approaches typically scale well (R2) with respect to runtime and memory consumption. A challenge, however, is handling *incremental* changes which becomes difficult when separate tools are used for each direction. Scalability then becomes a major problem if the scenario involves a *synchronization* in contrast to a *batch transformation*.

A further disadvantage of a combination of unidirectional approaches is that it is hard to maintain (R1): Changes to the forward transformation have to be carefully reflected in the backward transformation and vice-versa, and this gets increasingly difficult with the complexity of the transformation. Productivity (R3) also suffers as two separate specifications have to be implemented. A bidirectional language would be advantageous in both cases.

4.2 Tightly Integrated Software Development Environments

A second group of approaches are tightly integrated software development environments that provide *view-based*, *syntax directed* editing, keeping the concrete and abstract syntax of models synchronized at all times. This means that the editor operates directly on the abstract syntax of a model and reflects changes immediately in the presented concrete syntax (the view). Examples for such environments include *Furcas* [10], *MPS* and *Ipsen* [16].

A syntax directed editing approach usually has rich support from the corresponding framework/environment with which the transformation can easily be specified, i.e., although this depends on the concrete environment, the process is usually quite productive (R3) and the resulting transformation is maintainable (R1) as it is truly bidirectional. Scalability (R2), especially with respect to memory consumption, again depends on the concrete environment, but *incrementality* can easily be supported with such a tightly integrated approach.

A disadvantage is that there is a high dependency on the enclosing framework. This becomes problematic when the transformation is to be ported to a new

[12] OpenArchitectureWare, `http://www.eclipse.org/gmt/oaw/`

[13] The Apache Jakarta Project, `http://jakarta.apache.org/velocity/`

modelling standard or a component has to be replaced. A further disadvantage is that an on-the-fly synchronization of concrete and abstract syntax might not be possible in some application scenarios, as text files might have to be changed "offline". Furthermore, most approaches in this group are geared towards DSL development and are not suitable for, e.g., a scenario where large parts of static text must be generated, which is more suited for template-based code generation.

4.3 Grammar-Based Approaches

Grammar-based approaches such as *Xtext*, *Spoofax* [13], and *Monticore* [11] provide appropriate extensions to EBNF to allow context-sensitive relationships to a certain extent. As depicted in Fig. 13(a), the main idea is to derive as much as possible from the *grammar*, i.e., not only a parser, but also a metamodel, an editor, and an unparser. A metamodel can be extracted from the grammar either via an implicit transformation from EBNF to a modelling language (Ecore in the case of *Xtext*), or by extending EBNF to a complete modelling language which can be used to specify both the textual concrete syntax *and* the abstract syntax of the language combined in the grammar. The latter approach is taken by *Monticore*. Bidirectionality can be supported by using the non-terminals in the grammar to *pretty print* model elements to text.

Grammar-based approaches lead to very compact, concise specifications and are highly productive when the target language can be described with the grammar. Getting an editor "for free" is also a major productivity boost, especially when developing a textual DSL. In general, however, every grammar can only describe a limited class of languages, and, due to the fact that the grammar is used to derive all other components, a fallback to Java similar to what ANTLR offers cannot be supported. Every realistic transformation, therefore, will always require a subsequent model-to-model transformation, especially when the target metamodel was established *before* the textual syntax. In many cases, e.g., round-tripping as opposed to DSL development, the textual syntax *and* the target metamodel are fixed and already exist. In such a case it becomes quite challenging to specify a perfectly fitting grammar.

Supporting bidirectionality is also difficult in complex cases and most approaches do not place a strong focus on bidirectionality, only providing a default pretty printer that must be extended and refined. The price of having a compact, concise specification is that all components are merged making it difficult, if not impossible, to reuse the text comprehension part of the grammar for a different target metamodel, or to change the textual syntax but retain the same metamodel. Last but not least, grammar-based approaches are not suitable for cases where a lot of text is to be ignored or filtered and when a lot of static parts are to be generated.

4.4 Template-Based Approaches

Template-based approaches such as *Xround* [4] and [3] provide an interesting contrast to grammar-based approaches by deriving the complete bidirectional

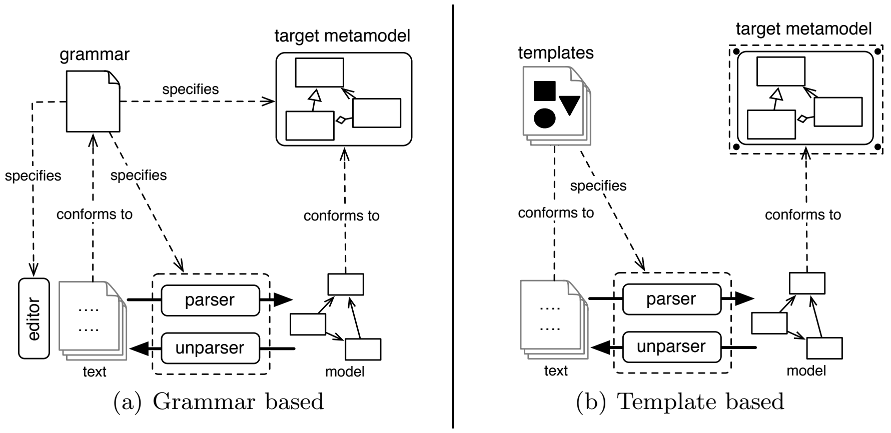


Fig. 13. Schematic overview of grammar-based and template-based approaches

transformation from a set of templates. As depicted in Fig. 13(b), a set of templates in a fixed template language is used to derive an unparser (code is simply generated with the templates) *and* a parser. The parser works by matching text fragments with potential templates until the exact sequence of chosen templates can be identified. Corresponding model elements can be derived from this sequence of templates as the target metamodel is fixed and known to the parser, i.e., there is a mapping between model elements and templates.

In contrast to a grammar-based approach, this works quite well for cases with large parts of static text which must be ignored/generated. For a typical textual DSL, however, with almost a 1-1 relationship between text and model elements for conciseness, the templates must contain a lot of logic and not so much static text, reducing readability and maintainability of the templates.

Although the generated textual syntax can be flexibly varied, the parser can only be realized efficiently if the template language *and* the target metamodel are fixed. This means that a template-based approach is a productive, maintainable solution for a *fixed metamodel*, i.e., a concrete application. The parser would, however, have to be almost completely re-implemented for every new metamodel. Depending on the complexity of the supported template language, it can also be quite challenging to parse textual content using templates in a scalable manner, i.e., complex logic in the templates can easily lead to an explosion of the template search space.

5 Conclusion and Future Work

In this paper, we presented a flexible, general framework for structuring bidirectional model-to-platform transformations. A set of core requirements derived from typical application domains was used to argue the advantages of a clear separation of the transformation into two distinct steps: A text-to-tree transformation (text comprehension/generation) which is held as simple as possible, and a tree-to-model transformation (typing and context-sensitive relations), which should be implemented with a bidirectional language. A realization of our framework in eMoflon[14] shows that a flexible blend of rule-based, declarative languages can be combined successfully. Existing approaches for bidirectional model-to-platform transformation are either not general enough, i.e., only work for a certain standard/domain, or not flexible enough, i.e., components cannot be exchanged. Our choice of TGGs as a bidirectional language opens up a large class of applications for TGGs, with new challenges. Future tasks include improving support for incrementality in our TGG implementation by exploiting the asymmetric nature of model-to-platform transformations (information loss is only in one direction) as compared to the general case. We are also working on optimizing our current TGG algorithm to deal with weakly typed trees, necessary for an efficient inference of context-sensitive relations, and are investigating concepts for improving modularity and reuse/composition of the transformation languages (string grammars, tree grammars, templates, TGGs).

References

1. Anjorin, A., Lauder, M., Patzina, S., Schürr, A.: eMoflon: Leveraging EMF and Professional CASE Tools. In: Heiß, H.U., Pepper, P., Schlingloff, H., Schneider, J. (eds.) Informatik 2011. LNI, vol. 192, p. 281. GI, Bonn (2011)
2. Bézivin, J., Gerbé, O.: Towards a Precise Definition of the OMG/MDA Framework. In: Feather, M., Goedicke, M. (eds.) ASE 2001, pp. 273–280. IEEE, New York (2001)
3. Bork, M., Geiger, L., Schneider, C., Zündorf, A.: Towards Roundtrip Engineering - A Template-Based Reverse Engineering Approach. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 33–47. Springer, Heidelberg (2008)
4. Chivers, H., Paige, R.F.: XRound: Bidirectional Transformations and Unifications Via a Reversible Template Language. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 205–219. Springer, Heidelberg (2005)
5. 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)
6. Efftinge, S., Völter, M.: oAW xText: A Framework for Textual DSLs. In: EclipseCon Europe 2006 (2006)
7. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 296–309. Springer, Heidelberg (2000)
8. Fowler, M.: Domain-Specific Languages. Addison-Wesley, Boston (2010)
9. 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)

[14] Available from `www.emoflon.org`

10. Goldschmidt, T., Becker, S., Uhl, A.: Textual Views in Model Driven Engineering. In: SEAA 2009, pp. 133–140. IEEE, New York (2009)
11. Grönniger, H., Krahn, H., Rumpe, B., Schindler, M., Völkel, S.: MontiCore: A Framework for the Development of Textual Domain Specific Languages Categories and Subject Descriptors. In: Schäfer, W., Dwyer, M.B., Gruhn, V. (eds.) ICSE Companion 2008, pp. 925–926. ACM, New York (2011)
12. Hu, Z., Schürr, A., Stevens, P., Terwilliger, J.: Bidirectional Transformations "bx" (Dagstuhl Seminar 11031). In: Dagstuhl Reports, vol. 1, pp. 42–67. Dagstuhl Publishing, Dagstuhl (2011)
13. Kats, L.C.L., Visser, E.: The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In: Cook, W.R., Clarke, S., Rinard, M.C. (eds.) OOPSLA 2010, pp. 444–463. ACM, New York (2010)
14. Klar, F., Lauder, M., Königs, A., Schürr, A.: Extended Triple Graph Grammars with Efficient and Compatible Graph Translators. In: Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (eds.) Nagl Festschrift. LNCS, vol. 5765, pp. 141–174. Springer, Heidelberg (2010)
15. Lauder, M., Schlereth, M., Rose, S., Schürr, A.: Model-Driven Systems Engineering: State-of-the-Art and Research Challenges. Bulletin of the Polish Academy of Sciences: Technical Sciences 58(3), 409–421 (2010)
16. Nagl, M.: Building Tightly Integrated Software Development Environments: The IPSEN Approach. Springer, Berlin (1996)
17. Parr, T.J.: Enforcing Strict Model-View Separation in Template Engines. In: Feldman, S., Uretsky, M. (eds.) WWW 2004, pp. 224–233. ACM, New York (2004)
18. Parr, T.J.: The Definitive ANTLR Reference: Building Domain-Specific Languages. The Pragmatic Bookshelf, Lewisville (2007)
19. Parr, T.J.: Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages. The Pragmatic Bookshelf, Lewisville (2009)
20. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley, Boston (2009)
21. Stürmer, I., Kreuz, I., Schäfer, W., Schürr, A.: The MATE Approach: Enhanced Simulink and Stateflow Model Transformation. In: MAC 2007. MathWorks, Natick (2007)

Model Transformation Co-evolution: A Semi-automatic Approach

Jokin García, Oscar Diaz, and Maider Azanza

Onekin Research Group, University of the Basque Country (UPV/EHU),
San Sebastian, Spain
{jokin.garcia,oscar.diaz,maider.azanza}@ehu.es

Abstract. Model transformations are precious and effortful outcomes of Model-Driven Engineering. As any other artifact, transformations are also subject to evolution forces. Not only are they affected by changes to transformation requirements, but also by the changes to the associated metamodels. Manual co-evolution of transformations after these metamodel changes is cumbersome and error-prone. In this setting, this paper introduces a semi-automatic process for the co-evolution of transformations after metamodel evolution. The process is divided in two main stages: at the detection stage, the changes to the metamodel are detected and classified, while the required actions for each type of change are performed at the co-evolution stage. The contributions of this paper include the automatic co-evolution of breaking and resolvable changes and the assistance to the transformation developer to aid in the co-evolution of breaking and unresolvable changes. The presented process is implemented for ATL in the CO-URE prototype.

1 Introduction

Model-Driven Engineering (MDE) describes software development approaches that are concerned with reducing the abstraction gap between the problem domain and the software implementation domain. The complexity of bridging the abstraction gap is tackled through the use of models that describe complex system at multiple levels of abstraction and from a variety of perspectives, combined with automated support for transforming and analyzing those models [6]. In this way developers can concentrate on the essence of the problem while reusing mapping strategies. Benefits include increased productivity, shorter development time, improved quality or better maintenance [15]. However, a Damocles' sword hanging over MDE is evolution. The main MDE artifacts are: *(i)* models, *(ii)* metamodels and *(iii)* transformations. While model and transformation evolution can be faced in isolation, metamodel evolution impacts models and transformations alike. Metamodel changes might have disturbing consequences on their instance models, and break apart the associated transformations. The former issue (a.k.a. model co-evolution) has been the subject of substantial work [4,9,16]. Unfortunately, transformation co-evolution has received less attention. Nevertheless, not only are transformations main enablers of the MDE advantages but their creation is programming intensive and frequently more costly than its model counterpart [5]. This substantiates the effort to provide solid basis to assist during the transformation co-evolution effort.

Unlike previous approaches [10], we do not force to describe the evolution in terms of *ad-hoc* operands, but evolution is ascertained from differences between the original and the evolved metamodel. Next, differences are classified as [4]: *(i) Non Breaking Changes (NBC)*, i.e., changes that do not affect the transformation; *Breaking and Resolvable Changes (BRC),* i.e., changes after which the transformations can be automatically co-evolved; and *Breaking and Unresolvable Changes (BUC),* i.e., changes that require human intervention to co-evolve the transformation. Finally, the transformation is subject to distinct actions based on the type of the change, i.e., no action for NBC, automatic co-evolution for BRC, and assisting the user for BUC. The outcome is an evolved transformation that tackles (or warns about) the evolved metamodel. This approach is realized in the CO-URE prototype that takes as input the original Ecore metamodel, the evolved Ecore metamodel and an ATL rule transformation [11], and outputs an evolved ATL transformation. CO-URE makes intensive use of High-Order Transformations (HOTs) whereby the original transformation is handled as a model which needs to be mapped into another model (i.e. the evolved transformation). The approach can be generalized to any transformation language that provides a metamodel representation. We regard as main contributions (1) the automatic co-evolution of BRC, (2) the assistance for BUC, and (3), the CO-URE prototype.

The paper starts with a motivating scenario. Next, we outline the co-evolution process whose two main stages, detection and co-evolution, are presented in more detail in Sections 4 and 5, respectively. Section 6 introduces the CO-URE architecture and describes one of its HOT rules. Related work and conclusions end the paper.

2 Motivating Scenario

As any other software artifact, metamodels are subject to evolution. During design alternative metamodel versions may be developed. During implementation metamodels may be adapted to a concrete metamodel formalism supported by a tool. Finally, during maintenance errors in a metamodel may be corrected. Moreover, parts of the metamodel may be redesigned due to a better understanding or to facilitate reuse [22]. Simultaneously, metamodels lay at the very center of the model-based software development process. Both models and transformations are coupled to metamodels: models *conform to* metamodels, transformations *are specified upon* metamodels. Hence, metamodel evolution percolates both models and transformations. We focus on transformation co-evolution after metamodel evolution.

We use the popular *Exam2MVC* transformation [13] as a running example. This scenario envisages different types of exam questions from which Web-based exams are automatically generated along the MVC pattern [13]. Figure 1 presents the *ExamXML* metamodel and the *AssistantMVC* metamodel. The *Exam2MVC* transformation generates an *AssistantMVC* model out of an *ExamXML* model.

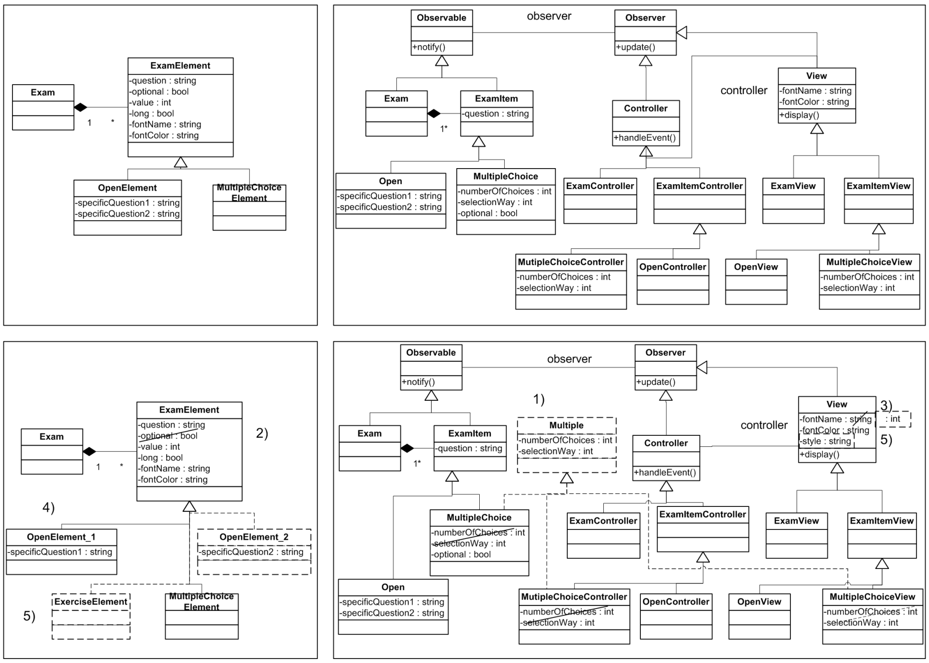


Fig. 1. *ExamXML* metamodel and *AssistantMVC* metamodel: original (above) and evolved (below)

Next, we introduce a set of evolution scenarios to be considered throughout the paper (see Figure 1):

– Scenario 1. The *AssistantMVC's Multiple* class is introduced in the target metamodel. This new class abstracts away the commonality of three existing classes: *MultipleChoiceController, MultipleChoiceView* and *MultipleChoice.*
– Scenario 2. Property *optional* is deleted from *ExamXML's ExamElement.*
– Scenario 3. The *AssistantMVC's fontColor* metaproperty is changed from *string* to *integer.*
– Scenario 4. The *ExamXML's OpenElement* class is splitted into *OpenElement_1* and *OpenElement_2.*
– Scenario 5. New subclass *ExerciseElement* is added to *ExamElement* metaclass, and a new property *style* is added to *View* target metaclass.

The question is now how these changes impact the *Exam2MVC* transformation, better said, how can the designer be assisted in propagating these changes to the transformation counterpart. Next section outlines the process.

3 Transformation Co-evolution Process: An Outline

This section outlines the transformation co-evolution process aiming at assisting designers by automating co-evolution whenever possible. This process comprises

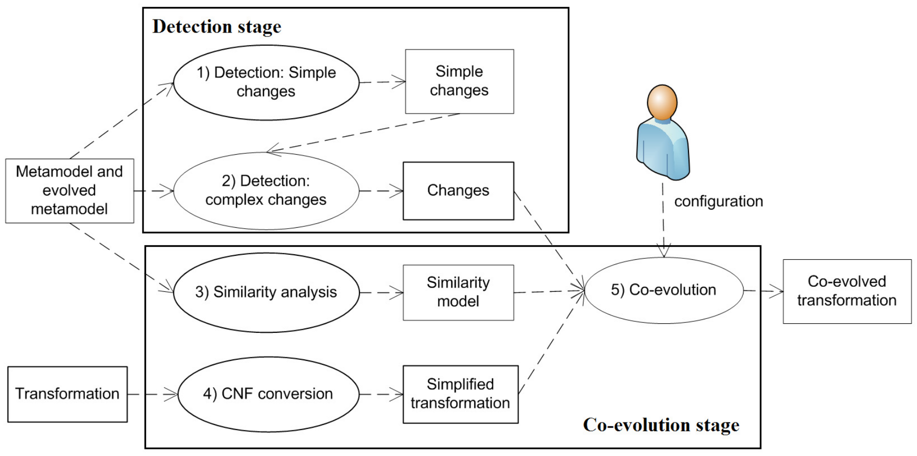


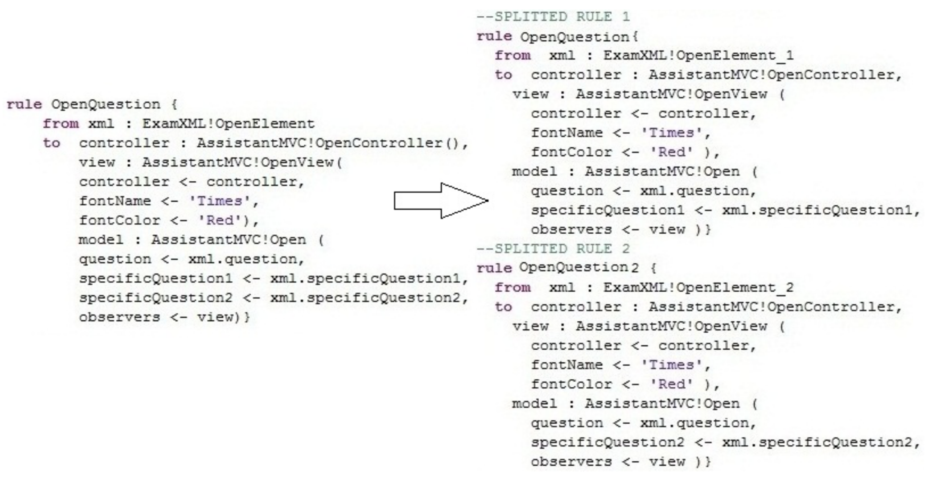
Fig. 2. Transformation co-evolution process

two main stages: *detection* and *co-evolution* (see Figure 2). Inputs include the original metamodel (M), the evolved metamodel (M') and the original transformation (T).

Detection Stage. The original metamodel and the modified metamodel are compared, and a set of differences are highlighted. Differences can range from simple cases (e.g. 'class renaming') to more complex ones (e.g. 'class splitting'). Simple changes are those that are conducted as a single shot by the user. By contrast, complex changes are abstractions over simple ones as they conform a meaningful transaction on the metamodel. Complex changes need to be treated as a unit not only from the perspective of the metamodel, but also from the co-evolution perspective. Otherwise, we risk to miss the intention of the designer when evolving the metamodel, and hence, to propagate this misunderstanding to the transformation. To this end, the detection stage includes two tasks: simple-change detection and complex-change detection. The outcome is a set of changes, both simple and complex.

Co-evolution Stage. Having a set of metamodel changes as input, this step first classifies changes based on their impact on the transformation rules. Based on the notation used in [4], we identify three types of metamodel changes:

1. *Non Breaking Changes (NBC).* These changes have no impact on the transformation. This case is illustrated by the first scenario: the introduction of the *Multiple* class as an abstraction of two existing classes. Superclass extraction has generally no impact on the transformation since metaclass properties are still reachable through inheritance. Therefore, this type of changes need to be detected, but no further action is required.
2. *Breaking and Resolvable Changes (BRC).* These changes do impact the transformation rules, but this impact is amenable to be automated. The fourth scenario is a case in point. Here, *OpenElement* is splitted into *OpenElement_ 1* and



```
rule OpenQuestion {
    from xml : ExamXML!OpenElement
    to  controller : AssistantMVC!OpenController(),
        view : AssistantMVC!OpenView(
        controller <- controller,
        fontName <- 'Times',
        fontColor <- 'Red'),
        model : AssistantMVC!Open (
        question <- xml.question,
        specificQuestion1 <- xml.specificQuestion1,
        specificQuestion2 <- xml.specificQuestion2,
        observers <- view)}
```

```
--SPLITTED RULE 1
rule OpenQuestion{
  from  xml : ExamXML!OpenElement_1
  to  controller : AssistantMVC!OpenController,
    view : AssistantMVC!OpenView (
      controller <- controller,
      fontName <- 'Times',
      fontColor <- 'Red' ),
    model : AssistantMVC!Open (
      question <- xml.question,
      specificQuestion1 <- xml.specificQuestion1,
      observers <- view )}
--SPLITTED RULE 2
rule OpenQuestion2 {
  from  xml : ExamXML!OpenElement_2
  to  controller : AssistantMVC!OpenController,
    view : AssistantMVC!OpenView (
      controller <- controller,
      fontName <- 'Times',
      fontColor <- 'Red' ),
    model : AssistantMVC!Open (
      question <- xml.question,
      specificQuestion2 <- xml.specificQuestion2,
      observers <- view )}
```

Fig. 3. *Exam2MVC* transformation: original (above), co-evolved (below)

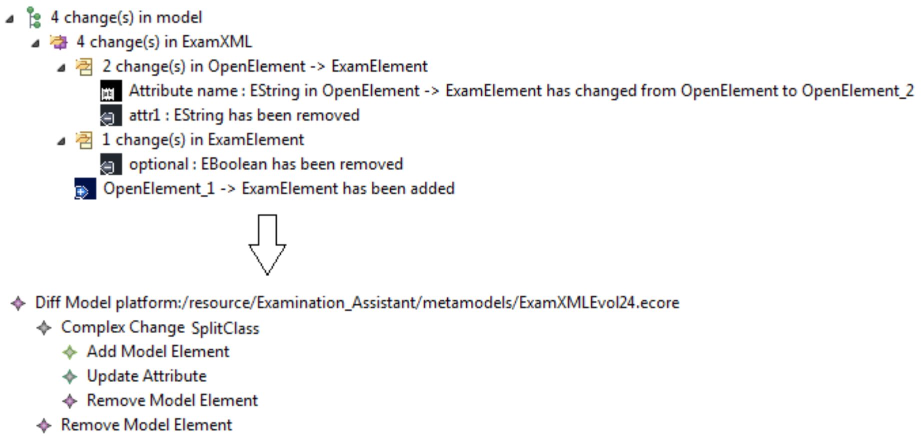
OpenElement_ 2 classes. Accordingly, rules having *OpenElement* as its source might give rise to two distinct transformation rules that tackle the specifics of *OpenElement_ 1* and *OpenElement_ 2* (see Figure 3*).*

3. *Breaking and Unresolvable Changes (BUC).* These changes also impact the transformation, but full automatization is not possible and user intervention is required. Reasons include: the semantics of the metamodel, the specific characteristics of the transformation language, or the specificity of the change. Hence, it will be designer's duty to manually guide the co-evolution. This is illustrated by scenario 3: *AssistantMVC's fontName* metaproperty is changed from *string* to *integer*. Type changes are the most ambiguous ones due to transformation languages being dynamically typed, and hence, susceptible to generate type errors at runtime. For instance, a rule could assign '*Times*' to *fontName*. *FontName* has now be turned into an integer, hence, making this rule inconsistent. In those cases, the option is to warn about the situation, and let the designer provide a contingency action (e.g. coming up with the *"integer"* counterpart of the formerly valid value '*Times*').

In short, for each type of change (i.e. NBC, BRC or BUC), we propose a course of action: no action, automatic transformation, and assisted transformation, respectively. To this end, the co-evolution process is complemented by two auxiliary steps: a **Conversion to Conjunctive Normal Form (CNF) step** (to address removals) and an optional **similarity analysis step** (to handle additions). Next two sections delve into the details.

4 Detection Stage

This stage takes as input both the original metamodel and the evolved metamodel, and infers the set of changes that went in between. This is achieved through two tasks: simple-change detection and complex-change detection.


4 change(s) in model
4 change(s) in ExamXML
2 change(s) in OpenElement -> ExamElement
Attribute name : EString in OpenElement -> ExamElement has changed from OpenElement to OpenElement_2
attr1 : EString has been removed
1 change(s) in ExamElement
optional : EBoolean has been removed
OpenElement_1 -> ExamElement has been added

Diff Model platform:/resource/Examination_Assistant/metamodels/ExamXMLEvol24.ecore
Complex Change SplitClass
Add Model Element
Update Attribute
Remove Model Element
Remove Model Element

Fig. 4. The *Difference* model (above) *&* *DiffExtended* model (below) for the running example. Simple changes that account for a more abstract complex change are arranged as descendants of the complex change (e.g. *AddModel*, *UpdateAttribute*, *RemoveElement* are now part of a *ComplexChange* whose *changeType* is *SplitClass*).

4.1 Simple-Change Detection

We detect simple changes as a difference between the original metamodel and the evolved metamodel. To this end, we use *EMF Compare* [19]. This tool takes two models as input and obtains the differences along the *Difference metamodel*. Back to our running example, EMF Compare is used to detect the simple changes between the original and evolved *ExamXML* metamodel as well as the original and evolved *AssistantMVC* metamodel. The output is a *Difference* model. Figure 4 (above) illustrates this *Difference* model for the *ExamXML* metamodel (scenarios 2 and 4): *UpdateAttribute*, *RemoveModelElement*, *AddModelElement* and *RemoveAttribute*. In other words, it detects that the name of the class is changed from *OpenElement* to *OpenElement_ 2*, the *specificQuestion1* metaproperty is removed from *OpenElement_ 2*, the attribute *optional* is being removed, and a new class with name *OpenElement_ 1* is added.

4.2 Complex-Change Detection

Simple changes might be semantically related to achieve a common higher-order modification. For a list of complex changes refer to [9] (we are going to analyze those relevant from the point of view the transformation co-evolution). For instance, the previous *AddModelElement* simple change hides a class split. We need to infer that a set of simple changes unitedly account for a split. Alternatively, we risk to treat each simple change on its own, which could lead to unwanted co-evolution in the transformation.

We regard complex changes as predicates over simple changes. These are auxiliary predicates needed to define them: C is the set of metaclasses and P the

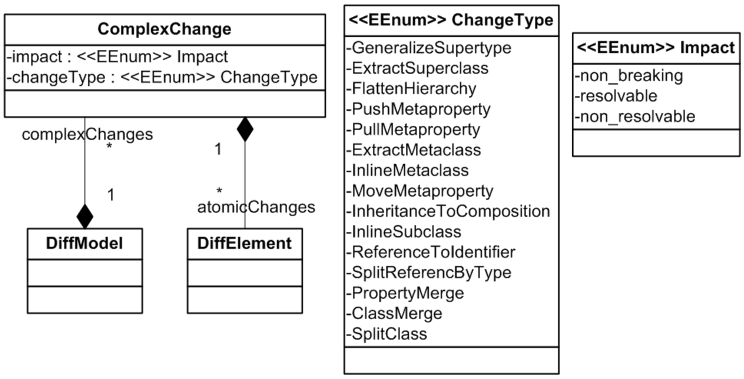


Fig. 5. *DiffExtended* metamodel: *EMFCompare*'s *Difference* metamodel is extended with the *ComplexChange* class

set of metaproperties of a metamodel. **Subclass(s: C, c: C)**: *s* is subclass of *c*; **Added_class(c: C)**: *c* is added to the metamodel; **Added_attribute(p: P, c: C)**: *p* has been added to *c;* **Deleted _attribute(p: P, c: C)**: *p* has been deleted from *c;* **IsAttributeOfClass(p: P, c: C)**: *p* belongs to *c;* **Added _supertype(s: C, c: C)**: supertype relationship has been added from *s* to *c;* **Deleted_supertype(s: C, c: C)**: supertype relationship has been deleted from the *s* to *c;* **Added_reference(p: P, c: C, d: D)**: Reference *p* from *c* to *d* is added; **Deleted_reference(p: P, c: C, d: D)**: Reference *p* from *c* to *d* is deleted; **Added_composition(s: C, c: C)**: composition relationship has been added from the *s* to the *c;* **Composed_name(z: string, p: string, x: string)**: delivers a new string *x* out of input strings *z* and *p;* **Splitted_name(c, x)** returns true if *c* can be obtained from *x* by concatenating such suffix. A notation convention exists to name split classes: the name of the original class concatenated with a number (e.g. *OpenElement_1, OpenElement_2*); **SplitClassName(c: C)**: returns true if the new name of c is the concatenation of the old name and "_1". The list of detection predicates follows:

- **ExtractSuperclass(c:C)** iff Added_class(c) ∧ ∃p∈P, ∃s∈C (Added_attribute(p, c) ∧ Added_supertype(s, c) ∧ Deleted _attribute(p, s))
- **PullMetaproperty(c:C, s:C)** iff ∃p∈P (Subclass(s, c) ∧ Added_attribute(p,c) ∧ Deleted _attribute(p, s))
- **PushMetaproperty(p: P)** iff ∃s, c∈C (Subclass(s, c) ∧ Deleted_attribute(p,c) ∧ Added _attribute(p, s))
- **FlattenHierarchy(c:C)** iff (Deleted_class(c) ∧ ∀p∈ P | IsAttributeOfClass(p, c) , ∀s∈ C | Subclass(s, c) (Deleted _attribute(p,c) ∧ Deleted_supertype(s, c) ∧Added_attribute(p, s)))
- **MoveMetaproperty(c:C, p:P, d:C)** iff (Deleted_attribute(p,c) ∧ Added _attribute(p, d))

- **ExtractMetaclass(c:C, d:C)** iff (Added_class(d) ∧ ∀p∈P | IsAttributeOfClass(p, c) (Added _attribute(p,d) ∧ Deleted_attribute(p, c)))
- **InlineMetaclass(c:C, d:C)** iff (Added_class(d) ∧ Deleted_class(c) ∧ ∀p∈ P | IsAttributeOfClass(p, c) (Added _attribute(p,d) ∧ Deleted_attribute(p, c)))
- **InheritanceToComposition(c:C, d:C)** iff (Deleted_supertype(d, c) ∧ Added _composition(d, c))
- **GeneralizeSupertype(c:C, s:C, d:C)** iff (Deleted_supertype(d, s) ∧ Added _supertype(d, c) ∧Subclass(s, c))
- **InlineSubclass(c:C, d:C)** iff (Deleted_class(c) ∧ Subclass(c, d) ∧ ∀p∈P | IsAttributeOfClass(p, c) (Added _ attribute(p,d) ∧ Deleted_attribute(p, c)))
- **ReferenceToIdentifier(c:C, d:C, p:P)** iff (Deleted_reference(p, c, d) ∧ Added _attribute(p, c) ∧ Added_attribute(p, d))
- **SplitReferenceByType(c:C, d:C, x:C, s:C, p:P, y:P, z:P)** iff (Deleted_ reference(p, c, d) ∧ Added_reference(y, c, x) ∧ Added_reference(z, c, s))
- **PropertyMerge(p:P, z:P, x:P)** iff ∃c∈C (Deleted_attribute(p,c) ∧ Deleted _attribute(z, c) ∧ Added_attribute(x, c) ∧ Composed_name(z, p, x)). The last predicate delivers x by concatenating strings z and p.
- **ClassMerge(c:C, d:C)** iff ∃y∈C (Subclass(c, y) ∧ Subclass(d, y) ∧ Deleted_class(d) ∧ Composed_name(c, d, x))
- **SplitClass(c:C, d:C, x:C)** iff ∃y∈C (Subclass(c, y) ∧ Subclass(d, y) ∧ Added_class(d) ∧ Splitted_name(d, c)∧ SplitClassName(c)). The latter predicate needs a bit of explanation.

Implementation wise, simple changes are obtained using *EMFCompare* using the *Difference* metamodel. We propose to extend the *Difference metamodel* to account also for complex changes. Figure 5 shows an extract of the *DiffExtended* metamodel. Using the predicates aforementioned we infer complex changes that are represented as a *DiffExtended* model. Figure 4 provides a *DiffExtended* model where complex changes are also introduced. In the case that a simple change can belong to different complex changes, the biggest one has priority, e.g. *Flatten-Hierarchy* over *MoveMetaproperty*, as the first one includes the second.

In short, this task is realized as a transformation that takes a *Difference* model as input and obtains a *DiffExtended* model that includes both single and complex changes. Now, we are ready to percolate those changes to the transformation rules.

5 Co-evolution Stage

5.1 Similarity Analysis Step

Additional degrees of automatization can be achieved by using metamodel matching techniques. A similarity analysis is conducted between the source and target metamodels using tools such as AML (AtlanMod Matching Language) [7]. These tools compute similarity based on the element names and the structural similarity of the metamodels. The output can be used to assist designers to fill the gaps. The approach rests on the matching effectiveness. We performed an empirical experiment based on a test-bed of 17 transformations from the ATL zoo[1]. Matching effectiveness had an average of 22-23% success (i.e., cases were an adequate binding could be suggested to the designer). This step is optional, and the weaving similarity model can be added as an input to the adaptation.

5.2 Conjunctive Normal Form Conversion Step

Rule filters are first-order predicates, normally specified using OCL. Equivalence rules of Predicate Calculus are applied to each boolean expression to get its equivalent *Conjunctive Normal Form (CNF)*, i.e., a conjunction of clauses, where a clause is a disjunction of literals (see [3] for futher details). Once in CNF, filters can be subject to "surgically removal", as explained in Subsection 5.4.

5.3 Co-evolution Step

We treat transformations as models. That is, transformations are described along a transformation metamodel. Therefore, it is possible to define (high order) transformations (HOTs) that take a transformation as input, and return a somehow modified transformation. This is precisely the approach: define correspondences that map the original transformation into an evolved transformation, taking the changes obtained during the detection stage as parameters. These HOTs are realized as ATL rules. In what follows, we summarize those rules in terms of co-evolution actions. These actions are expressed as predicates over the original transformation rules. To this end, we capture a transformation rule R as a tuple *Rule(id, source, targets, filters, mappings)* where *"source"* and *"targets"* refer to classes of the input and output metamodel, respectively; *"filters"* is a set of related predicates over the source element, such that the rule will only be triggered if the condition is satisfied; finally, *"mappings"* refer to a set of bindings to populate the attributes of the target element. A binding construct establishes the relationship between a source and a target metamodel elements. Normally, a mapping part contains a binding for each target metaclass' property. Its semantics denote what needs to be transformed into what instead of how the transformation must be performed. The left-hand side must be an attribute of the target element metaclass. The right-hand side can be a literal value, a query or an expression over the source model. Figure 3 illustrates an example of a transformation in ATL.

Transformation rules are the facts. Next, co-evolution actions are described through a set of operands and predicates over these rule facts. To avoid cluttering the description with iterations, we consider multi-valued predicates to return a single value. For instance, if a set of rules is used as parameter in the following

[1] http://www.eclipse.org/m2m/atl/atlTransformations/

Bindings(r), bindings of all the rules in the set will be returned. Underscore will be used similarly to Prolog, as "don't care" variables. Predicates are intensional definitions of rule sets, and include: **RulesBySource(s)** denotes the set of rules whose source is *s;* **RulesByTarget(t)** denotes the set of rules whose target is *t;* **Binding(r, p)** returns the bindings of rule *r* which hold property *p;* **Bindings(r)** returns the bindings of rule *r;* **TargetsOfRule(r)** returns the targets of rule *r;* **FiltersOfRule(r)** returns the filters of rule *r;* **FiltersOfProperty(p)** returns the filters where the property *p* appears.

Operands act on rules: **deleteRule(r)**, which deletes the rule *r;* **deleteTarget(r, t)** which deletes target *t* from rule *r;* **deleteBinding(r, b)**, which deletes binding *b* from rule *r;* **addRule(r)**, which adds rule *r;* **addTarget(r, t)**, which adds target *t* to rule *r;* **addBinding(r, b)**, which adds a binding *b* to rule *r;* **moveTarget(r1, t, r2)**, which moves *r1*'s target *t* together with its bindings to rule *r2;* **moveBinding(r1, b, r2)** which moves *r1*'s binding *b* to *r2*, provided *r2* holds a target that matches *b*'s lefthand side; **updateFilter(r1, f1, f2)**, which updates *f1* by *f2* among *r1*'s filters; **deleteFilter(r1, f1)**, which deletes one of *r1*'s filters; **updateBinding(r1,b1, b2)**, which substitutes *r1*'s binding *b1* by *b2;* **updateSource(r, s1, s2)**, which updates source *s* of rule *r* to *s2;* **concatClass(c1, c2)**, which concats two classes names. These operands are used to specify how metamodel changes impact the transformation rules, i.e. the co-evolution actions. The list below and the list at the end of this subsection summarize the actions related to simple and complex changes, respectively.

- **removeMetaclass (c: C)**: (BRC) deleteRule(RulesBySource(c)), deleteFilter(RulesBySource(c), FiltersOfProperty(c.properties)), deleteBinding(RulesBySource(c), Binding(RulesBySource(c), c.properties))
- **removeMetaproperty(p: P)**: (BRC) deleteFilter(RulesBySource(c), FiltersOfProperty(p)), deleteBinding(RulesBySource(c), Binding(RulesBySource(c), p)). Deletions should be minimal (in Subsection 5.4)
- **updateLowerBound (p: P,NewBound)**: (NBC) No action
- **updateUpperBound (p: P,NewBound)**: (NBC) updateFilter(_, FiltersOfProperty(p), f2), updateBinding(_, Binding(_, p), b2). In case *lowerBound* converts from 1 to *, *f2* will insert a *forAll* expressions to check that all instances fulfill the condition and *b2* will use the *first()* to take the first element of the sequence. In case *lowerBound* changes from * to 1, *asSequence()* will be used in *f2* and *b2* to convert an element into a sequence.
- **updateEType**: (BUC) Syntactically right, but possible runtime type errors (refer to [18]). A warning note is generated.
- **updateESuperTypes**: (BUC) (if a metaproperty of the ancestors is accessed) Propose to copy the metaproperty of the superclass in the class.
- **updateIsAbstract (c: C,NewValue)**: (NBCor BRC) If metaclass c is turned into abstract (NewValue = "true") : Delete (Rule(c)), Delete (RHS(c)). If metaclass c is turned into a non-abstract class (NewValue = "false") then, do nothing.

- **updateELiterals (c: C)**: (BUC) Comment the structure the literal is used in, in case the user wants to use another literal. Alternative: use the default one.
- **addClass(c: C)**: (NBC) see Subsection 5.5
- **add Metaproperty(p: P)**: (NBC) see Subsection 5.5

Next, we illustrate the distinct casuistic using our running example:

- Scenario 1. The *AssistantMVC's Multiple* class is introduced in the target metamodel. This is a NBC scenario.
- Scenario 2. The property *"optional"* is deleted from *AssistantMVC's ExamElement.* When a property is removed from the metamodel, different approaches can be taken, where the most simplistic one could be to remove the whole transformation rule where the property is used in a binding or boolean expression. However, this is a very restrictive and rather coarse-grained approach. We advocate the use of what we call the *principle of minimum deletion*, where only the part that is absolutely necessary is removed (see next subsection).
- Scenario 3. The *AssistantMVC's fontName* metaproperty is changed from *string* to *integer*. This is a BUC case.
- Scenario 4. The *AssistantMVC's OpenElement* class is splitted into *OpenElement_1* and *OpenElement_2*. As a result, rules having OpenElement as source should be co-evolved (see Figure 3). This is the case of the *OpenQuestion* rule, which is splitted in two rules: *OpenQuestion_1 and OpenQuestion_2.* The former contains the bindings related to *OpenElement_1* while the latter keeps the bindings for *OpenElement_2.*
- Scenario 5. New subclass *ExerciseElement* is added to *ExamElement* metaclass, and a new property *style* is added to *View* target metaclass. Additive evolution is a NBC case. Even though, it is not unusual to need new rules or bindings to maintain the metamodel coverage level. For this purpose we include in the co-evolution the option to generate partially new rules, as they are not fully automatable (see Subsection 5.5).

Complex Changes and their impact on transformation evolution:

- **MoveMetaproperty (c: C, p: P, d: C) if c, d ∈ SourceClasses**: updateBinding(RulesBySource(c), Binding(RulesBySource(c), p), newBinding(p)) where newBinding works out a binding by navigating to the new location of the property, in case both classes *c* and *d* are related (navigability exists through associations). If they are not related, user assistance will be needed.
- **MoveMetaproperty (c: C, p: P, d: C) if c, d ∈ TargetClasses**: deleteBinding(RulesByTarget(c), Binding(RulesByTarget(c), p)) or if Binding(RulesByTarget(d), p) > 0: moveBinding(RulesByTarget(c), Binding (RulesByTarget(c), p), RulesByTarget(d)).
- **FlattenHierarchy (c: C) if c ∈ SourceClasses**:
deleteRule(RulesBySource(c)) and {if RulesBySource(Subclass(c)) >0 then moveBinding(RulesBySource(c), Binding(RulesBySource(c), p),
RulesBySource (subclass(c))) else addRule(rule(_, subclass(c), _, _, _))}.

- **FlattenHierarchy (c: C) if c $\in$ TargetClasses**: deleteTarget(RulesBySource(c), c) and {if RulesByTarget(Subclass(c)>0 then moveBinding(RulesBySource(c), Binding (RulesBySource(c), p), RulesBySource(subclass(c))) else addTarget (RulesBySource(subclass(c)), Subclass(c))}.
- **ExtractMetaclass (c: C, d: C) if c, d $\in$ SourceClasses**: addRule(rule(id, c, d, _, _)) and moveBindings(RulesBySource(c), Bindings(RulesBySource(c)), id).
- **ExtractMetaclass (c: C, d: C) if c, d $\in$ TargetClasses**: addTarget(Rule(c, d), d) and moveBinding(RulesBySource(c), Bindings(RulesBySource(c)), addTarget(Rule(c, d), d)).
- **InlineMetaclass (c: C, d: C)**: "Extract metaclass" case and deleteRule(RulesBySource(c)).
- **InheritanceToComposition (c: C, d: C):** When *c* is the source: updateFilter(RulesBySource(c), FiltersOfRule(RulesBySource(c)), f2), where in f2 *refImmediateComposite()* will be used in the filter. For instance: select(v | v.oclIsTypeOf (OpenElement))[Expression] will be converted to ExamElement.refImmediateComposite() [Expression]. When *d* is the source: updateBinding(RulesBySource(c), Bindings(RulesBySource(c)), b2), where in *b2* the name of the composition relation will be introduced in the path of the binding. For instance, OpenElement.question [Expression] must be converted to OpenElement.examElement.question [Expression].
- **GeneralizeSupertype (c: C, s: C, d: C):** deleteBinding(RulesBySource(c), Binding (RulesBySource(c), Metaproperties(s))).
- **InlineSubclass (c: C, d: C):** deleteRule(RulesBySource(c)) and moveBinding (RulesBySource(c), Bindings(RulesBySource(c)), RulesBySource(d)).
- **ReferenceToIdentifier (c: C, d: C, p: P):** (As a convention, the id will have the same name as the deleted reference) updateBinding(RulesBySource(c), Binding (RulesBySource(c), p), newBinding), where the *newBinding* will replace *reference* by *metaclass.id,* e.g. if metaclass C with a relation p to D is converted to C with a metaproperty referring to the new id in D, bindings $p \leftarrow D$ (being D a reference to the generated element of type D) will be adapted to $p \leftarrow D.p.$
- **SplitReferenceByType (c: C, d: C, x: C, s: C, p: P, y: P, z: P):** deleteBinding(RulesBySource(d), p) and if x and s elements are created in the same rule:addBinding(RulesBySource(d), new_b).
- **PropertyMerge (p: P, z: P, x: P) if p, z, x $\in$ SourceProperties:** updateBinding(_, Binding(_, p), newBinding), where in newBinding x is used instead of p or z.
- **PropertyMerge (p: P, z: P, x: P) if p, z, x $\in$ Target**et**Properties**: updateBinding(_, Binding(_, p), newBinding) and deleteBinding(_, Binding(_, z)), where newBinding will use x instead of p and z.
- **ClassMerge (c: C, d: C) if c, d $\in$ SourceClasses:** deleteRule(RulesBySource(c)) and deleteRule (RulesBySource(d))

Table 1. Truth table for removed elements. R_T value will be interpreted as true, and R_F value as false. *L* represents a literal, which is an OCL expression that can be evaluated to a boolean value and does not include a boolean change.

L_1	L_2	L_1 AND L_2	L_1OR L_2	NOT L_1
R_T	L_2	L_2	R_T	R_F
R_F	L_2	R_F	L_2	R_T
R_T	R_F	R_F	R_T	-
R_T	R_T	R_T	R_T	-
R_F	R_F	R_F	R_F	-

and addRule(rule(_, concatClass(c, d), union(TargetsOfRule (RulesBySource(c)), TargetsOfRule(RulesBySource(d))), _ , union(Bindings (RulesBySource(c)), Bindings(RulesBySource(d)))). If there are filtes in the rules: updateSource(_, c, concat(c, d)) and updateSource(_, d, concat(c, d)).

- **ClassMerge (c: C, d: C) if c, d ∈ TargetClasses**: deleteTarget (RulesByTarget(c), c) and deleteTarget(RulesByTarget(d), d) and addTarget(RulesByTarget(c), concatClass(c, d)) and addTarget(RulesByTarget(d), concatClass(c, d)).
- **SplitClass (c: C, d: C):** deleteRule(RulesBySource(c)) and addRule(rule(_, d, TargetsOfRule(RulesBySource(c)), FiltersOfRule(RulesBySource(c)), Binding(RulesBySource(c), Metaproperties(d)))) and addRule(rule(_, SplitClassName(c), TargetsOfRule(RulesBySource(c)), FiltersOfRule(RulesBySource(c)), Binding (RulesBySource(c), Metaproperties(SplitClassName(c))))) .
- **PushMetaproperty (p: P):** (like move metaproperty)

5.4 The Case of the *removeProperty* Change

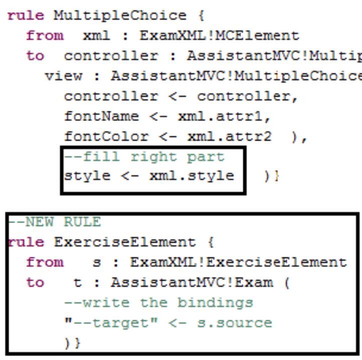
When a metaclass or a metaproperty is deleted, affected transformation elements have to be removed while keeping the transformation logic coherent. Coherence refers to deleting only the strictly necessary parts to prevent negative consequences. For instance, two rules might exist with complementary filters. Those filters may refer to a property. If the deletion of this property leads to the removal of the whole filter, these two rules will no longer have a discriminating filter. Therefore, the impact of metamodel element deletions should be as restrictive as possible. This is specially pressing for rule filters. This subsection discusses a way to "surgically" remove "dead" parts of rule filters. Casuistic includes:

- *Expressions with string concatenation.* This is the easiest case, let be *style ← fontName + fontColor;* an expression with the concatenation of two string metaproperties, if one of them *(e.g. fontName)* is removed, then the expression is re-adapted to contain the rest of the metaproperties, i.e. the new expression is changed to *style* ← *fontColor.*

- *Expressions with creator operations of collections:* Collection types are *sets*, *ordered sets*, *bags*, and *sequences*. With expressions like *Set{London, Paris, Madrid}* → *union(Set{birthCity, liveCity, workCity})*, after removing a metaproperty (e.g. *birthCity*), the new expression will keep the rest of the elements, i.e *Set{London, Paris, Madrid}* → *union(Set{liveCity, workCity}).*
- *Expressions with other operations on collections:* There are other operations to work with collections, as *append(obj)*, *excluding(obj)*, *including(obj)*, *indexOf(obj)*, *insertAt(index, obj)*, and *prepend(obj)*. In this case, if the removed metaproperty is the parameter of the function, this part of the expression is removed. So, with an expression like *Set{London, Paris, Madrid}* → *append(workCity)*, after removing the *workCity* metaproperty the new expression will maintain the left hand of the expression, i.e *Set{London, Paris, Madrid}.*
- *Boolean expressions:* since a removed metaproperty cannot be evaluated, that element in the expression must be considered as undefined. Moreover, before rewriting the expression with that undefined part, it is convenient to simplify the expression as much as possible, i.e. converting it into another equivalent expression, easier to deal with. Thus, equivalence rules of *Predicate Calculus* are applied to each boolean expression to get its equivalent CNF. Inspired by [3], table 1 is proposed as truth table which defines conversion rules for CNF expressions.

  As an example, consider a metamodel with three metaproperties: *ErasmusGrant*, that says if the student has an Erasmus type grant; *speakEnglish*, that says if s/he has a good English level; and *enrolledLastYear*, that indicates if s/he is in his/her last undergraduate year. In the process of metamodel redesign, the designer could help giving some clue about the reason to take the decision of removing a metaproperty from the metamodel (*removal policy*). For example, if all students in the university had a very good level of English (because it is a new precondition for the enrollment), it could be considered as satisfied by default, and in case of removing the *speakEnglish* metaproperty, its value could be reinterpreted as *removed&satisfied-by-default* ($\mathrm{R_T}$). On the other hand, if the university had decided not to participate in the Erasmus Program, no student would have such grant, and in case of removing the *ErasmusGrant* metaproperty, its value could be reinterpreted as *removed&unsatisfied-by-default* ($\mathrm{R_F}$).

  If, in the previous example, there had been this expression *not ErasmusGrant or (speakEnglish and enrolledLastYear)*, and later the redesign process decided to remove the *speakEnglish* metaproperty, then according to the truth table the expression would be rewritten as *not ErasmusGrant or enrolledLastYear*; if the removed metaproperty had been *ErasmusGrant* with $\mathrm{R_F}$ policy, then the new expression would have been *true.*
- *Expressions with loop operations:* In ATL the syntax used to call an iterative expression is the following: *source* → *operation_name (iterators / body).* Among these operations there are *any(expr)*, *collect(expr)*, *exists(expr)*,



```
rule MultipleChoice {
  from  xml : ExamXML!MCElement
  to  controller : AssistantMVC!MultipleChoiceController,
    view : AssistantMVC!MultipleChoiceView (
      controller <- controller,
      fontName <- xml.attr1,
      fontColor <- xml.attr2  ),
      --fill right part
      style <- xml.style    )}

--NEW RULE
rule ExerciseElement {
  from   s : ExamXML!ExerciseElement
  to   t : AssistantMVC!Exam (
      --write the bindings
      "--target" <- s.source
      )}
```

Fig. 6. Generated skeletons for the new *style* property (above) and the new *ExerciseElement* class (below)

forAll(expr), *one(expr)*, *select(expr)*, and so on. For instance, in *self.items* → *exists(i | i.question.size()>50),* if the removed metaproperty (e.g. *items*) takes part in the source, the whole expression is removed, but if the removed metaproperty takes part in the body, the rules for the boolean expressions must be applied.

– For more ambiguous cases, we resort to reporting the ambiguity and letting the designer decide. For instance, if the returned type of a helper is removed, the helper cannot be considered during binding, and a warning note is introduced. Or if the removal of a property makes the scope of two rules coincide then, the first one is commented.

Back to our second scenario (i.e. removal of *optional* from *ExamElement*), consider we have two rules whose filters refer to *optional*:

– *(value>5 and optional) or long.* Applying equivalences from table 1, the evolved filter becomes *(value > 5 or long)*
– *not ((value>5 and optional) or long).* Using Morgan's laws, its CNF counterpart is: *(not value>5 or not optional) and not long.* Applying equivalences from table 1, the evolved filter results in *(not value>5 and not long).*

In this way, "surgical removal" permits to limit the impact of deletion of properties in the associated rules.

5.5 The Case of *addClass* and *addProperty* Changes

Although additive evolution is considered *NBC*, it is not unusual to need new rules or bindings to maintain the metamodel coverage level. For this purpose, we include in the co-evolution the option to generate partially new rule skeletons. Our fifth scenario illustrates this situation: addition of the *ExerciseElement* metaclass, and addition of the *style* property to the *View* metaclasss. The engineered co-evolution can be seen at work in Figure 6: a rule is partially generated

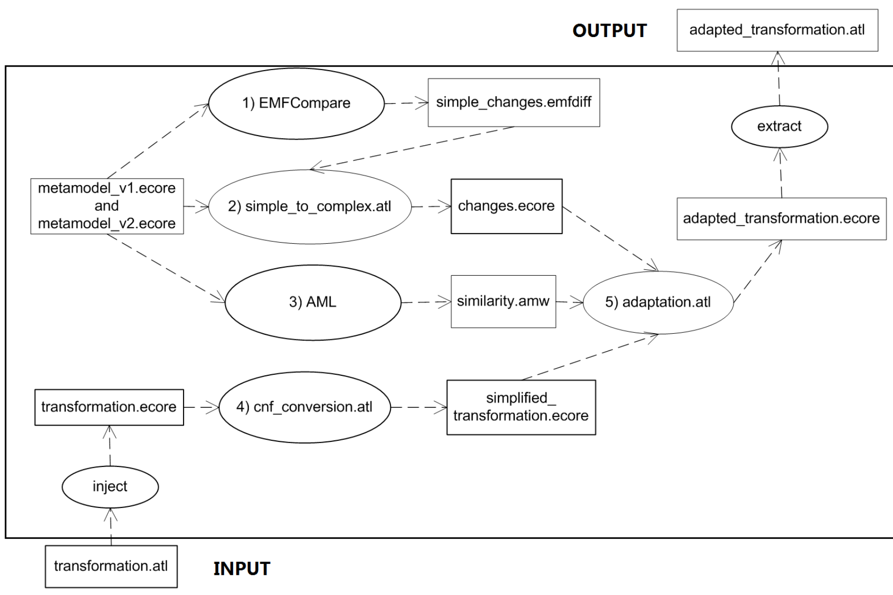


Fig. 7. CO-EUR architecture

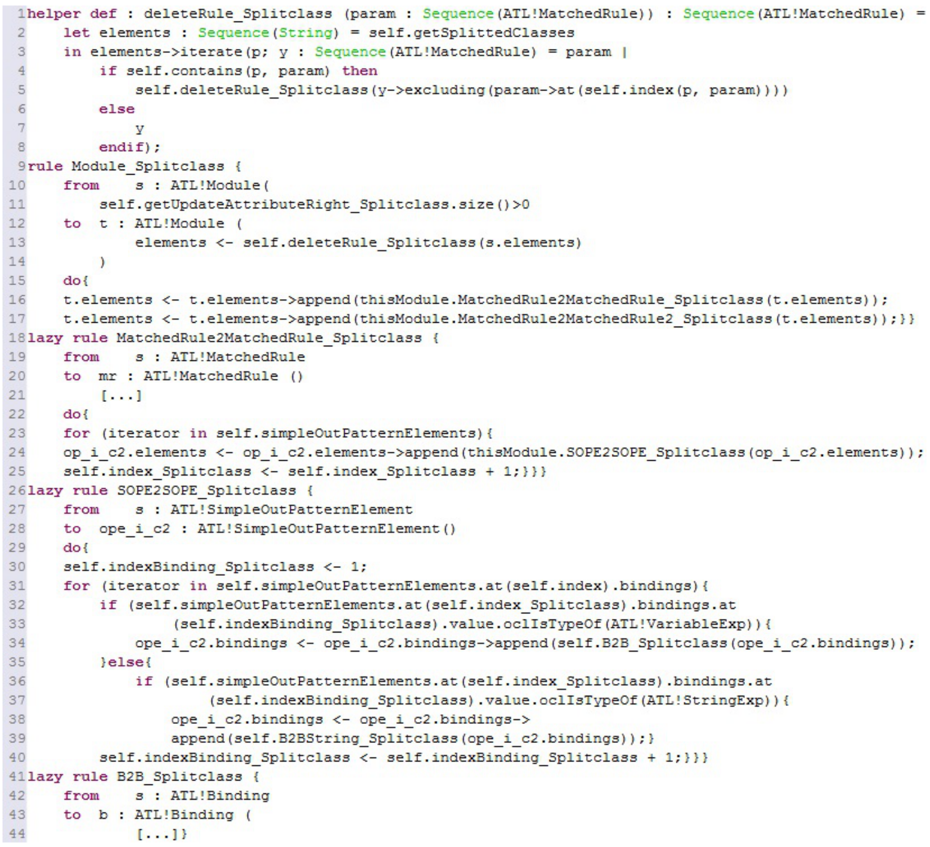
to tackle the addition of a new source metaclass while a new partial binding is proposed to address new properties. In the latter case, only a simple binding is generated (e.g. *target_ metaproperty* ← *source_ metaproperty*) which needs to be completed by the designer (in the example, *xml.style*).

6 Implementation

The CO-EUR prototype is available[2] as a proof-of-concept of the feasibility of this approach for ATL rule co-evolution. Figure 7 depicts the main CO-EUR modules that mimic the co-evolution workflow introduced in Section 2. CO-EUR takes an ATL file (.atl), two Ecore metamodels (.ecore) as input, and returns an ATL file that tackles the differences between the input Ecore models.

The main effort was devoted to the adaptation module. Implementation wise, this module also represents the main innovative approach since transformation co-evolution is achieved using HOT transformations. Along the MDE *motto*: "*everything is a model*" [2], transformations are models that conform to their own metamodel (i.e., the transformation language). Being models, *(Higher Order)* transformations can be used to map the original transformation model into a co-evolved transformation model that caters for the metamodel changes. Figure 8 outlines one such HOT transformation that tackles the *splitClass* case. The pattern includes: a "main" rule, some lazy rules that are called from it

[2] `www.onekin.org/downloads/public/examination-assistant.rar`



```
 1helper def : deleteRule_Splitclass (param : Sequence(ATL!MatchedRule)) : Sequence(ATL!MatchedRule) =
 2    let elements : Sequence(String) = self.getSplittedClasses
 3    in elements->iterate(p; y : Sequence(ATL!MatchedRule) = param |
 4        if self.contains(p, param) then
 5            self.deleteRule_Splitclass(y->excluding(param->at(self.index(p, param))))
 6        else
 7            y
 8        endif);
 9rule Module_Splitclass {
10    from    s : ATL!Module(
11        self.getUpdateAttributeRight_Splitclass.size()>0
12    to  t : ATL!Module (
13            elements <- self.deleteRule_Splitclass(s.elements)
14        )
15    do{
16    t.elements <- t.elements->append(thisModule.MatchedRule2MatchedRule_Splitclass(t.elements));
17    t.elements <- t.elements->append(thisModule.MatchedRule2MatchedRule2_Splitclass(t.elements));}}
18lazy rule MatchedRule2MatchedRule_Splitclass {
19    from    s : ATL!MatchedRule
20    to  mr : ATL!MatchedRule ()
21        [...]
22    do{
23    for (iterator in self.simpleOutPatternElements){
24    op_i_c2.elements <- op_i_c2.elements->append(thisModule.SOPE2SOPE_Splitclass(op_i_c2.elements));
25    self.index_Splitclass <- self.index_Splitclass + 1;}}}
26lazy rule SOPE2SOPE_Splitclass {
27    from    s : ATL!SimpleOutPatternElement
28    to  ope_i_c2 : ATL!SimpleOutPatternElement()
29    do{
30    self.indexBinding_Splitclass <- 1;
31    for (iterator in self.simpleOutPatternElements.at(self.index).bindings){
32        if (self.simpleOutPatternElements.at(self.index_Splitclass).bindings.at
33                (self.indexBinding_Splitclass).value.oclIsTypeOf(ATL!VariableExp)){
34            ope_i_c2.bindings <- ope_i_c2.bindings->append(self.B2B_Splitclass(ope_i_c2.bindings));
35        }else{
36            if (self.simpleOutPatternElements.at(self.index_Splitclass).bindings.at
37                    (self.indexBinding_Splitclass).value.oclIsTypeOf(ATL!StringExp)){
38                ope_i_c2.bindings <- ope_i_c2.bindings->
39                append(self.B2BString_Splitclass(ope_i_c2.bindings));}
40        self.indexBinding_Splitclass <- self.indexBinding_Splitclass + 1;}}}
41lazy rule B2B_Splitclass {
42    from    s : ATL!Binding
43    to  b : ATL!Binding (
44            [...]}
```

Fig. 8. HOT rules to cope with class split. HOTs' input and output models conform to the ATL metamodel.

to create elements, and some helpers to modularize the functionality. In this specific case, *Module_ Splitclass* is the "main" rule (line 9), which will be executed when there is any change of type *Splitclass*. In the *to part* of the rule, the helper *deleteRule_ Splitclass* is called (line 13) which causes the deletion of the rule referring to the deleted metaclass. Then, the imperative part of the rule (*do*) creates two new rules: *MatchedRule2MatchedRule_ Splitclass* and *MatchedRule2MatchedRule2 _ Splitclass* (lines 16 and 17). These rules in turn refer to rule *SOPE2SOPE _ Splitclass* (line 24), which creates a *SimpleOutPatternElement* for the new generated rules. Finally, this rule invokes *B2B_ Splitclass* to geneate the bindings (lines 34 and 39).

7 Related Work

Although co-evolution of models after metamodel evolution has been widely studied [4,9,16], transformations have raised less attention. A lot of research has been carried out in the model co-evolution area and some proposals have been done to semi-automatically adapt models to metamodel evolution. Three main strategies have been used [10]: *(i)* manual specification: these approaches provide transformation languages to manually specify the migration (e.g. [16]); *(ii)* matching approaches: they intend to automatically derive a migration from the matching between two metamodel versions (e.g. [4]); and *(iii)* co-evolution based on operators: they record the coupled operations which are used to evolve the metamodel and which also encapsulate a model migration (e.g. [9]). Following this classification, our approach would be in the second type, as we do not know changes in advance or make them in any specific tool. But on the other hand, we rely our complex changes in a taxonomy of operators based on the third type ([9]). Our approach is similar, as each change has an associated co-evolution, but the difference is that we do not create explicitly the operators, as they are automatically derived. In some cases changes in metamodels do not affect transformations, as studied in [18], where authors conclude that the addition of new classes and broadening of multiplicity constraints do not break the subtyping relationship between metamodel versions. But often changes do have an impact on transformations. To the best of our knowledge, two authors ([14] and [17]) have dealt with transformation co-evolution. The first case is limited to graph-based languages, considering simple changes and considering subtractive changes only as coarse-grained removals (i.e., rule level deletions). In contrast, we focus on rule-based declarative languages, deleting as little as possible, and considering complex changes. In [17] authors explain a fundamental idea, e.g., the convenience of using operators in the co-evolution of transformations. Compared to this publication, our contribution would be an automatic conversion from simple to complex changes, minimum deletion and an implementation of co-evolutions in ATL. First issue of the approach, the conversion of simple to complex changes is treated in [8] and [21]. The former is based on a DSL for expressing model matching and the later uses a sequence of operator instances as evolution trace, and they allow to make changes over changes.

Our co-evolution process only guarantees that the transformation is syntactically correct, and if other correctness properties need to be checked, other complementing works will have to be considered, as analysis and simulation [1], testing [12] or metamodel coverage [23]. In the case where co-evolution is done manually, coverage analysis can be used to determine whether the changes to a metamodel affect the transformation [20].

8 Conclusions

We addressed how metamodel evolution can be semi-automatically propagated to the transformation counterpart. The process flow includes: (1) detecting simple changes from differences between the original metamodel and the evolved metamodel, (2) deriving complex changes from simple changes, (3) translating boolean expressions to the CNF form, (4) if available, capitalize on model similarity, and finally, (5) map the original transformation into an evolved transformation that (partially) tackles the evolved metamodel. The approach is realized for EMOF/ Ecore-based metamodels, and ATL transformations. The approach relieves domain experts from handling routine cases so that they can now focus on the more demanding scenarios (e.g. additive evolution). The use of high-level transformations implies the existence of a transformation metamodel. So far this is available for main transformation languages such as ATL or RubyTL.

Acknowledgments. We thank Jordi Cabot, Arantza Irastorza and reviewers for their help. This work has been partially supported thanks to a doctoral grant Jokin enjoys from the Basque Government under the "Researchers Training Program". This work is co-supported by the Spanish Ministry of Education, and the European Social Fund under contract TIN2011-23839 *(Scriptongue).*

References

1. Anastasakis, K., Bordbar, B., Küster, J.M.: Analysis of Model Transformations via Alloy. In: Baudry, B., Faivre, A., Ghosh, S., Pretschner, A. (eds.) Proceedings of the 4th MoDeVVa Workshop Model-Driven Engineering, Verification and Validation, pp. 47–56 (2007), http://kyriakos.anastasakis.net/prof/pubs/modevva07.pdf
2. Bézivin, J.: In Search of a Basic Principle for Model-Driven Engineering. UPGRADE, The European Journal for the Informatics Professional, Special Issue on UML and Model Engineering 5(2), 21–24 (2004)
3. Cabot, J., Conesa, J.: Automatic Integrity Constraint Evolution due to Model Subtract Operations. In: Wang, S., Tanaka, K., Zhou, S., Ling, T.-W., Guan, J., Yang, D.-Q., Grandi, F., Mangina, E.E., Song, I.-Y., Mayr, H.C. (eds.) ER Workshops 2004. LNCS, vol. 3289, pp. 350–362. Springer, Heidelberg (2004)
4. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: Automating Co-evolution in Model-Driven Engineering. In: Enterprise Distributed Object Computing Conference (2008)
5. Di Ruscio, D., Iovino, L., Pierantonio, A.: What is Needed for Managing Co-evolution in MDE? In: Proc. of the 2nd International Workshop on Model Comparison in Practice, IWMCP 2011, pp. 30–38. ACM, New York (2011)
6. France, R., Rumpe, B.: Model-Driven Development of Complex Software: A Research Roadmap. In: Workshop on the Future of Software Engineering (FOSE 2007), at the 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, Minnesota, USA, pp. 37–54 (2007)
7. Garcés, K., Jouault, F., Cointe, P., Bézivin, J.: A Domain Specific Language for Expressing Model Matching. In: Proc. of the 5ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM 2009) (2009)
8. 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)
9. Herrmannsdoefer, M., Vermolen, S., Wachsmuth, G.: An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models. In: Software Language Engineering, Third International Conference, Software Language Engineering 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers (2011)
10. Herrmannsdoerfer, M.: COPE – A Workbench for the Coupled Evolution of Metamodels and Models. In: Malloy, B., Staab, S., van den Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 286–295. Springer, Heidelberg (2011)

11. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A Model Transformation Tool. Science of Computer Programming (SCP) 72(1-2), 31–39 (2008)
12. Küster, J.M., Abd-El-Razik, M.: Validation of Model Transformations – First Experiences Using a White Box Approach. In: Kühne, T. (ed.) MoDELS 2006 Workshops. LNCS, vol. 4364, pp. 193–204. Springer, Heidelberg (2007)
13. Kurtev, I.: Adaptability of Model Transformations, ch. 5. PhD thesis, University of Twente, Enschede (May 2005)
14. Levendovszky, T., Balasubramanian, D., Narayanan, A., Karsai, G.: A Novel Approach to Semi-automated Evolution of DSML Model Transformation. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 23–41. Springer, Heidelberg (2010)
15. Mohagheghi, P., Dehlen, V.: Where Is the Proof? - A Review of Experiences from Applying MDE in Industry. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 432–443. Springer, Heidelberg (2008)
16. 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)
17. Schätz, B., Deridder, D., Pierantonio, A., Sprinkle, J., Tamzalit, D.: On the Use of Operators for the Co-Evolution of Metamodels and Transformations. In: Proc. of the International Workshop on Models and Evolution (ME 2011) at MoDELS 2011, pp. 54–63 (2010)
18. Steel, J., Jézéquel, J.: On Model Typing. Software and System Modeling 6(4), 401–413 (2007)
19. Toulmé, A.: Presentation of EMF Compare Utility. In: Eclipse Modeling Symposium 2006, pp. 1–8 (2006)
20. van Amstel, M.F., van den Brand, M.G.J.: Model Transformation Analysis: Staying Ahead of the Maintenance Nightmare. In: Cabot, J., Visser, E. (eds.) ICMT 2011. LNCS, vol. 6707, pp. 108–122. Springer, Heidelberg (2011)
21. Vermolen, S.D., Wachsmuth, G., Visser, E.: Reconstructing Complex Metamodel Evolution. In: Sloane, A., Aßmann, U. (eds.) SLE 2011. LNCS, vol. 6940, pp. 201–221. Springer, Heidelberg (2012)
22. Wachsmuth, G.: Metamodel Adaptation and Model Co-adaptation. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 600–624. Springer, Heidelberg (2007)
23. Wang, J., Kim, S., Carrington, D.: Verifying Metamodel Coverage of Model Transformations. In: Proc. of the Australian Software Engineering Conference, pp. 270–282. IEEE Computer Society, Washington, DC (2006)

Guided Merging of Sequence Diagrams[⋆]

Magdalena Widl[1], Armin Biere[3], Petra Brosch[2], Uwe Egly[1], Marijn Heule[4], Gerti Kappel[2], Martina Seidl[3], and Hans Tompits[1]

[1] Institute of Information Systems, Vienna University of Technology, Austria
{uwe,tompits,widl}@kr.tuwien.ac.at

[2] Business Informatics Group, Vienna University of Technology, Austria
lastname@big.tuwien.ac.at

[3] Institute for Formal Models and Verification, Johannes Kepler University, Austria
firstname.lastname@jku.at

[4] Department of Computer Science, University of Texas, Austin, United States
firstname@cs.utexas.edu

Abstract. The employment of *optimistic model versioning systems* allows multiple developers of a team to work independently on their local copies of a software model. The merging process towards one consolidated version can be error-prone and time-consuming when performed without any tool support. Recently, several sophisticated approaches for model merging have been presented. However, even for multi-view modeling languages like UML, which distribute the information on the modeled system over different views, these views are merged independently of each other. Hence, inconsistencies are likely to be introduced into the merged model. We suggest to solve this problem by exploiting information stored in one view as constraint for the computation of a consolidated version of another view. More specifically, we demonstrate how state machines can guide the integration of parallel changes performed on a sequence diagram. We give a concise formal description of this problem and suggest a translation to the satisfiability problem of propositional logic.

1 Introduction

At least since Brooks' 1987 publication on software engineering, awareness has been brought to the collective consent that software is inherently complex [8]. According to Brooks, this complexity can be split into *essential complexity* introduced by the problem domain itself and *accidental complexity* emerging from inadequate representations of the problem domain. Essential complexity is enclosed in the very nature of software, and can thus hardly be reduced. To mitigate accidental complexity, software engineering practice is shifting from code-centric development to a model-driven engineering (MDE) [5] paradigm, which is based on multi-view modeling languages like the Unified Modeling Language (UML). In MDE, software models are not only employed as informal design sketches, but serve as first-class development artifacts used for automatic

[⋆] This work was supported by by the Vienna Science and Technology Fund (WWTF) through project ICT10-018, by the fFORTE WIT Program of TU Wien and the Austrian Federal Ministry of Science and Research, by the Austrian Science Fund (FWF) under grants P21698, J3159-N23, and S11409-N23, and by DARPA contract number N66001-10-2-4087.

code generation. UML introduces different views on the system under development in order to make the complexity of large systems manageable. These views are represented as different diagrams, each highlighting a certain aspect of the system while abstracting from others. For example, the internal behavior of objects is shown in state machines whereas interactions between objects are specified by sequence diagrams.

However, not only software itself, but also the process of building software is inherently complex. Already 40 years ago [21], software engineering was defined as multi-person construction of multi-version software. The combination of multiple persons and multiple versions of software is thus, in addition to the complexity of the software itself, another important source of complexity. Consequently, tools supporting team work and change management emerged [14], in particular, *version control systems* (VCS). Two different versioning paradigms are distinguished. On the one hand, *pessimistic versioning systems* grant exclusive access to a resource by locking it for all but one developer, with the consequence that no conflicts are possible, but also all but one developer are interrupted in their work. On the other hand, *optimistic versioning systems* manage parallel modifications of a software artifact by comparing and merging independently evolved versions with a common ancestor. In the rest of this paper, we consider optimistic versioning.

Initially, VCS were applied only to textual artifacts such as source code, but with the increasing importance of software models in the software engineering process, the need to version control also the modeling artifacts became evident. However, due to the graph-based nature of models, existing VCS, which have been successfully employed on source code, are only of limited value for model versioning. Thus, dedicated model versioning systems based on different algorithms are necessary. Several approaches have been presented recently [10] for both single-view modeling languages and multi-view modeling languages like UML. As far as version control for multi-view models is concerned, however, current approaches merge each diagram individually and ignore valuable information spread across different diagrams. By ignoring this information, false conflicts may be reported or unsatisfactory merge results may be returned, giving rise to inconsistencies between different views of the software model.

In this paper, we propose an approach in which information distributed over the state machine and the sequence view of a model is taken into account when merging sequence diagrams. In particular, we show how two versions of a sequence diagram can be consistently merged by taking the behavior expressed by state machines into account. Since the merged version is not unique in general, the goal is to precalculate a set of consistent merges to support the modeler in integrating the modifications. Given a multi-view modeling language with several concepts as found in UML, we give a formal specification of the merging problem, which allows for a direct encoding to a formalism for which tool support is available. We chose propositional logic as host language to represent the merging problem of sequence diagrams in terms of a satisfiability problem (SAT) [12] because the required constraints are directly transferable and powerful solvers are available.

This paper is structured as follows. In Section 2, we discuss an example illustrating the problems occurring during the merge of sequence diagrams. We review related work in Section 3. In Section 4, we give an overview of the modeling language concepts

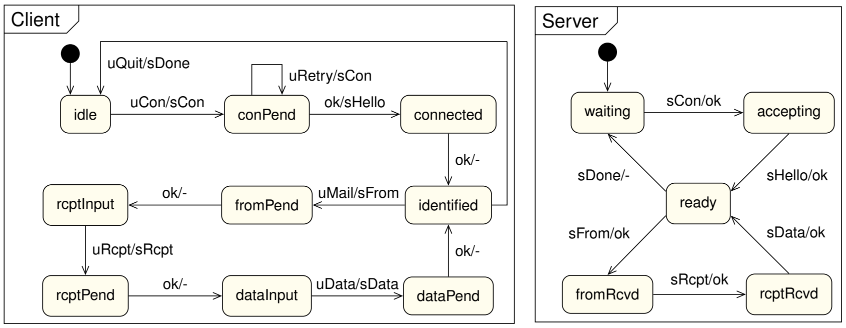


Fig. 1. State machines of an email client and an email server

considered in this paper in terms of a graphical metamodel, which we then transform to a formal representation. On this basis, we give a concise definition of the sequence diagram merging problem in Section 5. The translation of this problem to a satisfiability problem of propositional logic is explained in Section 6. We present our implementation and an evaluation in Section 7. Finally, we conclude and give an outlook to future work.

2 A Motivating Example

The following example about an email protocol is to motivate the approach developed in this paper. Figure 1 shows state machines of an email-client and an email-server implementing a simplified variant of the *Simple Mail Transfer Protocol* (SMTP). Only basic sending functionality is realized and no error handling is included. The initial state of each state machine is indicated by an incoming arrow from a black circle. States are connected to each other by transitions. Each transition carries a label that consists of two parts, separated by a "/". The string on the left indicates a *trigger*, whose receipt in the source state of the transition causes the state machine to change its state to the target state of the transition. The string on the right of each transition indicates a set of *effects*, which are symbols that are sent when the transition is executed and which may again trigger state transitions in the same or other state machines. For example, the state machine Client starts in state Idle and waits until it receives uCon, which causes its transition to state conPend. During the execution of the transition it sends the trigger sCon, which is received by the state machine Server and causes its transition from state waiting to accepting. During the execution of this transition Server triggers ok, which is again received by Client, triggering the transition to state connected, and so on.

A valid model may only partially specify the system under consideration. For example, the first transition on Client is triggered by a user, for which no state machine is defined. In this case, an unconstrained state machine is assumed. Such a state machine contains only one state from which any symbol is received and sent.

Communication scenarios between users, clients, and servers are modeled by *sequence diagrams* showing sequences of exchanged messages. Sequence diagrams describe interactions where the interaction partners, called *lifelines*, are instances of state machines. Figure 2 shows three sequence diagrams SD^o, SD^α, and SD^β. The lifelines are represented by labeled rectangles and vertical dashed lines. Each label contains the name of the lifeline on the left of the colon (e.g., c) and the name of the state machine instantiated by the lifeline on the right of the colon (e.g., Client). A message is represented as arrow connecting two lifelines and contains a symbol which can be found as effect on some transition of the sender's state machine and as trigger on some transition of the receiver's state machine. A sequence diagram is consistent with the state machines that are instantiated by its lifelines if for each lifeline the sequence of received messages is a path of triggers in the corresponding state machine. For example, for the uppermost diagram in Figure 2, SD^o, the sequence of received messages for lifeline c:Client causes triggers uCon $\rightarrow$ ok $\rightarrow$ ok. This sequence is also found as a path in the corresponding state machine, namely connecting states idle $\rightarrow$ conPend $\rightarrow$ connected $\rightarrow$ identified. A similar argument holds for s:Server. Since for u:User we assume an unconstrained state machine, no restriction is imposed on the order of the messages received by u:User, so this lifeline is also consistent.

Consider the following evolution scenario. Starting from the sequence diagram SD^o of Figure 2, which shows the authentication process of an email protocol, two modelers, Alice and Bob, independently perform some modifications. Alice extends the scenario with a logout message resulting in the revised sequence diagram SD^α, while Bob adds the communication necessary to send an email, manifesting in revision SD^β. Trying to merge the modifications of both modelers without any additional information, it is not automatically decidable in which order the added messages from both revisions should be arranged. Hence, we have a *merge conflict*.

It thus has to be decided manually how the changes are integrated. Several syntactically correct merges of the sequence diagram are possible, namely all possible permutations of the two concatenated sequences that preserve the relative order of the messages. However, many of these options turn out to be inconsistent with the state machines. When taking the state machines into account, then only one merged version is possible: Alice's modifications have to be appended after Bob's changes, otherwise the sequence diagram would model a scenario which is forbidden by the state machines.

3 Related Work

The requirements of model versioning systems strongly diverge from the requirements of traditional versioning systems for text-based artifacts like source code [2,3,4]. In consequence, several approaches to conflict detection algorithms and model merging strategies have been presented over the last few years (cf. the survey article by Brosch et al. [10]). These approaches are either realized on the generic metamodeling level resulting in language independent solutions or use language specific information in order to yield better support for a specific modeling language. Our approach falls into the latter category. However, we are not aware of any approach dealing with the merge problem of sequence diagrams where the sequence diagrams have to be kept consistent with the state machine view. Westfechtel [28] discusses the merge of ordered features in EMF models by aggregating elements into linearly ordered clusters. The order within a cluster is determined either at random or by a user. Since the merge is performed on the metamodel level to keep the approach generic, the information available within the

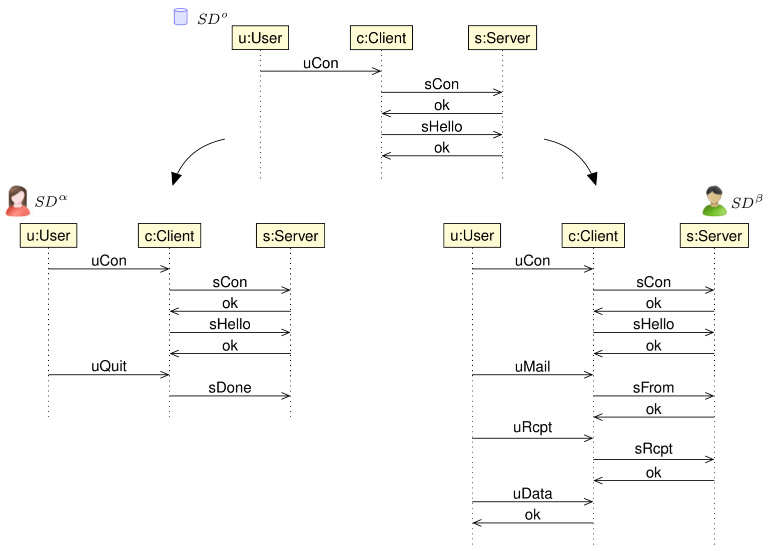


Fig. 2. Evolution of a sequence diagram

model cannot be used for merging. Gerth et al. [18] provide dedicated merge support for business process models ensuring a consistent outcome. They formalize process models as process terms and use a term rewriting system to detect and interactively resolve merge conflicts. However, there is no support for calculating all valid merge solutions. Cicchetti et al. [13] propose to define conflict patterns which can be tailored towards the application on sequence diagrams. Such a conflict pattern can be equipped with a reconciliation strategy for resolving the conflict. Nejati et al. [19] present an approach for merging two state machines. This approach exploits syntactical as well as semantical information provided by the models in order to compare variants and perform consistency checks.

Outside the research of model versioning, several approaches have been presented to verify the consistency of different views of a model and to eliminate inconsistencies. Diskin et al. [15] present a framework based on category theory for consistency checking of views. They first integrate the relevant parts of the metamodels into one global metamodel such that all instance models become instance models thereof. These instance models can then be checked for inconsistencies. Van Der Straeten et al. [26] use the SAT-based constraint solver *Kodkod* to detect and resolve inconsistencies between class and sequence diagrams. Egyed [16] proposes to identify inconsistencies in an incremental manner. Sabetzadeh et al. [23] present an approach to check consistency between a set of different, but overlapping models. Therefore, they merge this set of models to one model. Tsiolakis [25] suggests to collect constraints distributed over views like the class diagram or state machines and integrate them in the sequence diagram in terms of state invariants yielding pre- and postconditions for individual messages.

In the context of model versioning, these approaches can be used to check whether the merged version introduces inconsistencies, i.e., to perform quality control on the

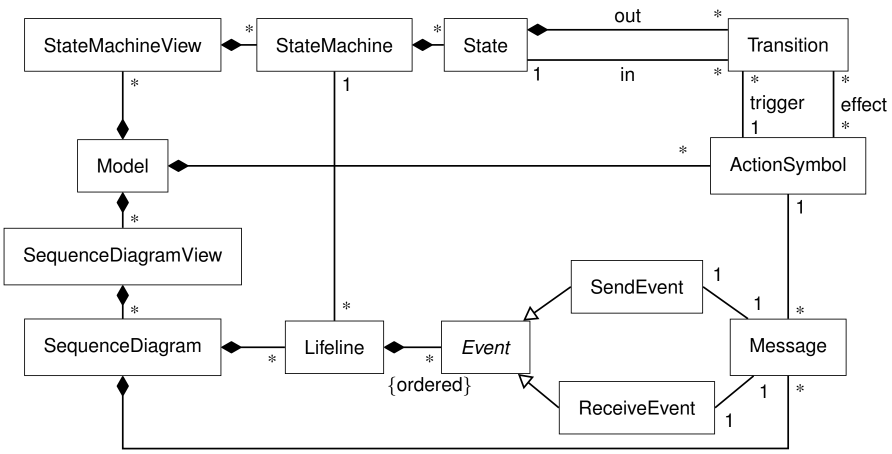


Fig. 3. Metamodel of the *tMVML*

merge result. In previous work [9], we propose to use model checking to validate the merged version of an evolving sequence diagram. No support for the merging process itself is provided.

4 The Modeling Language *tMVML*

In order to give a concise definition of the sequence diagram merging problem, an explicit statement on the modeling language concepts is essential. Therefore, we define the modeling language *tMVML*, the *Tiny Multi-View Modeling Language* in terms of a metamodel. The concepts and terminology of *tMVML* are inspired by the Unified Modeling Language (UML) of the OMG [20]. The compactness of the metamodel allows not only a focused presentation of our approach, but also a direct technical realization as discussed in Section 7. Concepts to describe the static structure of a system as found in a class diagram and specification facilities for behavior as offered by the activity diagram are not relevant for this work and therefore omitted. However, interfaces to other views and advanced concepts of state machines and sequence diagrams not discussed in this paper are prepared for later versions of *tMVML*.

With the same motivation as in other works on the formalization of UML [17], we then present a formalization of the concepts of *tMVML* suitable for our purposes. This formalization enables us to precisely define the sequence diagram merging problem in the context of model versioning.

4.1 The *tMVML* Metamodel

The implementation of the *tMVML* metamodel is available at our project website [1]. We consider the excerpt relevant for this work, which is depicted in Figure 3. For better readability we typeset instances of metaclasses in standard lowercase font using the same name as the metaclass, e.g., in order to to refer to an instance of the metaclass State we simply write "state".

The *tMVML* metamodel has a root class Model which contains two classes representing views, SequenceDiagramView and StateMachineView, and the class ActionSymbol. Action symbols realize the communication between different state machines and describe communication in sequence diagrams. The StateMachineView contains a set of state machines, each containing a set of states. States are connected by transitions. To each transition an action symbol can be assigned as trigger and one or more action symbols can be assigned as effects. The SequenceDiagramView contains a set of sequence diagrams. A sequence diagram consists of a set of lifelines and a set of messages. Each message carries an action symbol and is attached via a send event and a receive event to one or two lifelines. Each lifeline consists of a sequence of events. Messages are ordered relative to the lifelines they are attached to. In this work, we only consider synchronous message passing, for which a message order relative to the sequence diagram suffices. However, on order to stay close to the definition of sequence diagrams in the UML standard and to be able to extend our approach to asynchronous message passing and timed events, we base our approach on ordered events rather than on ordered messages.

In the following, we formalize the metamodel of *tMVML* as is required for a concise specification of the sequence diagram merging problem.

4.2 Formalization of the *tMVML* Metamodel

Let $\mathcal{L}_\mathcal{A}$ be the language describing *tMVML* models defined over the alphabet $\mathcal{A} = (\mathcal{A}_S, \mathcal{A}_A, \mathcal{A}_L, \mathcal{A}_M, \mathcal{A}_E)$ where $\mathcal{A}_S$ denotes a set of states, $\mathcal{A}_A$ denotes a set of action symbols, $\mathcal{A}_L$ denotes a set of lifelines, $\mathcal{A}_M$ denotes a set of messages, and $\mathcal{A}_E$ denotes a set of events. The class Model is the root element of the metamodel and contains the classes ActionSymbol, StateMachineView, and SequenceDiagramView. Each instance of ActionSymbol is an element of $\mathcal{A}_A$. The elements composing the classes StateMachineView and SequenceDiagramView, sets of state machines respectively sequence diagrams, are defined in the following along with their components and associations, Besides the language concepts and their interplay, we introduce and define important properties of a sequence diagram, namely *well-formedness*, *time consistency*, *lifeline conformance*, and *correctness*, which are required to formulate the sequence diagram merging problem.

By $\mathcal{P}(X)$ we refer to the power set of a set X and for any tuple $Y = (y_1, \ldots, y_n)$, by $\pi_i(Y) = y_i$ with $1 \leq i \leq n$, we refer to the projection to the i-th element. We continue to typeset instances of metaclasses in standard lowercase font.

Definition 1 (State machine). *Given the alphabet $\mathcal{A}$, a state machine is a quadruple (S, A^{tr}, A^{eff}, T), where*

- *$S \subseteq \mathcal{A}_S$ is a set of states,*
- *$A^{tr}, A^{eff} \subseteq \mathcal{A}_A$ are sets of action symbols, and*
- *$T \subseteq (S \times A^{tr} \times \mathcal{P}(A^{eff}) \times S)$ is a relation representing the transitions between states.*

A state machine consists of a set of states, two alphabets, and transitions between states. For a transition $t \in T$ with $t = (s, a, A, s')$, s is the source state of the transition, s' the target state, a an action symbol that, when received, triggers the execution of the transition, and A is a set of action symbols that are sent when the transition is executed. In Figure 1, state machine Server contains states $S = \{\mathsf{waiting}, \mathsf{accepting}, \mathsf{ready}, \mathsf{fromRcvd}, \mathsf{rcptRvd}\}$, triggers $A^{tr} = \{\mathsf{sCon}, \mathsf{sDone}, \mathsf{sFrom}, \mathsf{sRcpt}, \mathsf{sData}, \mathsf{sHello}\}$, and effect $A^{\mathit{eff}} = \{\mathsf{ok}\}$. Examples for transitions are $(\mathsf{waiting}, \mathsf{sCon}, \{\mathsf{ok}\}, \mathsf{accepting})$ and $(\mathsf{ready}, \mathsf{sDone}, \{\}, \mathsf{waiting})$. The formalization of the communication between state machines is not relevant for this problem formulation and is therefore omitted. It can be found in previous work [9].

Definition 2 (Sequence diagram). *Given the alphabet $\mathcal{A}$ and a set $\mathcal{SM}$ of state machines, a sequence diagram is a quadruple $(L, M, \mathsf{lprop}, \mathsf{msg})$, where*

- $L \subseteq \mathcal{A}_L$ *is a set of lifelines,*
- $M \subseteq \mathcal{A}_M$ *is a set of messages,*
- $\mathsf{lprop} : L \to (\mathcal{SM} \times \mathcal{P}(\mathcal{A}_E) \times \mathcal{P}(\mathcal{A}_E) \times \mathcal{P}(\mathcal{A}_E \times \mathcal{A}_E))$ *describes lifelines, and*
- $\mathsf{msg} : M \to (\mathcal{A}_A \times \bigcup_{l \in L} \pi_2(\mathsf{lprop}(l)) \times \bigcup_{l \in L} \pi_3(\mathsf{lprop}(l)))$ *describes messages.*

For a lifeline l with $\mathsf{lprop}(l) = (SM, E^{snd}, E^{rcv}, >)$, SM is the state machine associated to l, E^{snd} and E^{rcv} are sets of send and receive events handled by l, and the relation $>$ describes the $\{ordered\}$ constraint of the association between the classes lifeline and event in the *tMVML* metamodel. We assume that

- E^{snd} and E^{rcv} are disjoint,
- the relation $>$ is transitive, antisymmetric, and irreflexive,
- for all $(e_1, e_2) \in >$, it holds that $e_1, e_2 \in E^{snd} \cup E^{rcv}$, and
- for two lifelines, the sets of send and receive events are pairwise disjoint.

For a message m with $\mathsf{msg}(m) = (a, s, r)$, a is the action symbol, s the send event, and r the receive event associated to m. We assume that

- for each message, its send event is distinct from its receive event
- for any two messages, their send events are pairwise distinct
- for any two messages, their receive events are pairwise distinct

Figure 4 shows the sequence diagram SD^o of Figure 2 indicating additional elements described in Definition 2. Usually, much information is omitted in the concrete syntax as in Figure 2 to avoid an information overflow for the human reader. In the sequence diagram SD^o, the set L contains the lifelines u, c, and s. Lifelines c and s are instances of the state machines Client and Server of Figure 1. The state machine for u, User, is not shown. Each message is depicted as arrow between two lifelines. Each arrowhead represents a receive event and each arrowtail a send event. Sender and receiver lifelines may be identical. In Figure 4, lifeline s handles events $\mathsf{sConRcv1}$, $\mathsf{okSnd1}$, $\mathsf{sHelloRcv1}$, and $\mathsf{okSnd2}$, hence $\mathsf{lprop}(\mathsf{s}) = (\mathsf{Server}, \{\mathsf{okSnd1}, \mathsf{okSnd2}\}, \{\mathsf{sConRcv1}, \mathsf{sHelloRcv1}\}, >)$ with $\mathsf{sConRcv1} > \mathsf{okSnd1} > \mathsf{sHelloRcv1} > \mathsf{okSnd2}$. Further, $\mathsf{msg}(\mathsf{m2}) = (\mathsf{sCon}, \mathsf{sConSnd1}, \mathsf{sConRcv1})$.

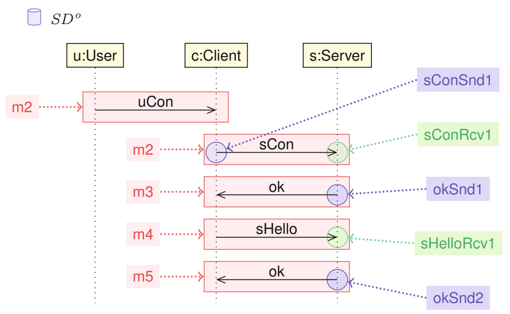


Fig. 4. Sequence diagram SD^o from Figure 2 indicating messages and events

We use the following functions to refer to elements of a sequence diagram: Given the alphabet $\mathcal{A}$ and a sequence diagram $SD = (L, M, \mathsf{lprop}, \mathsf{msg})$, let $\mathsf{lprop}(l) = (SM_l, E_l^{snd}, E_l^{rcv}, >_l)$ for each $l \in L$ and let $E = \bigcup_{l \in L}(E_l^{snd} \cup E_l^{rcv})$. Then we have:

– $\mathsf{act} : M \to \mathcal{A}_A$, $\mathsf{snd} : M \to \mathcal{A}_E$, and $\mathsf{rcv} : M \to \mathcal{A}_E$, such that $\mathsf{act}(m) = \pi_1(\mathsf{msg}(m))$, $\mathsf{snd}(m) = \pi_2(\mathsf{msg}(m))$, and $\mathsf{rcv}(m) = \pi_3(\mathsf{msg}(m))$, i.e. the action symbol, send event and receive event of a message.
– $\mathsf{symb} : E \to \mathcal{A}_A$ is a partial function such that $\mathsf{symb}(e) = a$ iff
  - $e \in E_l^{snd}$ for some $l \in L$ and there exists an $m \in M$ with $\mathsf{act}(m) = a$ and $\mathsf{snd}(m) = e$, or
  - $e \in E_l^{rcv}$ for some $l \in L$ and there exists an $m \in M$ with $\mathsf{act}(m) = a$ and $\mathsf{rcv}(m) = e$,

  i.e., the action symbol of the message an event is associated to. Note that each function value is unique due to the pairwise disjointness of sets of events on lifelines and distinctness of events on messages as described in Definition 2.
– $\mathsf{life} : E \to L$ such that $\mathsf{life}(e) = l$ iff $e \in \pi_2(\mathsf{lprop}(l)) \cup \pi_3(\mathsf{lprop}(l))$. Note that each function value is unique due to the pairwise disjointness of sets of events on lifelines as described in Definition 2.

We define properties of sequence diagrams to specify correct merge results: First, the *well-formedness* of a sequence diagram enforces an order on the events w.r.t. a lifeline.

Definition 3 (Well-Formedness). *A sequence diagram* $(L, M, \mathsf{lprop}, \mathsf{msg})$ *is* well-formed *iff for each* $l \in L$ *the relation* $\pi_4(\mathsf{lprop}(l))$ *is total.*

This total order over events per lifeline imposes a relation over the messages of a sequence diagram, which we need for the second property: A sequence diagram is *time consistent* when any message m is not received after a message n if m has been sent before n on the same lifeline, or in other words, messages cannot overtake one another. We first define the *message ordering* relation over a sequence diagram, which describes an order of a sequence diagram's messages according to the order of its events.

Definition 4 (Message Ordering). *Given a well-formed sequence diagram of the form* $(L, M, \mathsf{lprop}, \mathsf{msg})$*, the* message ordering *relation* $\succ \subseteq M \times M$ *contains a pair* (m, n)

iff for $\mathsf{msg}(m) = (a_m, s_m, r_m)$, $\mathsf{msg}(n) = (a_n, s_n, r_n)$, $l = \mathsf{life}(s_m)$, $>_l = \pi_4(\mathsf{lprop}(l))$, $k = \mathsf{life}(r_m)$, *and* $>_k = \pi_4(\mathsf{lprop}(k))$ *it holds that*

- $\mathsf{life}(s_n) = l$ *and* $s_m >_l s_n$,
- $\mathsf{life}(r_n) = l$ *and* $s_m >_l r_n$,
- $\mathsf{life}(s_n) = k$ *and* $s_m >_k s_n$, *or*
- $\mathsf{life}(r_n) = k$ *and* $s_m >_k r_n$.

In SD^o of Figure 2, the message order is given by $\mathsf{m1} \succ \mathsf{m2} \succ \mathsf{m3} \succ \mathsf{m4} \succ \mathsf{m5}$.

Definition 5 (Time Consistency). *A well-formed sequence diagram is called time consistent iff the transitive closure of its message ordering relation* $\succ$ *is antisymmetric.*

The third property is called *lifeline conformance* and concerns the lifelines of a sequence diagram and the state machines modeling their behavior. Roughly, a lifeline l is conformant with the state machine SM defined in $\pi_1(\mathsf{lprop}(l))$, if the sequence of action symbols of the messages received by l occurs as a path of triggers in SM.

Definition 6 (Lifeline Conformance). *Let*

- $SD = (L, M, \mathsf{lprop}, \mathsf{msg})$ *be a well-formed, time consistent sequence diagram,*
- $l \in L$ *be a lifeline with* $\mathsf{lprop}(l) = (SM, E^{snd}, E^{rcv}, >_l)$,
- $SM = (S, A^{tr}, A^{eff}, T)$ *be a state machine modelling the behavior of* l, *and*
- $(e_1, \ldots, e_n)$ *be the sequence of events where for all* i, j *with* $1 \leq i, j \leq n$ *it holds that* $e_i, e_j \in E^{rcv}$ *and* $e_i >_l e_j$ *iff* $i > j$.

Then, the lifeline l *is* conformant *to* SM *iff there exists a sequence of transitions* $(s_1, a_1, A_1, s_2), (s_2, a_2, A_2, s_3), \ldots, (s_n, a_n, A_n, s_{n+1})$ *such that* $a_i = \mathsf{symb}(e_i)$.

In Figure 4, consider lifeline s of sequence diagram SD^o. The state machine defined for s is Server, shown in Figure 1. The sequence e for s is $(\mathsf{sConRcv1}, \mathsf{sHelloRcv1})$. The sequence resulting from the action symbols connected to these events, $(\mathsf{sCon}, \mathsf{sHello})$, can be found as path of triggers in Server, namely connecting the states waiting to accepting and accepting to ready. The lifeline s is therefore conformant with its state machine. If the action symbol of m4 was sData instead of sHello, then s would not be conformant, as from the only state that can be reached by a transition triggered by sCon there is no outgoing transition triggered by sData. Note that effects are not considered because they do not change the state of their sender.

Finally, a sequence diagram is *correct*, if it has the three discussed properties.

Definition 7 (Correctness of a Sequence Diagram). *A sequence diagram SD is* correct *iff*

1. *SD is well-formed;*
2. *SD is time consistent;*
3. *all lifelines of SD are conformant to their state machine.*

This concludes the specification of the relevant language concepts and their properties. We provided a formal description, which will be necessary to define the sequence diagram merging problem in the next section.

5 Problem Definition

In the context of optimistic model versioning, two versions of a concurrently evolved model, the revisions, have to be combined into one consolidated version. We consider the problem of merging two *revisions* of a sequence diagram into a consolidated, correct sequence diagram using information from the original sequence diagram and the associated state machines.

Definition 8 (Revision). *A sequence diagram* $SD^\alpha = (L^\alpha, M^\alpha, \mathsf{lprop}^\alpha, \mathsf{msg}^\alpha)$ *defined over the alphabet* $\mathcal{A}$ *is a* revision *of a correct sequence diagram* $SD^o = (L^o, M^o, \mathsf{lprop}^o, \mathsf{msg}^o)$ *defined over the same alphabet iff*

- $L^o \subseteq L^\alpha$*,* $M^o \subseteq M^\alpha$*,*
- *for each* $l \in L^o$ *it holds that* $\pi_1(\mathsf{lprop}^o(l)) = \pi_1(\mathsf{lprop}^\alpha(l))$ *and* $\pi_i(\mathsf{lprop}^o(l)) \subseteq \pi_i(\mathsf{lprop}^\alpha(l))$ *for each* $i \in \{2, 3, 4\}$
- *for each* $m \in M^o$ *it holds that* $\mathsf{msg}^\alpha(m) = \mathsf{msg}^o(m)$*, and*
- SD^α *is correct.*

In Figure 2, the sequence diagrams SD^α and SD^β are revisions of sequence diagram SD^o. Please note that we consider only additions in this work. We will treat deletions and updates in future work.

In the following, we use the position function pos defined over messages for the integration of two revisions of a sequence diagram. Given a correct sequence diagram $S = (L, M, \mathsf{lprop}, \mathsf{msg})$, $\mathsf{pos} : M \rightarrow \{1, \ldots, |M|\}$ such that for all $m, n \in M$ it holds that $\mathsf{pos}(m) = \mathsf{pos}(n)$ iff $m = n$ and $\mathsf{pos}(m) > \mathsf{pos}(n)$ iff $m \succ n$.

A *consolidated version* of a sequence diagram and two revisions is a correct sequence diagram that contains the messages and lifelines of the original sequence diagram and all added messages and lifelines from the revisions. The order of messages relative to the original diagram and the revisions is maintained. Merging the two revisions, each message can be placed on one of a set of positions. This set of positions is returned by the function allow, which is defined as follows.

Definition 9 (Allowed Positions). *Given three correct sequence diagrams* $SD^x = (L^x, M^x, \mathsf{lprop}^x, \mathsf{msg}^x)$*, for* $x \in \{o, \alpha, \beta\}$*,* SD^α *and* SD^β *being revisions of* SD^o*, and the position function* $\mathsf{pos}^x : M^x \rightarrow \{1, \ldots, |M^x|\}$ *with* $x \in \{o, \alpha, \beta\}$ *let* $M = M^o \cup M^\alpha \cup M^\beta$ *and* $I = \{1, 2, \ldots, |M|\}$*. Then* $\mathsf{allow} : M \rightarrow \mathcal{P}(I)$ *assigns to each message* m *a set of positions, such that*

- *if* $m \in M^o$ *and* $\mathsf{pos}^o(m) = \mathsf{pos}^\alpha(m) = \mathsf{pos}^\beta(m)$ *then* $\mathsf{allow}(m) = \{\mathsf{pos}^o(m)\}$ *(*m *remains at the same position),*
- *if* $m \in M^o$ *and* $\mathsf{pos}^o(m) \neq \mathsf{pos}^\alpha(m)$ *or* $\mathsf{pos}^o \neq \mathsf{pos}^\beta(m)$*, then* $\mathsf{allow}(m) = \{\mathsf{pos}^o(m) + |N|\}$ *where* $N = \{n \in M^\alpha \mid \mathsf{pos}^\alpha(n) < \mathsf{pos}^\alpha(m)\} \cup \{n \in M^\beta \mid \mathsf{pos}^\beta(n) < \mathsf{pos}^\beta(m)\}$
- *if* $m \notin M^o$ *and* $m \in M^x$ *with* $x \in \{\alpha, \beta\}$ *then* $\mathsf{allow}(m) = \{i \in I \mid \mathsf{pos}^x(m) + |N'| \leq i \leq \mathsf{pos}^x(m) + |N''|\}$ *where for* $n' \in M^o$*,* $y \in \{\alpha, \beta\}$*,* $y \neq x$*,* $\mathsf{pos}^x(n') = \max_{n \in M^o \mid \mathsf{pos}^x(n) < \mathsf{pos}^x(m)} \mathsf{pos}(n)$*,* $n'' \in M^o$*, and* $\mathsf{pos}^o(n'') = \mathsf{pos}^o(m) + 1$*,*

$$N' = \begin{cases} \{n \in M^y \mid \mathsf{pos}^y(n) < \mathsf{pos}^y(n')\} & \text{if } \exists\, n' \in M^o \text{ s.t } \mathsf{pos}^x(n') < \mathsf{pos}^x(m) \\ \emptyset & \text{otherwise} \end{cases}$$

and

$$N'' = \begin{cases} \{n \in M^y \mid \mathsf{pos}^y(n) < \mathsf{pos}^y(n'')\} & \text{if } \exists\, n'' \in M^o \text{ s.t. } \mathsf{pos}^x(n'') > \mathsf{pos}^x(m) \\ M^y \setminus M^o & \text{otherwise} \end{cases}$$

Consider the sequence diagrams SD^o, SD^α and SD^β shown in the upper part of Figure 5, where SD^α and SD^β are revisions of SD^o. In SD^α, the message a4, and in SD^β the messages b4 and b5 are added between the original messages o2 and o3. In a merged sequence diagram, each of a4, b4 and b5 must again be placed between o2 and o3. Also, in order to maintain their order from the revisions, b5 must be placed after b4. Similar conditions are given for the messages a5 and b6 inserted after o3. Table 1 shows the values $\mathsf{pos}^x(m)$ and $\mathsf{allow}(m)$ for each message m in the upper part of Figure 5.

If in the merged sequence diagram each message m is placed on one of the positions defined in $\mathsf{allow}(m)$ and exactly one message has been placed at each position, the merged sequence diagram is time consistent. However, in order for the merged sequence diagram to be correct, the messages have to be placed such that the lifelines conform to their state machines. If this is also the case, then the merged diagram is a *consolidated version*, defined as follows.

Definition 10 (Consolidated Version). *Given the correct sequence diagrams* $SD^o = (L^o, M^o, \mathsf{lprop}^o, \mathsf{msg}^o)$, $SD^\alpha = (L^\alpha, M^\alpha, \mathsf{lprop}^\alpha, \mathsf{msg}^\alpha)$, *and* $SD^\beta = (L^\beta, M^\beta, \mathsf{lprop}^\beta, \mathsf{msg}^\beta)$, *where* SD^α *and* SD^β *are revisions of* SD^o, *a consolidated version* $SD^\gamma = (L^\gamma, M^\gamma, \mathsf{lprop}^\gamma, \mathsf{msg}^\gamma)$ *is a sequence diagram such that*

1. $L^\gamma = L^\alpha \cup L^\beta$,
2. $M^\gamma = M^\alpha \cup M^\beta$,
3. *for each* $i \in \{\alpha, \beta, o\}$ *and for each* $m \in M^i$ *it holds that* $\mathsf{msg}^\gamma(m) = \mathsf{msg}^i(m)$,
4. *for each* $l \in L^\gamma$ *and for* $i \in \{\alpha, \beta\}$ *it holds that if* $l \in L^i$ *then*
   - $\pi_1(\mathsf{lprop}^\gamma(l)) = \pi_1(\mathsf{lprop}^i(l))$, *and*
   - $\pi_j(\mathsf{lprop}^\gamma(l)) \subseteq \pi_j(\mathsf{lprop}^i(l))$ *for each* $j \in \{2, 3, 4\}$
5. *for each* $m \in M^\gamma$ *it holds that* $\mathsf{pos}(m) \in \mathsf{allow}(m)$, *and*
6. S^γ *is correct.*

The source of complexity in the computation of a consolidated version arises from the exponential number of possible message orderings under consideration of the allow function and the constraint arising from the lifeline-conformance requirement.

Consider the example shown in Figure 5 and Figure 6. The upper part of Figure 5 depicts an original sequence diagram (SD^o) and two revisions (SD^α and SD^β) with the values of the respective pos^x function. The revised diagrams contain added messages between messages o2 and o3 and at the end of the diagram. The lower part of the figure depicts six different time consistent merged diagrams. Figure 6 shows two state machines describing the behavior of the lifelines. The two rightmost merged diagrams are also consolidated versions, i.e. correct with respect to the state machines depicted in Figure 6. It can be easily checked that the sequence of actions on messages received by lifeline y of the rightmost diagram, (o1, b4, b5, a4, a5) occurs as path in state machine Y and so does the sequence of lifeline x (o2, o3, b6), but in the leftmost diagram, for sequence (o1, a4, b4, b5, a5) of lifeline y this is not the case.

Definition 11 (Merging Problem). *Given a triple* $(SD^o, SD^\alpha, SD^\beta)$ *where* $SD^o = (L^o, M^o, \mathsf{lprop}^o, \mathsf{msg}^o)$, $SD^\alpha = (L^\alpha, M^\alpha, \mathsf{lprop}^\alpha, \mathsf{msg}^\alpha)$, *and* $SD^\beta = (L^\beta, M^\beta, \mathsf{lprop}^\beta, \mathsf{msg}^\beta)$ *are valid sequence diagrams, and* SD^α *and* SD^β *are revisions of* SD^o*, the merging problem is to find a consolidated version of* $SD^o, SD^\alpha, SD^\beta$.

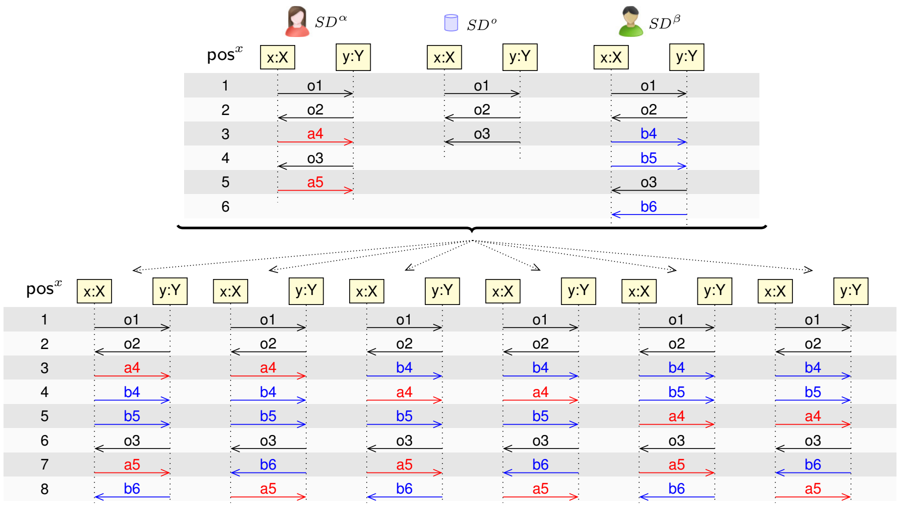


Fig. 5. Sequence diagram (SD^o) and two revisions (SD^α and SD^β) with its six time consistent, but not necessarily lifeline-conformant, merges (below), and the values of the respective pos^x function (left column).

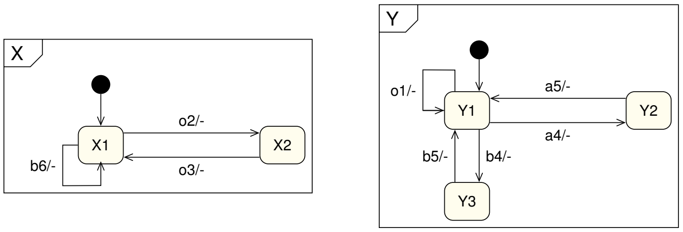


Fig. 6. The state machines modeling the behavior of the lifelines in Figure 5

Table 1. Allowed positions for each message of Figure 5

m	$\mathsf{pos}^o(m)$	$\mathsf{pos}^\alpha(m)$	$\mathsf{pos}^\beta(m)$	$\mathsf{allow}(m)$	
o1	1	1	1	{1}	
o2	2	2	2	{2}	
o3	3	4	5	{6}	$N =$ {a4,b4,b5}
a4	-	3	-	{3,4,5}	$N' = \emptyset$, $N'' =$ {b4,b5}
a5	-	5	-	{7,8}	$N' =$ {b4,b5}, $N'' =$ {b4,b5,b6}
b4	-	-	3	{3,4}	$N' = \emptyset$, $N'' =$ {a4}
b5	-	-	4	{4,5}	$N' = \emptyset$, $N'' =$ {a4}
b6	-	-	6	{7,8}	$N' =$ {a4}, $N'' =$ {a4,a5}

6 Encoding to SAT

We propose to translate the sequence diagram merging problem to a satisfiability problem of propositional logic (SAT) [7]. Over the last years, propositional logic has proven to be a powerful host language for a wide range of real-life problems like verification and planning, not least because the availability of efficient and stable solvers [22]. For our merging problem, we take advantage of this technology. An introduction to propositional logic and the satisfiability problem can be found in many elementary textbooks on logic, for example by Büning and Lettmann [12].

Given an instance $(SD^o, SD^\alpha, SD^\beta)$ of the sequence diagram merging problem, with $SD^x = (L^x, M^x, \mathsf{lprop}^x, \mathsf{msg}^x)$ for $x \in \{o, \alpha, \beta\}$, defined over a set $\mathcal{SM}$ of state machines, let

- $S_{all} = \bigcup_{SM \in \mathcal{SM}} \pi_1(SM)$ be the set of all states of all state machines,
- $T_{all} = \bigcup_{SM \in \mathcal{SM}} \pi_4(SM)$ be the set of all transitions of all state machines,
- $M = M^o \cup M^\alpha \cup M^\beta$ be the set of all messages, and
- $k = |M|$ the total number of messages.

Further, duplicate and rename each state machine referenced by more than one lifeline in a way that $\pi_1(\mathsf{lprop}(l))$ is pairwise distinct for all lifelines. Then the encoding is represented by a non-CNF formula ϕ over three sets of variables as follows.

- $\mathsf{vm} = \{m_i \mid m \in M \wedge i \in \mathsf{allow}(m)\}$. Variables of this set encode the placement of each message at each of its allowed positions. If m_i evaluates to true, it means that message m is placed at position i.
- $\mathsf{vc} = \{c_i^s \mid 1 \leq i \leq k, s \in S_{all})\}$. Variables of this set encode the source state of a state machine for each position before a message is received. If c_i^s evaluates to true, it means that at position i, before the message placed on i is received, the state machine containing s is in state s, or in other words, s is the source state of the transition triggered by the action symbol of the message placed on i.
- $\mathsf{vt} = \{t_i^s \mid 1 \leq i \leq k, s \in S_{all}\}$. Variables of this set encode the target state of a state machine for each position after a message has been received. If t_i^s evaluates to true, it means that at position i, after the message placed on i has been received, the state machine containing s is in state s, or in other words, s is the target state of the transition triggered by the action symbol of the message placed on i.

ϕ is a conjunction of subformulas encoding the requirements given by Definition 10 of a consolidated version SD^γ of sequence diagrams SD^o, SD^α and SD^β.

1. $L^\gamma = L^\alpha \cup L^\beta$,
2. $M^\gamma = M^\alpha \cup M^\beta$,
3. for each $i \in \{\alpha, \beta, o\}$ and for each $m \in M^i$ it holds that $\mathsf{msg}^\gamma(m) = \mathsf{msg}^i(m)$,
4. for each $l \in L^\gamma$ and for $i \in \{\alpha, \beta\}$ it holds that if $l \in L^i$ then
   (a) $\pi_1(\mathsf{lprop}^\gamma(l)) = \pi_1(\mathsf{lprop}^i(l))$, and
   (b) $\pi_j(\mathsf{lprop}^\gamma(l)) \subseteq \pi_j(\mathsf{lprop}^i(l))$ for each $j \in \{2, 3, 4\}$
5. for each $m \in M^\gamma$ it holds that $\mathsf{pos}(m) \in \mathsf{allow}(m)$, and
6. S^γ is well-formed,

7. S^γ is time consistent, and
8. S^γ all lifelines are conformant to their state machines.

The first set of subformulas encodes point 2 (union of messages), point 5 (allow function), and point 6 (well-formedness): Each message must be placed on exactly one of the positions returned by its allow function.

$$\bigwedge_{m \in M} \left(\bigvee_{i \in \mathsf{allow}(m)} m_i \right) \wedge \bigwedge_{m \in M} \bigwedge_{\substack{i,j \in \mathsf{allow}(m) \\ i \neq j}} \left(\neg m_i \vee \neg m_j \right)$$

The next subformula encodes point 4b (order of messages on lifelines) and point 7 (time-consistency): The message order from the revisions is maintained in the consolidated version.

$$\bigwedge_{x \in \{o,\alpha,\beta\}} \bigwedge_{m \in M^x} \bigwedge_{i \in \mathsf{allow}(m)} \left(\neg m_i \vee \bigvee_{\substack{n \in M^x, \\ n \succ m}} \bigvee_{\substack{j > i, \\ j \in \mathsf{allow}(n)}} n_j \right)$$

Finally point 8 (lifeline conformance) is the most difficult. It is encoded as conjunction of the following three subformulas, two of which are non-CNF. In our implementation these formulas are converted to CNF using Tseitin variables [24]. Note that point 4a is implicitly included in this part of the encoding.

1. For each message m and each of its allowed positions one pair of source and target states attached to a transition carrying the message symbol as trigger must be placed on the same position:

$$\bigwedge_{m \in M} \bigwedge_{i \in \mathsf{allow}(m)} \left(\neg m_i \vee \bigvee_{t \in \mathsf{trans}(m)} (c_i^{\pi_1(t)} \wedge t_i^{\pi_4(t)}) \right)$$

   where $\mathsf{trans}(m) = \{t \in \pi_4(\pi_1(\mathsf{lprop}(\mathsf{life}(\mathsf{rcv}(m))))) \mid \pi_1(m) = \pi_2(t)\}$, i.e. all transitions of the state machine instanced by the lifeline that receives the message and carrying the same label as the message.
2. At each position exactly one source state and one target state must be placed:

$$\bigwedge_{i=1}^{k} \left(\left(\bigvee_{c_i^s \in \mathsf{vc}} c_i^s \right) \wedge \left(\bigvee_{t_i^s \in \mathsf{vt}} t_i^s \right) \wedge \bigwedge_{s \in S_{all}} \bigwedge_{r \in S_{all} \setminus s} \left((\neg c_i^s \vee \neg c_i^r) \wedge (\neg t_i^s \vee \neg t_i^r) \right) \right)$$

3. If a state machine stops in state s at position i, then, when it eventually continues at position $i + l$, it must still be in state s. Other state machines may be placed at positions $i + j$, $j < k$.

$$\bigwedge_{i=1}^{k-1} \bigwedge_{M \in \mathcal{SM}} \bigwedge_{s \in \pi_1(SM)}$$

$$\left(\left(t_i^s \rightarrow \bigwedge_{r \in \pi_1(SM) \setminus s} \neg c_{i+1}^r \right) \wedge \left(\bigwedge_{j=1}^{i} \left(t_i^s \wedge \bigwedge_{l=1}^{j} \neg c_l^s \rightarrow \bigwedge_{r \in \pi_1(SM) \setminus s} \neg c_{j+1}^r \right) \right) \right)$$

In this section we clearly see the benefit of the formal specification of the sequence diagram merge problem given in the previous section. On the basis of this specification, the encoding to SAT is straightforward and relies only on standard techniques of modeling with propositional logic. To implement a sequence diagram merging tool, only the mapping to the presented SAT encoding has to be realized. The actual problem solving is completely handed over to a SAT solver.

7 Case Study

The problem description given in Section 5 allows us to encode the merging problem of sequence diagrams as satisfiability problem of propositional logic (SAT). Since the SAT representation of such an encoding can become very large, tool support is needed for the automatic generation of both the translation to SAT and the reverse translation of the models of the SAT problem to the merged diagrams. We implemented a prototype described in the following. With this prototype, we conducted a first case study on a representative benchmark set.

7.1 Implementation

Our prototype implementation is available on our project website [1]. It consists of four modules: *difference provider*, *SAT encoder*, *SAT solver*, and *model merger*. All modules except for the SAT solver are implemented in Java. The difference provider, which is an implementation of the *tMVML* metamodel, and the model merger are based on the Eclipse Modeling Framework (EMF).[1] To solve SAT instances, we use the off-the-shelve SAT solver PicoSAT [6].

The workflow is depicted in Figure 7. First, the *diff provider* takes a *tMVML* conformant sequence diagram and two revisions thereof as input and calculates the set of atomic differences based on EMF Compare's[2] three-way comparison. Even if EMF Compare reports different kinds of atomic changes, such as add, update, or delete, our current implementation is limited to process changes where messages are added to the sequence diagrams. The differences are then analyzed and a table that implements the allow function is created.

The *SAT Encoder* receives as input three sequence diagrams and the table implementing the allow function, and generates the encoding as described in Section 6.

The non-CNF part of the encoding is converted to CNF on-the-fly. As explained in Section 6, each combination of a message m and one of its allowed positions $p \in \mathsf{allow}(m)$ is represented by a boolean variable. This information is maintained in a table that maps message/position pairs to boolean variables and vice versa.

Next, the *SAT solver* processes the generated formula and either returns a logical model consisting of a boolean variable assignment or *unsatisfiable*. If a logical model is found, it is handed over to the *merger*. The merger identifies the variables that encode message/position pairs and maps them back to messages and positions. Based on this

[1] http://www.eclipse.org/modeling/emf/
[2] http://wiki.eclipse.org/EMF_Compare

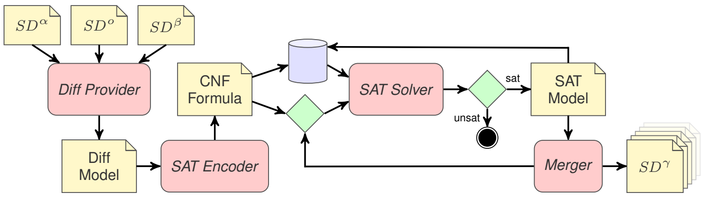


Fig. 7. Implementation workflow

information, a consolidated version conforming to the *tMVML* metamodel is retrieved by copying the origin sequence diagram and inserting the added messages of the two revisions at their respective positions.

Finally, the negation of the logical model is added to the previous encoding. This way we make sure that in the following iteration a different logical model, if one exists, is found. This procedure is repeated until the SAT solver returns *unsatisfiable*, i.e. no more solutions exist.

7.2 Benchmark Set

To study the impact of using information provided by state machines to guide the merging of two differently evolved sequence diagrams, a representative benchmark set is required. Available benchmark sets as presented by Brosch et al. [11] contain only modeling scenarios of a single view and focus on class diagrams. Since versioning systems do not explicitly store the two revisions but only the merged versions, suitable test cases cannot be extracted from available projects. We therefore established our own benchmark set.

The benchmark set consists of three different families, each containing five different versioning scenarios. The first family on a subset of the SMTP protocol is based on the state machines presented in Section 2. The second family models the behavior of a coffee machine and its users similar to the example presented in previous work [9]. Finally, in the third family we model the behavior of the dining philosophers problem, inspired by the running example in the work of Varró [27]. For each versioning scenario, we distinguish between three different cases: (1) All lifelines are fully specified by state machines, (2) some lifelines are specified by state machines, and (3) no lifeline is specified by a state machine. If no state machine is specified for a lifeline, we assume an unconstrained state machine, which contains only one state from which any action symbol can be received. Note that although the models of our benchmark set are formulated in *tMVML*, they can be reused in other case studies because they are realized as Ecore models. Hence, a translation to other modeling languages like full UML can be achieved by the means of model transformations. The benchmark set is available at our project website.

7.3 Evaluation

We tested our approach on 45 test cases consisting of three families, each with five versioning scenarios and three different state machine setups as described in Section 7.2.

Table 2. Statistics on state machines of the benchmark sets

Set	# SM	# action symbols	# states	# transitions
email	3	15	16	19
coffee	2	9	7	8
philosopher	2	8	7	8

Table 3. Overview on benchmarks

Set	ID	SD^o		SD^α		SD^β		full SM		some SM		no SM	
		LL	Ms	LL	Ms	LL	Ms	# Sol	Time	# Sol	Time	# Sol	Time
email	1	3	5	3	7	3	12	1	<1	1	<1	55	3.8
	2	3	5	3	8	3	15	0	<1	2	<1	110	8.0
	3	3	5	3	14	3	14	2	<1	2	<1	1,000	205
	4	3	5	4	14	3	16	2	<1	2	<1	1,000	215
	5	3	5	4	14	3	18	2	1.5	2	<1	1,000	232
coffee	1	2	5	2	6	2	9	2	<1	2	<1	5	<1
	2	2	5	2	6	2	6	0	<1	0	<1	2	<1
	3	2	0	2	2	2	2	2	<1	6	<1	6	<1
	4	2	5	2	9	2	9	2	<1	70	6.0	70	6.0
	5	2	5	2	9	2	9	34	2.9	34	2.9	70	6.0
philosopher	1	4	0	4	2	4	5	6	<1	15	1.4	15	1.8
	2	4	0	4	1	4	5	0	<1	5	<1	5	<1
	3	4	0	4	9	4	9	506	90	1,000	201	1,000	164
	4	4	0	4	9	4	5	253	33	1,000	167	1,000	120
	5	4	0	4	9	4	5	0	<1	1,000	168	1,000	121

Details of the different test cases are shown in Table 2. For each experiment we verified the correctness of the merged version with a verifying tool we implemented. The verification process is trivial: For each lifeline the sequence of received message symbols is retrieved, and then it is checked whether this sequence is found as path of trigger symbols in the corresponding state machine.

Table 3 shows statistics on the number of found solutions and runtime of the different instances. The leftmost columns show the names, number of lifelines (LL), and number of messages (Ms) of each instance, and the three rightmost columns show the number of found solutions (#Sol) and runtime (Time) of each instance. For those instances whose number of solutions exceeded 1,000, we stopped the algorithm when 1,000 solutions were found, as having too many solutions is impractical. The evaluation shows that in general a specification of all state machines results in few solutions quickly found.

For the instances without state machines, we can compute the number of models by $\prod_{f \in F} \frac{(n_f+m_f)!}{n_f! m_f!}$ where F is the set of fragments of the merged model, a fragment being a set of messages with each message inserted between the same two messages of the original diagram, each message inserted at the beginning, or each message inserted at the end of the original diagram. n_f is the number of messages in fragment f inserted from one revision, and m_f the number of messages inserted from the other.

8 Conclusion and Future Work

In this paper, we demonstrated how information encoded in the state machine view of a software model can be used to guide the merging of concurrently evolved versions of a sequence diagram. Such merging support is urgently needed to realize optimistic model versioning systems.

We illustrated our approach for the modeling language *tMVML*, which borrows many concepts from UML. For *tMVML*, we specified a metamodel in Ecore, which provided us with the powerful tool support of the Eclipse environment to build a prototype of our approach. In order to give a formal specification of the merging problem itself, we first formalized the concepts of *tMVML* along with some important properties of the sequence diagram. On this basis, we derived an exact problem specification, which can be directly encoded as a satisfiability problem of propositional logic, the prototypical problem for the complexity class NP. To solve such a problem, highly optimized solving tools, SAT solvers, are available. This way, instead of implementing a complicated merging algorithm, our tool encodes the constraints of a merging problem into a propositional formula, and the computation of a set of consolidated versions, that are correct sequence diagrams merged from two different versions, is done by the SAT solver. From this set of sequence diagrams the software modeler can select a convenient version. By this means, user effort is reduced and merging errors are avoided.

As we showed in our experiments, our approach in general generates too many solutions to be considered, checked, and compared by a human, particularly when the state machines are only partially or not at all specified. Therefore, an automatic approach to decide upon a set of relevant solutions is needed. It is subject to future work to develop ranking and filtering techniques to offer helpful pre-selections. For example, heuristics could be used to avoid unnecessary interleavings of new messages, supposing that the intention of a modeler is to keep the newly introduced sequences together. Further, it should be possible for the modeler to specify additional constraints for the merged model in order to cut down the number of solutions. To determine user intentions, we plan to conduct extensive user experiments.

So far we represent the models only in abstract syntax. However, in an ongoing project we are developing dedicated visualization techniques for sequence diagram merging. First mockups are available at our project website [1].

We aim to extend *tMVML* and plan to consider more concepts like hierarchical states for state machines, or combined fragments for sequence diagrams. At the moment our prototype supports additions only, but in future deletions and updates will be included.

References

1. Project Website on Sequence Diagram Merging(September 2012), `http://www.modelevolution.org/prototypes/sdmerge`
2. Altmanninger, K., Brosch, P., Kappel, G., Langer, P., Seidl, M., Wieland, K., Wimmer, M.: Why Model Versioning Research is Needed!? In: Proc. MoDSE-MCCM Workshop (2009)
3. Barrett, S., Chalin, P., Butler, G.: Model Merging Falls Short of Software Engineering Needs. In: Proc. of the 2nd MoDSE Workshop @ MoDELS 2008 (2008)

4. Bendix, L., Emanuelsson, P.: Requirements for Practical Model Merge – An Industrial Perspective. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 167–180. Springer, Heidelberg (2009)
5. Bézivin, J.: On the Unification Power of Models. SoSyM 4(2), 171–188 (2005)
6. Biere, A.: Picosat essentials. JSAT 4(2-4), 75–97 (2008)
7. Biere, A., Heule, M., van Maaren, H., Walsh, T.: Handbook of Sat. IOS Press (2009)
8. Brooks Jr., F.P.: No Silver Bullet—Essence and Accidents of Soft. Eng. Comp. 20(4) (1987)
9. Brosch, P., Egly, U., Gabmeyer, S., Kappel, G., Seidl, M., Tompits, H., Widl, M., Wimmer, M.: Towards Scenario-Based Testing of UML Diagrams. In: Brucker, A.D., Julliand, J. (eds.) TAP 2012. LNCS, vol. 7305, pp. 149–155. Springer, Heidelberg (2012)
10. Brosch, P., Kappel, G., Langer, P., Seidl, M., Wieland, K., Wimmer, M.: The Past, Present, and Future of Model Versioning. In: Emerging Technologies for the Evolution and Maintenance of Software Models, ch. 15, pp. 410–443. IGI Global (2011)
11. Brosch, P., Langer, P., Seidl, M., Wieland, K., Wimmer, M.: Colex: A Web-based Collaborative Conflict Lexicon. In: Proc. Workshop on Model Comp. in Pract. ACM (2010)
12. Büning, H., Lettmann, T.: Propositional logic. Camb. Univ. Pr. (1999)
13. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: Managing Model Conflicts in Distributed Development. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 311–325. Springer, Heidelberg (2008)
14. Conradi, R., Westfechtel, B.: Version Models for Software Configuration Management. ACM Computing Surveys 30(2), 232–282 (1998)
15. Diskin, Z., Xiong, Y., Czarnecki, K.: Specifying Overlaps of Heterogeneous Models for Global Consistency Checking. In: Dingel, J., Solberg, A. (eds.) MODELS 2010. LNCS, vol. 6627, pp. 165–179. Springer, Heidelberg (2011)
16. Egyed, A.: Instant Consistency Checking for the UML. In: Proc. of the 28th Int. Conf. on Software Engineering (ICSE 2006), pp. 381–390. ACM (2006)
17. France, R.B., Evans, A., Lano, K., Rumpe, B.: The UML as a Formal Modeling Notation. Computer Standards & Interfaces 19(7), 325–334 (1998)
18. Gerth, C., Küster, J., Luckey, M., Engels, G.: Detection and Resolution of Conflicting Change Operations in Version Management of Process Models. In: SoSym, pp. 1–19 (2011)
19. Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., Zave, P.: Matching and Merging of Statecharts Specifications. In: Proc. ICSE 2007, pp. 54–64. IEEE (2007)
20. OMG. Unified Modeling Language (OMG UML), Superstructure V2.4.1 (August 2011), http://www.omg.org/spec/UML/2.4.1/
21. Parnas, D.: Software Engineering or Methods for the Multi-Person Construction of Multi-Version Programs. In: Hackl, C.E. (ed.) IBM 1974. LNCS, vol. 23, pp. 225–235. Springer, Heidelberg (1975)
22. Prasad, M.R., Biere, A., Gupta, A.: A Survey of Recent Advances in SAT-based Formal Verification. STTT 7(2), 156–173 (2005)
23. Sabetzadeh, M., Nejati, S., Liaskos, S., Easterbrook, S.M., Chechik, M.: Consistency checking of conceptual models via model merging. In: Proc. RE 2007. IEEE (2007)
24. Tseitin, G.: On the Complexity of Derivation in Propositional Calculus. Studies in Constructive Mathematics and Mathematical Logic 2(115-125), 10–13 (1968)
25. Tsiolakis, A.: Integrating Model Information in UML Sequence Diagrams. In: Proc. Worksh. ICALP 2001. El. Notes in Theor. Comp. Sc, vol. 50, pp. 268–276. Elsevier (2001)
26. Van Der Straeten, R., Pinna Puissant, J., Mens, T.: Assessing the Kodkod Model Finder for Resolving Model Inconsistencies. In: France, R.B., Kuester, J.M., Bordbar, B., Paige, R.F. (eds.) ECMFA 2011. LNCS, vol. 6698, pp. 69–84. Springer, Heidelberg (2011)
27. Varró, D.: Automated Formal Verification of Visual Modeling Languages by Model Checking. SoSyM 3(2), 85–113 (2004)
28. Westfechtel, B.: A Formal Approach to Three-way Merging of EMF Models. In: Proc. of the 1st Int. Workshop on Model Comparison in Practice @ TOOLS 2010, pp. 31–41. ACM (2010)

Bridging the Chasm between Executable Metamodeling and Models of Computation[⋆]

Benoît Combemale[1], Cécile Hardebolle[2],
Christophe Jacquet[2], Frédéric Boulanger[2], and Benoit Baudry[3]

[1] University of Rennes 1, IRISA
[2] Supélec E3S – Computer Science Department
[3] Inria Rennes - Bretagne Atlantique

Abstract. The complete and executable definition of a Domain Specific Language (DSL) includes the specification of two essential facets: a model of the domain-specific concepts with actions and their semantics; and a scheduling model that orchestrates the actions of a domain-specific model. Metamodels can capture the former facet, while Models of Computation (MoCs) capture the latter facet. Unfortunately, theories and tools for metamodeling and MoCs have evolved independently, creating a cultural and technical chasm between the two communities. Consequently, there is currently no framework to explicitly model and compose both facets of a DSL. This paper introduces a new framework to combine a metamodel and a MoC in a modular fashion. This allows (i) the complete and executable definition of a DSL, (ii) the reuse of a given MoC for different domain-specific metamodels, and (iii) the use of different MoCs for a given metamodel, to account for variants of a DSL.

1 Introduction

Domain-specific languages (DSLs) offer a limited, dedicated set of concepts to domain experts to let them express their concerns about a system. Previous studies have shown that the limited expressiveness of DSLs, combined with dedicated tools, can increase the productivity in the construction of software-intensive systems, while reducing the number of errors [1]. A recent study by Hutchinson *et al.* has even demonstrated that DSLs are one of the main motors for an industrial adoption of model-driven engineering [2].

Defining a DSL completely and precisely is difficult, in particular when it comes to the formal definition of its semantics. However, Bryant *et al.* [3] point out that the formal definition of DSL semantics is the foundation for the major expected benefits of DSLs: the automatic generation of the DSL tooling (*e.g.,* editor and compiler), the formal analysis of model behavior, or the rigorous composition of multiple concerns modeled with different languages.

⋆ This work has been partially supported by VaryMDE, a collaboration between Inria and Thales Research and Technology.

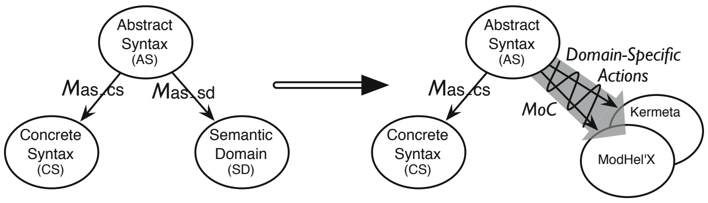


Fig. 1. Our approach to implement the behavioral semantics of a DSL

As described in the left of Figure 1, Harel *et al.* synthesizes the construction of a DSL as the definition of a triple: abstract syntax, concrete syntax and semantic domain [4]. This work focuses on the definition of the abstract syntax (AS), the semantic domain (SD) and the respective mapping between them (M_{as_sd}). Several techniques can be used to define those three elements. This paper focuses on executable metamodeling techniques, which allow one to associate operational semantics to a metamodel. In this context, we argue that the formal definition of the semantic domain must rely on two essential assets: the semantics of domain-specific actions and the scheduling policy that orchestrates these actions. It is currently possible to capture the former in a metamodel and the latter in a *Model of Computation (MoC)*, but the supporting tools and methods are such that it is very difficult to connect both to form a whole semantic domain (see right of Figure 1).

We propose to model domain-specific actions and MoCs in a modular and composable manner, resulting in a complete and executable definition of a DSL. We experiment this proposal by leveraging two state-of-the-art modeling frameworks developed in both communities: the Kermeta workbench [5] that supports the investigation of innovative concepts for metamodeling, and the ModHel'X environment [6] that supports the definition of MoCs. We foresee two major benefits for this composition: the ability to reuse a MoC in different DSLs, and the ability to reuse domain-specific actions with different MoCs to implement semantic variation points of a DSL. Saving the verification effort on MoCs and domain-specific actions also reduces the risk of errors when defining and validating new DSLs and their variants. We illustrate this approach and the reuse capacities through the actual composition of the fUML DSL with a sequential and then a concurrent version of the discrete event MoC.

The rest of the paper proceeds as follows: Section 2 introduces fUML, our case study throughout the paper. Then we describe how to design the domain-specific actions of a DSL and the MoC, respectively using Kermeta (Section 3) and ModHel'X (Section 4). We propose in Section 5 a tool-supported approach to combine them to implement the complete behavioral semantics of a DSL in a modular and reusable fashion. Finally, we present in Section 6 the application of our approach to vary the MoC of fUML. Section 7 presents related work, and Section 8 concludes and proposes directions of future work.

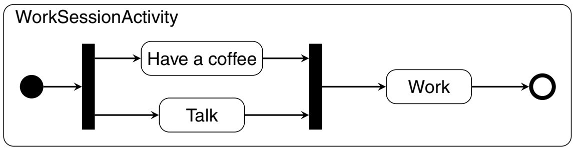


Fig. 2. Activity of the members of our team during our work sessions

2 Case Study: fUML

The *Semantics of a Foundational Subset for Executable UML Models (fUML)* [7] is an executable subset of UML that can be used to define the structural and behavioral semantics of systems. It is computationally complete by specifying a full behavioral semantics for activity diagrams. This means that this DSL is well-defined and enables implementors to execute well-formed fUML models (here *execute* means to actually *run a program*).

As an example, Figure 2 shows an executable fUML model representing the activity of our team when we meet for work sessions. We are used to first having a coffee while talking together about the latest news. When we finish drinking our coffee and talking, we begin to work.

The fUML specification includes both a subset of the abstract syntax of UML, and an execution model of that subset supported by a behavioral semantics. We introduce these two parts of the specification in the rest of this section.

2.1 The fUML Abstract Syntax

Figure 3 shows an excerpt of the fUML metamodel corresponding to the main concepts of the abstract syntax. The core concept of fUML is *Activity* that defines a particular behavior. An *Activity* is composed of different elements called *Activity Nodes* linked by *Activity Edges*. The main nodes which represent the executable units are the *Executable Nodes*. For instance, *Actions* are associated to a specific executable semantics. Other elements define the activity execution flow, which can be either a control flow (*Control Nodes* linked by *Control Flow*) or a data flow (*Object Nodes* linked by *Object Flow*).

The example in Figure 2 uses an illustrative set of elements of the abstract syntax of fUML. The start of the *Activity* is modeled using an *Initial Node*. A *Fork Node* splits the control flow in two parallel branches: one for the *Action* of having a coffee, the other for the *Action* of talking to each other. Then a *Join Node* connects the two parallel branches to the *Action* of working.

Of course, the abstract syntax also includes additional constraints in the metamodel to precise the well-formedness rules (a.k.a. static semantics). For example, such an additional constraint expresses that control nodes can only be linked by control flows. fUML uses the Object Constraint Language (OCL) [8] in order to define those constraints.

We refer the reader to the specification of fUML for all the details about the comparison with UML2 and the whole description of the fUML metamodel [7].

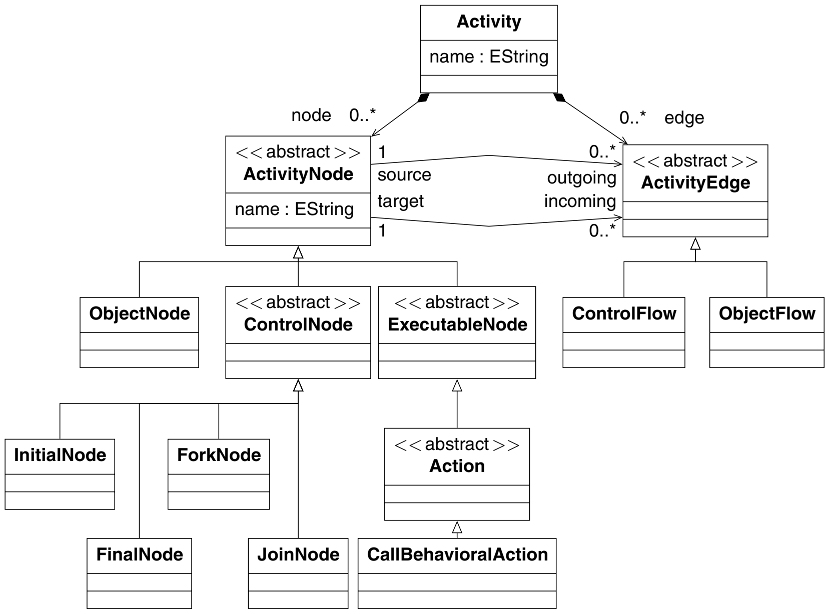


Fig. 3. Excerpt of the fUML Metamodel

2.2 The fUML Behavioral Semantics

To support the execution of models, fUML introduces a dedicated *Execution Model*. The activity execution model has a structure largely parallel to the abstract syntax using the *Visitor* design pattern [9] (called *SemanticVisitor*). Note that although the semantics is explained using visitors, which are rather at the implementation level, it is left open by the fUML specification to implement the language using other means than visitors.

In addition, to capture the behavioral semantics, the interpretation needs to define how the execution of the activity proceeds over time. Thus, concepts are introduced in the execution model for which there is no explicit syntax. Such concepts support the behavior of an activity in terms of *tokens* that may be held by nodes and *offers* made between nodes for the movement of these tokens.

Based on the execution model, the specification denotationally describes the behavioral semantics of fUML using axioms in first order logic. Moreover, a reference implementation of the fUML semantics has been proposed in Java[1]. Both define the *domain-specific actions* (i.e., the behavioral semantics of the domain-specific concepts defined in the abstract syntax) as a concrete implementation of the visitor, including a deeply scattered scheduling of such domain-specific actions. We refer to the latter concern as (part of) the *Model of Computation* (MoC) of the language. Such an implementation prevents the reuse of a MoC for different DSLs (e.g., the fact that all the domain-specific actions should run

[1] Cf. http://portal.modeldriven.org/

in sequence is a behavioral specification that can be reused in many domains), as well as its easy replacement with another one for the same DSL. Indeed, several semantic variation points exist in the MoC. As stated by the specification itself, some semantic areas "are not explicitly constrained by the execution model: The semantics of time, the semantics of concurrency, and the semantics of inter-object communications mechanisms" [7]. We investigate in the rest of this paper an approach to modularly define the domain-specific actions and the MoC of a software language. Such an approach aims at supporting the reuse and the variability in languages, and is illustrated through the fUML case study.

3 Using Kermeta for an Executable Metamodel of fUML

Kermeta is a workbench that can be used for implementing domain-specific languages (DSLs). It supports different meta-languages depending on the DSL concern (abstract syntax, static semantics, behavioral semantics and connection to concrete syntax), and modularization features. In this section, we provide some background on Kermeta, and we show benefits and drawbacks in implementing both the abstract syntax and the behavioral semantics.

3.1 Abstract Syntax Definition

First of all, to build a DSL in Kermeta, one implements its abstract syntax (i.e., the metamodel), which specifies the domain concepts and their relations. The abstract syntax is expressed in an object-oriented manner using Ecore [10], an implementation aligned with the meta-language MOF (*Meta Object Facility*)[11].

MOF provides language constructs for specifying a DSL metamodel: packages, classes, properties, multiple inheritance and different kinds of associations between classes. The semantics of these core object-oriented constructs is close to the object model common to various languages such as Java and C#.

To implement fUML's abstract syntax, we reuse the metamodel standardized by the OMG (cf. Figure 3 for an excerpt). In practice, the OMG provides the fUML metamodel in terms of MOF, and we automatically translate it into an Ecore-based metamodel (the format supported by the Kermeta workbench).

The static semantics of a DSL (i.e. the context conditions) corresponds to the well-formedness rules on top of the abstract syntax (expressed as invariants of metamodel classes) [4]. The static semantics is used to filter syntactically incorrect DSL models before actually running them. Kermeta supports OCL to express the static semantics[2] and it translates this semantics into equivalent Kermeta constraints, directly woven into the relevant metamodel classes using the Kermeta keyword `aspect`. Listing 1.1 shows the well-formedness rule previously introduced for fUML as expressed in the Kermeta workbench using OCL.

[2] Note that Kermeta fully support OCL, and thus similarly supports the axiomatic semantics (expressed as pre- and post-conditions on operations of metamodel classes). It is used to check the correctness of a DSL model's execution either at design time using model-checking or theorem proving, or at runtime using assertions, depending on the execution domain of the DSL.

Listing 1.1. Weaving the Static Semantics of fUML into the Standard Metamodel

```
package fuml;
require "fuml.ecore"
aspect class ControlFlow {
  inv : self.source.oclIsKindOf(ControlNode) and
        self.target.oclIsKindOf(ControlNode)
}
```

In Kermeta, the abstract syntax and the static semantics are conceptually and physically (at the file level) defined in two different modules. Consequently, it is possible to define several variants of the static semantics for the same domain, i.e. to share a single MOF metamodel between different static semantics.

3.2 Behavioral Semantics Definition

To define the behavioral semantics of a DSL, one must first define the required data structure (i.e., the execution model) using MOF. The abstract syntax and the execution model are then the basis to implement the behavioral semantics. Nevertheless, MOF does not include concepts for the definition of the behavioral semantics and OCL is a side-effect-free language. To define the behavioral semantics of a DSL, Kermeta provides an action language [5]. It can be used to define either a translational semantics (for building a compiler) or an operational semantics [12] (for building an interpreter).

The Kermeta language is imperative, statically typed, and includes classical control structures such as blocks, conditional statements, loops and exceptions. It also implements traditional object-oriented mechanisms for handling multiple inheritance and generics, and provides an execution semantics to all MOF constructs that must have a semantics at runtime, such as containment and associations. For example, if a reference is part of a bidirectional association, the assignment operator semantics has to handle both ends of the association. Kermeta also borrows the semantics of multiple inheritance from the Eiffel programming language [13].

Using the Kermeta language, the domain-specific actions are expressed as methods of the classes of the abstract syntax [5]. Similarly to the static semantics, the methods are added to the relevant metamodel classes (using the keyword `aspect`). Unlike the specification and the Java implementation that largely duplicate the structure of the abstract syntax to describe the structure of the visitor (see Section 2.2), aspects avoid duplicating the structure while keeping a conceptual and physical separation. For instance, in Listing 1.2, the method *execute* is added to the concept *CallBehavioralAction* of the fUML abstract syntax. This method is the Kermeta-based specification of the corresponding fUML behavioral semantics that consists in calling the behavior associated to the action.

Listing 1.2. Weaving the Behavioral Semantics of fUML into the Standard Metamodel

```
aspect class CallBehavioralAction {
  operation execute() : Integer is do
    result := self.behavior.call()  // call the associated behavior
  end
}
```

Once the domain-specific actions have been described, it is necessary to describe their scheduling according to a particular model of computation. We describe in the next section the usual way to do this in Kermeta, and we discuss the drawbacks of this approach.

3.3 Mashup of the DSL Concerns

As introduced above, all pieces of static semantics and domain-specific actions are encapsulated in metamodel classes. The `aspect` keyword enables DSL designers to relate the language concerns (abstract syntax, static semantics, and domain-specific actions) together. It allows designers to reopen a previously created class to add some new information such as new methods, new properties or new constraints. It is inspired from open-classes [14].

In addition, Kermeta provides the keyword `require` that one uses to actually *mash up* those concerns. A DSL implementation *requires* an abstract syntax, a static semantics and the domain-specific actions. Listing 1.3 shows how such an implementation looks like in Kermeta. Three `require` keywords are used to import three modules, each of which specifies one of the three concerns. The *require* mechanism also provides some flexibility with respect to the static semantics and the domain-specific actions. For example, several sets of domain-specific actions could be defined in different modules and then chosen depending on particular needs. It is also convenient to support semantic variations of the same concept.

Listing 1.3. Mashup of the fUML Concerns

```
package fuml;
require "fuml.ecore" // abstract syntax
require "fuml.ocl"   // static semantics
require "fuml.kmt"   // domain-specific actions
class Main {
  operation Main(): Void is do
    // Scheduling calls to domain-specific actions
    // to drive the execution of an fUML model
  end
}
```

The Kermeta-based implementation of fUML follows the approach above. All fUML concerns are separated in different units, and the fUML runtime environment is the result of the mashup.

Finally, to implement the entire behavioral semantics, it is necessary to specify how the domain-specific actions are scheduled. One approach is to scatter the scheduling policy across all the methods defining the domain-specific actions in the visitor, as done in the specification and in the Java-based reference implementation. While this approach is easy to implement, it clearly prevents the modularization of the scheduling policy, which would be required to enable its variability. To avoid scattering the scheduling policy, another approach is to extract it in the main class that starts the execution of the model for a particular purpose, and therefore according to a particular MoC (see Listing 1.3). Although this approach of separating the MoC from the domain-specific actions allows the use of the same MoC for variants of the domain-specific actions, the MoC is strongly coupled to the DSL and must be redefined from scratch for every new DSL. It is thus impossible to reuse or to adapt a MoC for different DSLs.

In the next section, we present how the MoC-based modeling framework called ModHel'X can be used to improve the aforementioned approach. This new approach paves the way for reusing the implementation of MoCs. Then, we introduce in Section 5 an approach to combine such a MoC with the domain-specific actions, enabling the reuse of a MoC in different domains, and the implementation of different MoCs for a specific domain.

4 Using ModHel'X Models of Computation for fUML

ModHel'X [6,15] is a framework for building and executing multi-paradigm models, that is to say models built from parts described using different modeling paradigms. In ModHel'X, the behavioral semantics of a modeling paradigm is given by the combination of two elements:

1. A *Model of Computation (MoC)*, which is a set of rules defining the semantics of control and concurrency, the semantics of communications and the semantics of time of the modeling paradigm. Synchronous Data-Flows (SDF), Discrete Events (DE), and Kahn Process Networks (KPN) [16] are examples of models of computation.
2. A library of components with predefined behavior. For instance, the component library of the synchronous data-flow MoC of ModHel'X includes components representing mathematic functions like addition, multiplication, etc. The behavior of those components correspond to the *domain-specific actions* introduced in Section 2.2.

Therefore, building a ModHel'X model is a two-step process: (1) choose components to assemble and (2) choose the MoC according to which the components interact. In the following, we present how MoCs and components are represented in ModHel'X to allow model execution. Then we will show how they can be used in the description of any DSL.

4.1 Generic Abstract Syntax

At the core of ModHel'X is a generic metamodel for describing the structure of models, and a generic execution engine for interpreting such structures using the semantics defined by MoCs. This means that all ModHel'X models have the same abstract syntax (given by the generic metamodel of ModHel'X) but each model may have a different execution semantics depending on the MoC which is used by the execution engine to interpret it. The fact that all models have the same abstract syntax is what allows ModHel'X to support the composition of models which have different semantics. The composition mechanism itself is not presented in this paper because it is not used yet in the presented methodology, but a detailed description can be found in [15].

Figure 4 shows a simplified excerpt of the metamodel of ModHel'X. To illustrate the concepts in this metamodel, we show in Figure 5 how the fUML model of Figure 2 would be described in ModHelX. An element of a ModHel'X model which has a behavior is a *block*, represented as a gray rectangle in Figure 5. Indeed, blocks are the mechanism used in ModHel'X to represent *domain-specific actions*. In the fUML example, ActivityNodes are represented by blocks because they all have a behavior (which can relate to control in the case of ControlNodes, or to executable behaviors in the case of ExecutableNodes).

Blocks communicate with their environment through *pins* (black circles in Figure 5), and the structure of a model is defined by establishing *relations* between the pins of blocks (the lines with arrows on the figure).

There are two different kinds of blocks in ModHel'X. *Interface blocks* are blocks whose behavior is described by an internal ModHel'X model. They are the mechanism we use for supporting heterogeneity through hierarchy (see [15] for more details). *Atomic blocks* are the basic building blocks which are the leaves of the hierarchy of models. For instance, the control nodes of fUML (join and fork in particular), would be atomic blocks in ModHel'X. While we provide libraries of atomic blocks for all the MoCs implemented in ModHel'X, we do not provide specific tools to allow users to define their own domain-specific atomic blocks because it is out of the scope of ModHel'X as a MoC-based experimentation platform for heterogeneous modeling. This means that the behavior of atomic blocks has to be described using a formalism which is external to our framework (for instance C or Java). Therefore, it is interesting for ModHel'X to benefit from techniques such as those provided by Kermeta to allow users to design and specify easily their own domain-specific atomic blocs, i.e. their own domain-specific actions. We will show in Section 5 how the approach proposed in this paper allows the use of Kermeta specifications of the behavior of the ActivityNodes of fUML in ModHel'X.

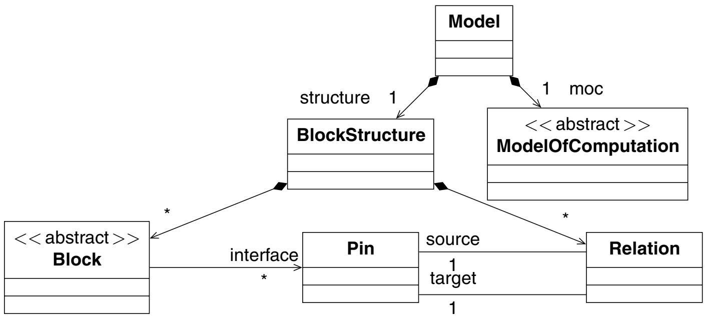


Fig. 4. Simplified excerpt of the generic metamodel of ModHel'X

4.2 Abstract Semantics for Models of Computation

As introduced previously, ModHel'X is dedicated to the execution of models. The execution of a model in ModHel'X is performed by the generic execution

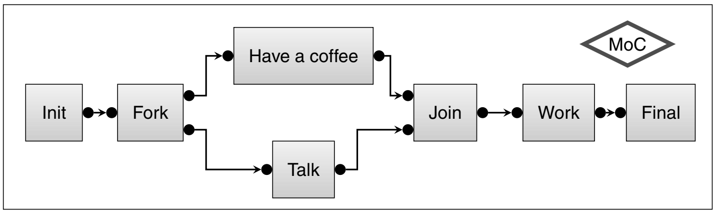


Fig. 5. fUML example model in ModHel'X

engine, which computes a series of observations, or *snapshots*. The MoC of a model is responsible for computing each snapshot of the model according to its specific semantics.

As illustrated by the sequence diagram in Figure 6, to compute a snapshot of a model, the MoC repeatedly chooses a block to observe (*schedule* operation), observes its behavior through its interface (*update*) and propagates observed information between blocks (*propagate*). It repeats this process until the computation of the reaction of the model is complete.

The schedule and propagate operations, as well as the rules for determining the stopping conditions of the algorithm, form the generic interface of MoCs. Together with the generic execution algorithm, they form the *abstract semantics* of ModHel'X. The scheduling and propagation operations must be specified as *MoC-specific actions* for each MoC (see Figure 7) because they are the implementation of the rules that define the control, concurrency, time and communication semantics of the corresponding modeling paradigm. The update operation is delegated to each block to provide the MoC with an observation of its behavior through its interface, while keeping the internal details in a complete black box.

To implement the semantics of fUML in ModHel'X, we would have to choose or to build a MoC which implements the scheduling and propagation policies defined in the specification. Although the specification does not say much about

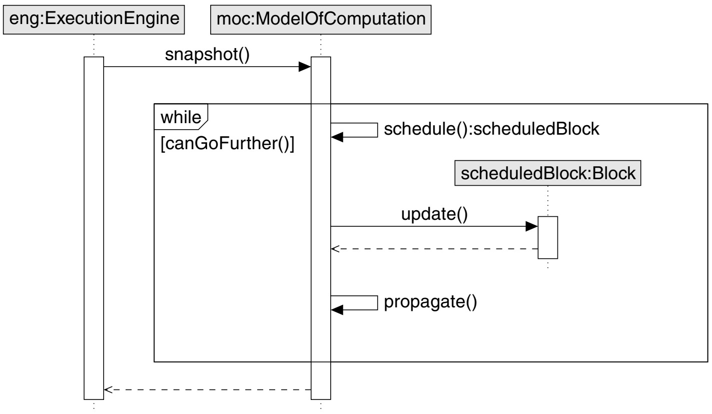


Fig. 6. Abstract semantics of ModHel'X

time, concurrency, and inter-object communications, we know that Activity-Nodes are linked by control and object flows on which tokens are propagated using the mechanism of offers. This is in favor of an event-based model of computation. Moreover, ExecutableNodes can represent the call of actions, so their behavior can take time. We could therefore choose to use the well-known Discrete Events (DE) model of computation [16], which already exists in ModHel'X. We present its semantics in the following section.

4.3 The Discrete Events (DE) MoC

DE is a MoC for the simulation of communicating processes, for instance hosts exchanging messages on a network. In DE, blocks exchange events at given dates. A block is observed when it receives an event from an upstream block or when it has spontaneous behavior. When observed, a block may produce outgoing events, to be transmitted to downstream blocks. If several events have the same timestamp, they are delivered at the same time, but in a sequence of *microsteps* (determined by a topological ordering of the blocks), so that the overall observation at that time is causal and deterministic.

The classical version of DE allows blocks to run concurrently. The scheduling algorithm for DE in ModHel'X relies on a global event queue. At a given instant, the MoC looks for all the events e_i with the smallest time tag t_{now} and advances the current time to t_{now}. It then looks for the blocks b_j which are the targets of the e_i events and schedules one of the minimal elements among the b_j according to the topological ordering of the blocks. The choice of a minimal element guarantees that events produced at t_{now} during the update of a block can be processed by their target at t_{now} in one iteration. A mechanism which is out of the scope of this paper is used to avoid cycles in the graph of blocks. The snapshot is complete when no event with time-stamp t_{now} remains in the queue.

The fUML specification leaves open the type of scheduling of the activity nodes: their execution may be concurrent or sequential. The classical version of DE, the "concurrent" one described above, may therefore be used as *one* possible scheduling for fUML, but a sequential variant of DE is also possible.

We have designed a "Sequential DE" MoC that has the following differences with the "Concurrent DE" MoC. At a given moment, only one block may be *active*. A block is said to be active if it has been given control (i.e. events have been provided to the block and the block has been observed), but it has not released control (i.e. it has not produced events yet). Sequential DE keeps track of the blocks that are *active-able*. If a block receives an event at t, then it is active-able starting at time t. But contrary to Concurrent DE that systematically and immediately activates all the active-able blocks, Sequential DE waits for the currently active block to release control (which will involve taking a new snapshot if the release is not immediate) before activating another active-able block.

The following section shows how the ModHel'X implementation of these two MoCs, "ConcurrentDE" and "SequentialDE", can be used to drive the execution of an fUML model in which the domain-specific actions are described in Kermeta.

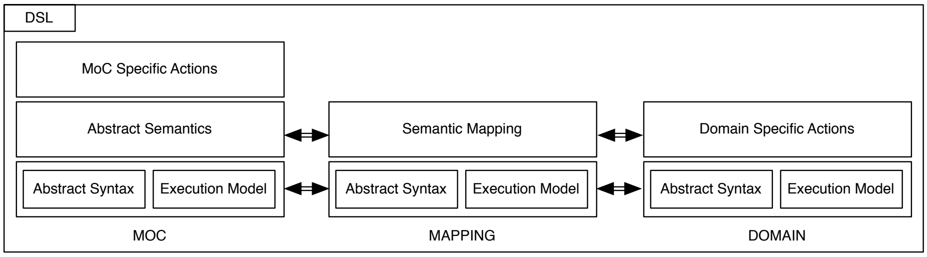


Fig. 7. Elements of the semantics of a DSL in our approach

5 Bridging the Chasm

The previous sections have shown that:

- It is possible to easily define the abstract syntax and the execution model of a DSL, together with the semantics of its domain-specific actions, using Kermeta. However, a model of computation has to be written from scratch for each DSL in order to define the scheduling policy of these actions.
- ModHel'X offers various models of computation and provides means to define customized models of computation on top of an execution engine which allows the simulation of heterogeneous models. However, no specific tool is provided to help the user specify domain-specific actions which are represented as blocks in a ModHel'X model.

In this section, we now bridge the chasm between these two worlds in order to benefit from the advantages of both. As a result, we present a general methodology that allows us to *execute* models described with DSLs defined in a modular way. We present the application of this methodology to the fUML case study. We have also applied the methodology to the Software and Systems Process Engineering Metamodel specification of the OMG [17]. The corresponding experiments are available online at http://www.gemoc.org/kermeta-modhelx.

Our approach is based on the decomposition of a DSL semantics as shown in Figure 7. The structure of the MoC, on the left, and of the domain, on the right, have been presented in the previous sections. In the following sections, we show how the abstract syntax and execution models on both sides can be mapped, and how the abstract semantics of the MoC modeling framework (ModHel'X in our case), combined with the concrete execution semantics of the MoC, is used to schedule domain-specific actions in order to execute a model.

5.1 Abstract Syntax Mapping

First, the abstract syntax of the DSL is mapped onto the abstract syntax of ModHel'X, to enable model execution through the generic engine. In the case of fUML, the control structure and the activity nodes must be mapped onto ModHel'X elements. Activity nodes have domain-specific actions that must be

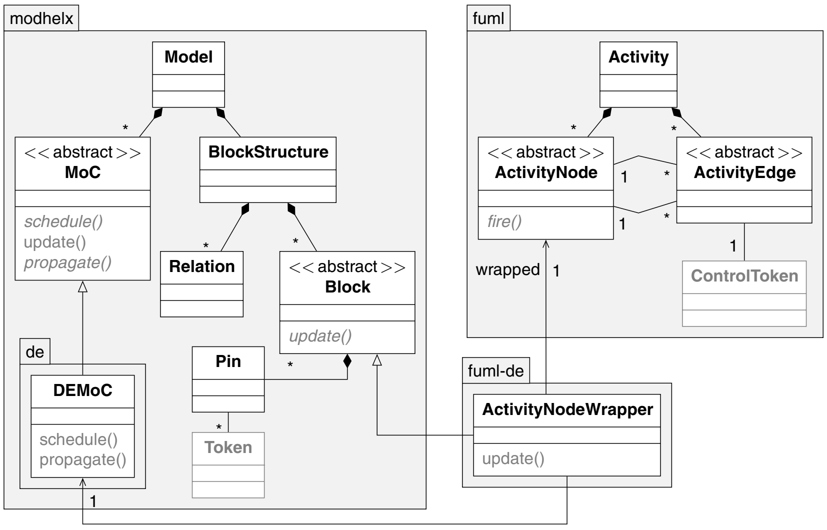


Fig. 8. Mapping (package `fuml-de`) between the kermeta-based implementation of fUML (package `fuml`) and ModHel'X (package `modhelx`) to use any of its MoCs (*e.g.*, here the discrete event MoC, package `de`)

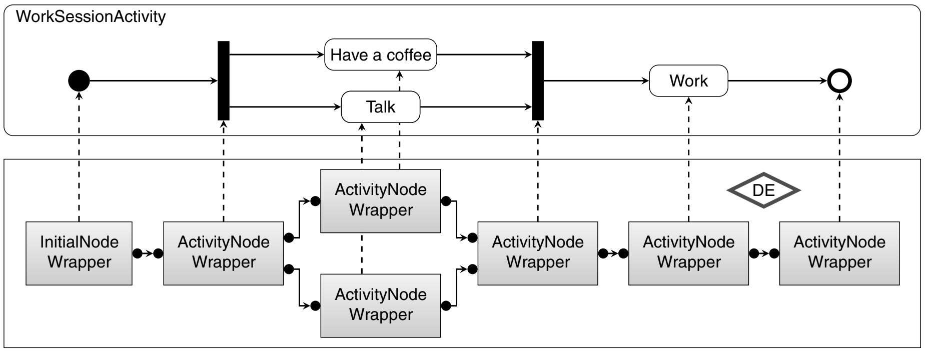


Fig. 9. Example fUML model and its wrapping ModHel'X model using DE

callable, so they are naturally mapped onto atomic blocks (that can be observed through the *update* operation). Control edges are mapped onto relations between blocks, which represent the possible flow of control.

Figure 8 shows the mapping between the two metamodels, and Figure 9 shows the result of the syntactic transformation of an fUML model into a ModHel'X model using the DE MoC. This model transformation is made before the execution starts, by instantiating a wrapper for each activity node. For fUML, we need two kinds of wrappers. *ActivityNodeWrapper* wraps *reactive* activity nodes, i.e. nodes that start a behavior, which possibly takes some time, when they are given control. All activity nodes except *InitialNodes* are of this kind. However *InitialNodes* must *create* *control* at the start of the simulation, without receiving control explicitly. That is why they are wrapped into specific *InitialNodeWrappers*. Then, for each edge in the fUML model, we create a relation between the pins of the relevant blocks in the ModHel'X model. It must be noted that in this case, the mapping between activity nodes and blocks, and between control edges and relations is straightforward. However, in the general case, the structure of the ModHel'X model may be different from the structure of the domain-specific model.

5.2 Abstract Semantics to Domain Specific Actions Mapping

We must now map the abstract semantics of ModHel'X onto the domain-specific actions. The entry point of the abstract semantics for blocks is the *update* operation. Therefore, an activity node is wrapped into a special kind of block, which has an *update* operation that calls the domain-specific actions of the node. The wrapper acts as a block in the ModHel'X model, so its class is a subclass of *Block*. On the other hand it must execute the associated domain-specific actions, so it relies on the DSL's method signatures. Figure 10 shows how the wrapper maps the abstract semantics of ModHel'X onto the domain-specific semantics of fUML. When DE gives control to the wrapper block by calling its *update* method, the wrapper calls the domain-specific action (the *fire* operation). If the wrapped activity node is an action which takes time, the wrapper also requests to be observed in the future, so that it can handle the termination of the action.

The *schedule* and *propagate* operations allow the MoC to choose which block should be updated next, and how information produced by the update should be propagated to the other blocks.

5.3 Execution Model Mapping

The last item of Figure 7 to be mapped is the *execution model*, which represents the state of the execution of the model. The *update* operation of the wrapper synchronizes the execution models on both sides. In the case of the DE MoC and of fUML, DE events represent control on the MoC side, and must be translated into fUML control tokens before the domain-specific actions are called. When the fUML model has updated its execution model, control tokens must be converted into DE events so that the MoC has the necessary information to schedule the rest of the execution. Time must also be synchronized so that the MoC knows when to schedule a block, and activity nodes know when they terminate.

In the general case, the wrapper has to synchronize three aspects of the execution model: control, time and data. In this example, the DE/fUML wrapper adapts control and time only; we did not deal with the adaptation of data in this work.

One of the difficulties of the approach is to decide what to model in the MoC and what to model in the domain-specific actions. In order to favor the modularity and the reuse of the MoC for different DSLs, we decided to handle only the control and time aspects in the MoC and the wrapper. An example of such a design decision is the choice of whether to check in the MoC or in a

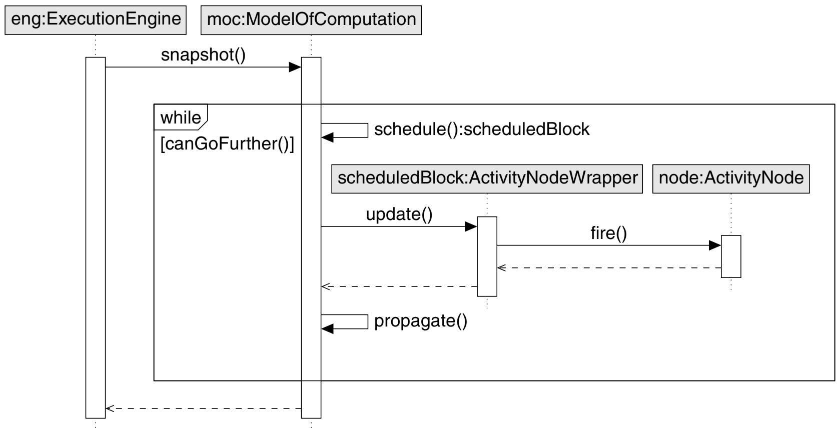


Fig. 10. Mapping fUML domain-specific semantics on ModHel'X abstract semantics

domain-specific action if an activity node can be activated. Both can be done: *the wrapper* can be responsible for calling *fire* only when all the inputs of the block have received an event, or *fire* can be responsible for executing the activity only when all incoming control edges have a control token. We chose to implement the latter behavior in the *fire* domain-specific action even though this is related to control, because it is the *core semantics* of fUML that states that an activity node is executed only when it has received control on all its incoming edges.

6 Implementations and Execution Traces

We have experimented the approach proposed in this paper using Kermeta and ModHel'X to implement fUML[3]. Using our implementation, we have been able to execute the fUML WorkSessionActivity example, wrapped as shown in Figure 9. The following sections present the execution traces obtained using the classical "Concurrent" DE MoC, then its "Sequential DE" variant. To help differentiating the two executions, we have chosen different durations for the *Have a coffee* action (10 minutes), the *Talk* action (15 minutes) and the *Work* action (45 minutes). The execution traces obtained using our implementation are graphically depicted by the timing diagrams shown on Figure 11. Those diagrams illustrate the time at which the different actions respectively start and complete.

6.1 Using the Concurrent DE MoC

The execution obtained using the Concurrent DE MoC is illustrated by the timing diagram shown on the left part of Figure 11. With Concurrent DE, the two actions after the Fork start concurrently at $t = 0$, the beginning of the execution of the overall activity. So a first snapshot is taken at time $t = 0$,

[3] The experiments are provided at http://www.gemoc.org/kermeta-modhelx

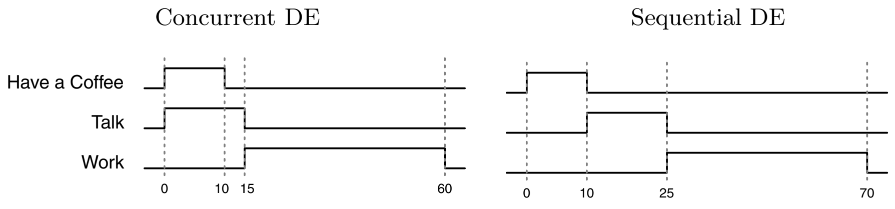


Fig. 11. Timing diagrams of the execution traces when the model is scheduled by different MoCs

and we see on the timing diagram that both the *Have a coffee* and the *Talk* actions start. After that, two more snapshots are taken when each of the actions completes: at $t = 10$ for *Have a coffee*; at $t = 15$ for *Talk*. Within the latter snapshot, the Join is activated since the two preceding actions are finished and it releases control to the *Work* action, which therefore starts at $t = 15$. A last snapshot is taken when the *Work* action completes at $t = 15 + 45 = 60$.

6.2 Using the Sequential DE Variant

The execution obtained using the Sequential DE MoC is illustrated by the timing diagram shown on the right part of Figure 11. With Sequential DE, the two actions after the Fork become active-able at the initial time ($t = 0$). But since only one of them can be active at the same time, the MoC chooses to start one of them, for instance *Have a coffee*. So a first snapshot is taken at time $t = 0$. A second snapshot is taken when the *Have a coffee* action completes ($t = 10$). At that time, the *Talk* action can start. A third snapshot is taken when the *Talk* action completes ($t = 25 = 10 + 15$). During this snapshot, the Join is activated and it releases control to the *Work* action, which therefore starts at $t = 25$. A last snapshot is taken at $t = 25 + 45 = 70$, when the *Work* action completes.

As illustrated by the timing diagrams of Figure 11, we have managed to obtain two different executions of the same fUML model by changing the model of computation which is used to schedule it. This shows how the modular description of the semantics of DSLs as the association of a model of computation and a set of domain-specific actions facilitates the obtention of variants of a given DSL. In the following, we compare our approach to related work in the domains of modeling language engineering and MoC-based modeling.

7 Related Work

Much work have been done on the design and implementation of both software languages and models of computation. In this paper, we propose a conceptual and technical framework to bridge the chasm between them. This framework leverages experiences of both communities. This section presents related work in the field of language design and implementation, and then in the field of models of computation.

The problem of the modular design of languages has been explored by several authors (e.g. [18,19]). For example, JastAdd [19] combines traditional use of higher order attribute grammars with object-orientation and simple aspect-orientation (static introductions) to get a better modularity mechanism. With a similar support for object-orientation and static introductions, Kermeta and its aspect paradigm can be seen as an analogue of JastAdd in the DSML world. Rebernak et al. [20] and Krahn et al. [21] contributed to the field in the context of model-driven DSLs. While they also advocate modularity of DSL compilers and interpreters, we go further: we take advantage of modularity mechanisms for integrating the body of knowledge on models of computation, and allow their reuse and variability.

A language workbench is a software package for designing software languages [22,23]. For instance, it may encompass parser generators, specialized editors, DSLs for expressing the semantics and others. Early language workbenches include Centaur [24], ASF+SDF [25], TXL [26] and Generic Model Environment (GME) [27]. Among more recent proposals, we can cite Metacase's MetaEdit+ [28], Microsoft's DSL Tools [29], Clark et al.'s Xactium [30], Krahn et al's Monticore [21], Kats and Visser's Spoofax [31], Jetbrain's MPS [32]. The important difference of our approach is that we explicitly address the MoC concern in the design of a language, providing a dedicated tooling for its implementation and reuse. Our approach is also 100% compatible with all EMF-based tools (at the code level, not only at the abstract syntax level provided by Ecore), hence designing a DSL with our approach easily allows reusing the rich ecosystem of Eclipse/EMF. This issue was previously addressed in the context of the Smalltalk ecosystem [33]. Our contribution brings in a much more lightweight approach using one dedicated meta-language per language design concern, and providing the user with advanced composition mechanisms to combine the concerns in a fully automated way.

In the context of component-based modeling, models of computation are used to define the interactions between the behavior of the components of a model. [34] describes several characteristics of models of computation, as well as a framework for comparing them.

Several approaches to the definition of models of computation have been proposed. Connector-based approaches like BIP [35] describe the interactions between behaviors using connectors, which can be considered as operators in a process algebra. From the properties of the connectors, it is possible to predict global properties of the models. In the case of BIP, the choice of connectors guarantees that the synthesized controller fires only interactions valid in the model. The result is therefore correct by construction.

The Clock Constraints Specification Language (CCSL) [36] can also be used to describe models of computation. It defines the semantics of the MARTE UML profile and it has been used for instance to model communication patterns in AADL [37]. We also used it in previous work to describe models of computation and the interactions between heterogeneous models of computation [38].

However, these approaches describe how component behaviors are combined in an instance of a model. Other approaches like Ptolemy [16] and ModHel'X [39] allow the definition of models of computation independently of any model instance. Such definitions are therefore reusable for any model which obeys the abstract semantics of the framework. This abstract semantics defines a set of operations which drive the execution of models. Each model of computation provides concrete semantics to these abstract operations. The approach presented in this article relies on such reusable definitions of models of computation.

8 Conclusion and Perspectives

Although previous work has been done on the execution of UML models, as discussed in section 7, to the best of our knowledge, we introduce in this paper the first conceptual and technological bridge between executable metamodeling and models of computation at the level of the metamodels. We leverage on the experience of their respective fields and we provide an approach for a modular design and implementation of executable DSLs.

This approach includes a generic design pattern for metamodels bridging the gap between domain-specific actions woven into the metamodel and a reusable model of computation. We provide an actual implementation of this pattern, using Kermeta to weave executable actions into metamodels, and ModHel'X to schedule their execution according to a reusable MoC. The tools as well as the different fUML bridges presented in the paper can be freely downloaded on line.

Such a modular design and implementation of a behavioral semantics leverages on experience coming from two communities to achieve many expectations. As we illustrate with the fUML example coming from the OMG, many languages have variants of their model of computation, which current implementations do not take into consideration. Moreover, since the correct behavior of models is very dependent on the properties of their MoC, the design and implementation of a MoC can be critical. Being able to reuse validated MoCs, or validating an implementation of a MoC through reuse in various contexts is an advantage. Our approach addresses these two considerations by offering the reuse of MoCs between DSLs. The other way around, being able to reuse the domain-specific actions of a DSL with different MoCs in order to implement semantic variation points is also an advantage.

This first step to combine executable metamodeling and MoCs opens many exciting perspectives that we are proactively exploring. We first plan to examine very carefully the perimeter of the possible wrappers to propose suitable abstractions (*e.g.,* control, time, communication, etc) and patterns for their definition. Then, we would like to fully exploit the benefits coming from the two communities. In particular, we explore the application of this approach for heterogeneous executable modeling, taking advantage of the composition features supported by ModHel'X for multi-paradigm modeling.

References

1. Karna, J., Tolvanen, J.P., Kelly, S.: Evaluating the use of Domain-Specific Modeling in Practice. In: 9th OOPSLA Workshop on Domain-Specific Modeling (2009)
2. Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: Empirical assessment of MDE in industry. In: ICSE, pp. 471–480. ACM (2011)
3. Bryant, B.R., Gray, J., Mernik, M., Clarke, P.J., France, R.B., Karsai, G.: Challenges and directions in formalizing the semantics of modeling languages. Comput. Sci. Inf. Syst. 8(2), 225–253 (2011)
4. Harel, D., Rumpe, B.: Meaningful Modeling: What's the Semantics of "Semantics"? Computer 37(10), 64–72 (2004)
5. Muller, P.-A., Fleurey, F., Jézéquel, J.-M.: Weaving Executability into Object-Oriented Meta-languages. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 264–278. Springer, Heidelberg (2005)
6. Boulanger, F., Hardebolle, C.: Simulation of Multi-Formalism Models with ModHel'X. In: Proceedings of ICST 2008, pp. 318–327. IEEE Comp. Soc. (2008)
7. Object Management Group, Inc.: Semantics of a Foundational Subset for Executable UML Models (fUML), v1.0. (2011)
8. Object Management Group, Inc.: UML Object Constraint Language (OCL) 2.0 Specification (2003)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Professional (1995)
10. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley (2008)
11. Object Management Group, Inc.: Meta Object Facility (MOF) 2.0 Core Specification (2006)
12. Combemale, B., Crégut, X., Garoche, P.L., Thirioux, X.: Essay on Semantics Definition in MDE. An Instrumented Approach for Model Verification. Journal of Software 4(9) (2009)
13. Meyer, B.: Eiffel: the language. Prentice-Hall, Inc. (1992)
14. Clifton, C., Leavens, G.T.: MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In: OOPSLA, pp. 130–145 (2000)
15. Boulanger, F., Hardebolle, C., Jacquet, C., Marcadet, D.: Semantic Adaptation for Models of Computation. In: ACSD, pp. 153–162 (2011)
16. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity – the Ptolemy approach. Proc. of the IEEE 91(1), 127–144 (2003)
17. Object Management Group, Inc.: Software and Systems Process Engineering Metamodel specification (SPEM) Version 2.0. (2008)
18. Van Wyk, E., de Moor, O., Backhouse, K., Kwiatkowski, P.: Forwarding in Attribute Grammars for Modular Language Design. In: Nigel Horspool, R. (ed.) CC 2002. LNCS, vol. 2304, pp. 128–142. Springer, Heidelberg (2002)
19. Ekman, T., Hedin, G.: The JastAdd system – modular extensible compiler construction. Sci. Comput. Program. 69, 14–26 (2007)
20. Rebernak, D., Mernik, M., Wu, H., Gray, J.: Domain-specific aspect languages for modularising crosscutting concerns in grammars. IET Software 3(3), 184–200 (2009)
21. Krahn, H., Rumpe, B., Volkel, S.: MontiCore: Modular Development of Textual Domain Specific Languages. In: Paige, R.F., Meyer, B. (eds.) TOOLS EUROPE 2008. LNBIP, vol. 11, pp. 297–315. Springer, Heidelberg (1974)

22. Fowler, M.: Language workbenches: The killer-app for domain specific languages (2005), http://www.martinfowler.com/articles/languageWorkbench.html (accessed online)
23. Volter, M.: From Programming to Modeling-and Back Again. IEEE Software 28(6), 20–25 (2011)
24. Borras, P., Clement, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B., Pascual, V.: Centaur: the system. In: 3rd ACM Software Engineering Symposium on Practical Software Development Environments, pp. 14–24. ACM (1988)
25. Klint, P.: A meta-environment for generating programming environments. ACM TOSEM 2(2), 176–201 (1993)
26. Cordy, J.R., Halpern, C.D., Promislow, E.: TXL: a rapid prototyping system for programming language dialects. In: Conf. Int Computer Languages, pp. 280–285 (1988)
27. Sztipanovits, J., Karsai, G.: Model-Integrated Computing. IEEE Computer 30(4), 110–111 (1997)
28. Tolvanen, J., Rossi, M.: MetaEdit+: defining and using domain-specific modeling languages and code generators. In: Companion of the 18th Annual ACM SIGPLAN Conference OOPSLA, pp. 92–93. ACM (2003)
29. Cook, S., Jones, G., Kent, S., Wills, A.: Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley Professional (2007)
30. Clark, T., Sammut, P., Willans, J.: Applied Metamodelling – A Foundation for Language Driven Development, 2nd edn (2008)
31. Kats, L.C., Visser, E.: The spoofax language workbench: rules for declarative specification of languages and IDEs. In: OOPSLA 2010, pp. 444–463. ACM (2010)
32. Voelter, M., Solomatov, K.: Language Modularization and Composition with Projectional Language Workbenches illustrated with MPS. In: SLE 2010. LNCS. Springer (2010)
33. Renggli, L., Gîrba, T., Nierstrasz, O.: Embedding Languages without Breaking Tools. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 380–404. Springer, Heidelberg (2010)
34. Lee, E.A., Sangiovanni-Vincentelli, A.L.: A framework for comparing models of computation. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 17(12), 1217–1229 (1998)
35. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time systems in BIP. In: 4th IEEE SEFM, pp. 3–12 (September 2006)
36. Mallet, F., DeAntoni, J., André, C., de Simone, R.: The clock constraint specification language for building timed causality models. Innovations in Systems and Software Engineering 6, 99–106 (2010)
37. André, C., Mallet, F., Simone, R.: Modeling AADL Data Communications with UML MARTE. In: Embedded Systems Specification and Design Languages. Lecture Notes in Electrical Engineering, vol. 10, pp. 155–168. Springer (2008)
38. Boulanger, F., Dogui, A., Hardebolle, C., Jacquet, C., Marcadet, D., Prodan, I.: Semantic Adaptation Using CCSL Clock Constraints. In: Kienzle, J. (ed.) MODELS 2011 Workshops. LNCS, vol. 7167, pp. 104–118. Springer, Heidelberg (2012)
39. Hardebolle, C., Boulanger, F.: Multi-Formalism Modelling and Model Execution. International Journal of Computers and their Applications 31(3), 193–203 (2009);Special Issue on the International Summer School on Software Engineering

Grammatical Inference in Software Engineering: An Overview of the State of the Art

Andrew Stevenson and James R. Cordy

Queen's University, Kingston ON, Canada
{andrews,cordy}@cs.queensu.ca

Abstract. Grammatical inference – used successfully in a variety of fields such as pattern recognition, computational biology and natural language processing – is the process of automatically inferring a grammar by examining the sentences of an unknown language. Software engineering can also benefit from grammatical inference. Unlike the aforementioned fields, which use grammars as a convenient tool to model naturally occuring patterns, software engineering treats grammars as first-class objects typically created and maintained for a specific purpose by human designers. We introduce the theory of grammatical inference and review the state of the art as it relates to software engineering.

Keywords: grammatical inference, software engineering, grammar induction.

1 Introduction

The human brain is extremely adept at seeing patterns by generalizing from specific examples, a process known as inductive reasoning. This is precisely the idea behind grammatical induction, also known as grammatical inference, where the specific examples are sentences and the patterns are grammars. Grammatical inference is the process of identifying an unknown language by examining examples of sentences in that language. Specifically, the input to the process is a set of strings and the output is a grammar.

The main challenge of identifying a language of infinite cardinality from a finite set of examples is knowing when to generalize and when to specialize. Most inference techniques begin with the given sample strings and make a series of generalizations from them. These generalizations are typically accomplished by some form of state-merging (in finite automata), or non-terminal merging (in context-free grammars).

Grammatical inference techniques are used to solve practical problems in a variety of different fields: pattern recognition, computational biology, natural language processing and acquisition, programming language design, data mining, and machine learning. Software engineering, in particular software language engineering, is uniquely qualified to benefit because it treats grammars as first-class objects with an intrinsic value rather than simply as a convenient mechanism to model patterns in some other subject of interest.

Historically there have been two main groups of contributors to the field of grammatical inference: theorists and empiricists. Theorists consider language classes and learning models of varying expressiveness and power, attempting to firm up the boundaries of what is learnable and how efficiently it can be learned, whereas empiricists start with a practical problem and, by solving it, find that they have made a contribution to grammatical inference research.

Grammatical inference is, intuitively as well as provably, a difficult problem to solve. The precise difficulty of a particular inference problem is dictated by two things: the complexity of the target language and the information available to the inference algorithm about the target language. Naturally, simpler languages and more information both lead to easier inference problems. Most of the theoretical literature in this field investigates some specific combination of language class and learning model, and presents results for that combination.

In Section 2 we describe different learning models along with the type of information they make available to the inference algorithm. In Section 3 we explore the learnability, decidability, and computational complexity of different learning models applied to language classes of interest in software engineering: finite state machines and context-free grammars. Section 4 discusses the relationship between theoretical and empirical approaches, and gives several practical examples of grammatical inference in software engineering. In Section 5 we list the related surveys, bibliographies, and commentaries on the field of grammatical inference and briefly mention the emphasis of each. Finally, in Section 6 we discuss the main challenges currently facing software engineers trying to adopt grammatical inference techniques, and suggest future research directions to address these challenges.

2 Learning Models

The type of learning model used by an inference method is fundamental when investigating the theoretical limitations of an inference problem. This section covers the main learning models used in grammatical inference and discusses their strengths and weaknesses.

Section 2.1 describes *identification in the limit*, a learning model which allows the inference algorithm to converge on the target grammar given a sufficiently large quantity of sample strings. Section 2.2 introduces a teacher who knows the target language and can answer particular types of queries from the learner. This learning model is, in many cases, more powerful than learning from sample strings alone. Finally, Section 2.3 discusses the PAC learning model, an elegant method that attempts to find an optimal compromise between accuracy and certainty. Different aspects of these learning models can be combined and should not be thought of as mutually exclusive.

2.1 Identification in the Limit

The field of grammatical inference began in earnest with E.M. Gold's 1967 paper, titled "Language Identification in the Limit" [24]. This learning model provides the inference algorithm with a sequence of strings one at a time, collectively known as a presentation. There are two types of presentation: positive presentation, where the strings in the sequence are in the target language; and complete presentation, where the sequence also contains strings that are not in the target language and are marked as such. After seeing each string the inference algorithm can hypothesize a new grammar that satisfies all of the strings seen so far, i.e. a grammar that generates all the positive examples and none of the negative examples. The term "information" is often used synonymously with "presentation" (e.g. positive information and positive presentation mean the same thing).

The more samples that are presented to the inference algorithm the better it can approximate the target language, until eventually it will converge on the target language exactly. Gold showed that an inference algorithm can identify an unknown language in the limit from complete information in a finite number of steps. However, the inference algorithm will not know when it has correctly identified the language because there is always the possibility the next sample it sees will invalidate its latest hypothesis.

Positive information alone is much less powerful, and Gold showed that any superfinite class of languages cannot be identified in the limit from positive presentation. A superfinite class of languages is a class that contains all finite languages and at least one infinite language. The regular languages are a superfinite class, indicating that even the simplest language class in Chomsky's hierarchy of languages is not learnable from positive information alone.

There has been much research devoted to learning from positive information because the availability of negative examples is rare in practice. However, the difficulty of learning from positive data is in the risk of overgeneralization, learning a language strictly larger than the target language. Angluin offers a means to avoid overgeneralization via "tell-tales", a unique set of strings that distinguish a language from other languages in its family [2]. She states conditions for the language family that, if true, guarantee that if the tell-tale strings are included in the positive presentation seen so far by the inference algorithm then it can be sure its current guess is not an overgeneralization.

2.2 Teacher and Queries

This learning model is similar in spirit to the game "twenty questions" and uses a teacher, also called an oracle, who knows the target language and answers queries from the inference algorithm. In practice, the teacher is often a human who knows the target language and aids the inference algorithm, but in theory can be any process hidden from the inference algorithm that can answer particular types of questions. Angluin describes six types of queries that can be asked of the teacher, two of which have a significant impact on language learning: membership and equivalence [6]. A teacher that answers both membership and equivalence queries is said to be a *minimally adequate teacher* because she is sufficient to help identify DFAs in polynomial time without requiring any examples from the target language [5].

For a membership query, the inference algorithm presents a string to the teacher who responds with “yes” if the string is in the language or “no” if it is not. Likewise for an equivalence query, the inference algorithm presents a grammar hypothesis to the teacher who answers “yes” or “no” if the guess is equivalent to the target grammar or not. In the case when the teacher answers “no” she also provides a counter-example, a string from the symmetric difference of the target language and the guessed language, allowing the inference algorithm to zero in on the target grammar. The symmetric difference of two sets A and B are the elements in either A or B but not both: $A \bigoplus B = (A \cup B) - (A \cap B)$.

Queries provide an alternate means to measure the learnability of a class of languages. They can be used on their own or in conjunction with a presentation of samples, either positive or complete, to augment the abilities of the learner. Section 3 discusses how learning with queries differs in difficulty from learning in the limit for various language classes.

2.3 PAC Learning

In 1984 Valiant proposed the Probably Approximately Correct (PAC) learning model [55]. This model has elements of both identification in the limit and learning from an oracle, but differs because it doesn’t guarantee exact identification with certainty. As its name implies, PAC learning measures the correctness of its result by two user-defined parameters, ϵ and δ, representing accuracy and confidence respectively. This learning model is quite general and thus uses different terminology than typically found in formal languages, but of course applies just as well to grammatical inference. The goal is still to learn a “concept” (grammar) from a set of “examples of a concept” (strings).

Valiant assumes there exists a (possibly unknown) distribution D over the examples of a target concept that represent how likely they are to naturally occur, and makes available to the inference algorithm a procedure that returns these examples according to this distribution. As with Gold’s identification in the limit, PAC learning incrementally approaches the target concept with more accurate guesses over time.

A metric is proposed to measure the distance between two concepts, defined as the sum of probabilities $D(w)$ for all w in the symmetric difference of $L(G)$ and $L(G')$. In Figure 1, the lightly shaded regions represent the symmetric difference between $L(G)$ and $L(G')$. The area of this region decreases as the distance between the two concepts decreases. In the case of grammatical inference, these two concepts refer to the target grammar and the inference algorithm’s current guess.

The PAC learning model’s criteria for a successful inference algorithm is one that can confidently (i.e. with probability at least $1-\delta$) guess a concept with high accuracy (i.e. distance to the target concept is less than ϵ). Valiant demonstrates the PAC learning model with a polynomial time algorithm that approximates bounded conjunctive normal form (k-CNF) and monotone disjunctive normal form (DNF) expressions using just the positive presentation from D and a membership oracle.

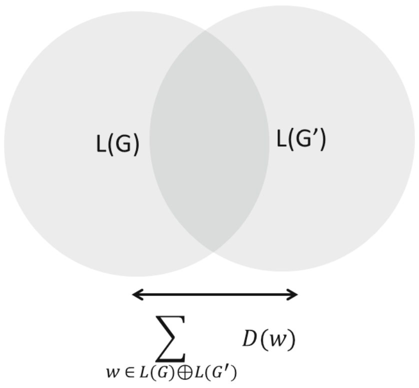


Fig. 1. The PAC-learning measure of distance between two language concepts

The novelty and uniqueness of Valiant's model intrigued the grammatical inference community, but negative and NP-hardness equivalence results (e.g. [33,49]) dampened enthusiasm for PAC learning. Many feel Valiant's stipulation that the algorithm must learn polynomially under *all* distributions is too stringent to be practical since the learnability of many apparently simple concept classes are either known to be NP-hard, or at least not known to be polynomially learnable for all distributions.

Li and Vitanyi propose a modification to the PAC learning model that only considers simple distributions [43]. These distributions return simple examples with high probability and complex examples with low probability, where simplicity is measured by Kolmogorov complexity. Intuition is that simple examples speed learning. This is corroborated by instances of concepts given by the authors that are polynomially learnable under simple distributions but not known to be polynomially learnable under Valiant's more general distribution assumptions.

Despite the learnability improvements that simple PAC learning offers, the PAC learning model has attracted little interest from grammatical inference researchers in recent years. Identification in the limit and query-based learning models remain far more prevalent, with newer models such as neural networks and genetic algorithms also garnering interest.

3 Complexity

A significant portion of the grammatical inference literature is dedicated to an analysis of its complexity and difficulty, with results typically stated for a specific grammar class or type. The broadest form of result is simply whether a language class can be learned or not, while other results consider learning in polynomial time, learning the simplest grammar for the target language, or identifying the target language with a particular probability. Table 1 outlines the complexity results for different language classes and learning models.

Table 1. Learnability and complexity results for various language classes using different learning models

Language Class	**Presentation**		**Queries**		
	Complete	**Positive**	**Membership Only**	**Equivalence Only**	**Both**
Finite	Identifiable in the limit [24]	Identifiable in the limit [24]			
k-reversible automata		Polynomial [4]			
Strictly deterministic automata		Identifiable in the limit [60]			
Superfinite	Identifiable in the limit [24]	Not identifiable in the limit [24]			
Regular	Finding the minimum state DFA is NP-hard [25]		Polynomial for representative sample [3]	No polynomial algorithm [7]	Polynomial [5]
		Polynomial [15]			
Reversible context-free		Identifiable in the limit with structured strings [52]			
Noncounting context-free		Identifiable with structured strings [16]			
Very simple		Polynomial identifiable in the limit [59]		Polynomial [59]	
Structurally reversible context-free					Polynomial [11]
Simple deterministic					Polynomial [28]
Context-free			As hard as inverting RSA [8]		Polynomial with structured strings [51]

Gold showed that a large class of languages can be identified in the limit from complete information including the regular, context-free, context-sensitive, and primitive recursive classes. This identification can be accomplished by a brute-force style of technique called *identification by enumeration* where, as each new example is presented, the possible grammars are enumerated until one is found that satisfies the presentation seen so far. By contrast, positive information alone cannot identify the aforementioned classes in the limit, nor any other superfinite class [24]. The subsequent sections describe the two easiest grammar classes to infer from the Chomsky hierarchy: regular grammars and context-free grammars. Very little research has been attempted on the inference of more powerful grammar classes such as context-sensitive and unrestricted grammars, so they are omitted from this overview.

3.1 Deterministic Finite Automata

For any non-trivial language, multiple different grammars can be constructed to generate it. Likewise, there can exist DFAs that differ in their size but are equivalent in the sense that they accept the same language. When inferring a DFA from examples, it is naturally desirable to find the smallest DFA that accepts the target language. There exists only one such minimal DFA for a given language, known as the *canonical* DFA acceptor for that language. Despite the strong identification power of complete information, finding the minimal DFA that accepts an unknown regular language from a finite set of positive and negative samples is NP-complete [25].

Early claims of polynomial-time inference algorithms use the number of states in the target language's canonical acceptor as the input size. With this criteria, Angluin gives negative results for the polynomial identification of regular languages using membership queries only [3] or equivalence queries only [7]. However, if the membership oracle is augmented with a *representative sample* of positive data, a set of strings that exercise all the live transitions in the target language's canonical acceptor, then it is possible to identify a regular language in polynomial time [3]. By combining the power of both membership and equivalence queries, a regular language can be identified in polynomial time even without a single positive example in the unknown language [5]. Her proposed algorithm runs in time polynomial to the number of states in the minimum DFA and the longest counter-example provided by the equivalence oracle.

Several algorithms have been developed for the inference of DFAs from examples. These algorithms generally start by building an augmented prefix tree acceptor from positive and negative samples, then perform a series of state merges until all valid merges are exhausted. Each state merge has the effect of generalizing the language accepted by the DFA. The algorithms differ by how they select the next states to merge, constraints on the input samples, and whether or not they guarantee the inference of a minimal DFA.

An early state-merging algorithm is described by Trakhtenbrot and Barzdin that infers a minimal DFA in polynomial time, but requires that all strings up to a certain length are labeled as positive or negative [54]. The *regular positive* *and negative inference* (RPNI) algorithm also finds the canonical acceptor but allows an incomplete labeling of positive and negative examples which is more common in practice [48]. RPNI does, however, require the positive examples contain a characteristic set with respect to the target acceptor. A *characteristic set* is a finite set of positive examples $S \subset L(A)$ such that there is no other smaller automata A' where $S \subset L(A') \subset L(A)$. Lang provides convincing empirical evidence that his *exbar* algorithm out-performs comparable algorithms, and represents the state of the art in minimal DFA inference [38].

The algorithms discussed so far are guaranteed to infer a minimal DFA for the given examples, but *evidence driven state merging* (EDSM) algorithms relax this requirement for better scalability and performance. The order that states in a prefix tree are merged has a significant impact on an algorithm's performance because each merge restricts possible future merges. Bad merge decisions cause a lot of backtracking that can be avoided with smarter merge decisions. EDSM algorithms are so named because they use evidence from the merge candidates to determine a merge that is likely to be a good generalization, such as the heuristic proposed by Rodney Price in the first EDSM algorithm [39] and a winner of the Abbadingo Learning Competition. Differences in EDSM algorithms come down to the search heuristic used to select merges, and several have been tried such as beam search [38], stochastic search and the *self-adaptive greedy estimate* (SAGE) algorithm [32]. These search heuristics are comparable in performance and are the best known inference algorithms for large or complex DFAs.

3.2 Context-Free Grammars

Polynomial-time algorithms to learn higher grammar classes have also been investigated, in particular for context-free grammars. Identifying context-free grammars in polynomial time is considerably more difficult than for DFAs, so most polynomial results in the literature either learn a strict subset of context-free grammars, use structured strings as input, or both. Unlike DFA inference, there is currently no known polynomial algorithm to identify a general context-free language from positive and negative samples.

Angluin and Kharitonov give a hardness result that applies to all context-free languages: constructing a polynomial-time learning algorithm for context-free grammars using membership queries only is computationally equivalent to cracking well-known cryptographic systems, such as RSA inversion [8].

Anecdotally, it appears a fruitful method to find polynomial-time learning algorithms for context-free languages from positive samples is to adapt corresponding algorithms from DFA inference, with the added stipulation that the sample strings be structured. A structured string is a string along with its unlabelled derivation tree, or equivalently a string with nested brackets to denote the shape of its derivation tree. Sakakibara has shown this method effective by adapting Angluin's results for learning DFAs by a minimally adequate teacher [5] and learning reversible automata [4] to context-free variants with structured strings [51,52].

Clark et al. have devised a polynomial algorithm for the inference of languages that exhibit two special characteristics: the finite context property and the finite kernel property [15]. These properties are exhibited by all regular languages, many context-free languages, and some context-sensitive languages. The algorithm is based on positive data and a membership oracle. More recently, Clark has extended Angluin's result [5] of learning regular languages with membership and equivalence queries to a larger subclass of context-free languages [14].

Despite the absence of a general efficient context-free inference algorithm, many researchers have developed heuristics that provide relatively good performance and accuracy by sacrificing exact identification in all cases. We describe several such approaches related to software engineering in Section 4.

4 Applications in Software Engineering

Grammatical inference has its roots in a variety of separate fields, a testament to its wide applicability. Implementors of grammatical inference applications often have an unfair advantage over purely theoretical GI research because theorists must restrict themselves to inferring abstract machines (DFAs, context-free grammars, transducers, etc.) making no additional assumptions about the underlying structure of the data. Empiricists, on the other hand, can make many more assumptions about the structure of their data because their inference problem is limited to their particular domain.

Researchers attempting to solve a practical inference problem will usually develop their own custom solution, taking advantage of structural assumptions about their data. Often this additional domain knowledge is sufficient to overcome inference problems that theorists have proved impossible or infeasible with the same techniques in a general environment. The applications described in the following sections use grammatical inference techniques, but rarely result from applying a purely theoretical result to a practical problem.

4.1 Inference of General Purpose Programming Languages

Programming language design is an obvious area to benefit from grammatical inference because grammars themselves are first-class objects. Programming languages almost universally employ context-free, non-stochastic grammars to parse a program, which narrows the possible inference approaches considerably when looking for an inductive solution. When discussing the inference of programming language grammars here, the terms "sample" and "example" refer to instances of computer programs written in the target programming language.

Crespi-Reghizzi et al. suggest an interactive system to semi-automatically generate a programming language grammar from program samples [17]. This system relies heavily on the language designer to help the algorithm converge on the target language by asking for appropriate positive and negative examples. Every time the learning algorithm conjectures a new grammar, it outputs all sentences for that grammar up to a certain length. If the conjectured grammar is too large, there will be sentences in the output that don't belong and the designer marks them as such. If the conjectured grammar is too small, there will be sentences missing from the output and the designer is expected to provide them. The designer's corrections are fed back into the algorithm which corrects the grammar and outputs a new conjecture, and the process repeats until the target grammar is obtained.

Another system is proposed by Dubey et al. to infer a context-free grammar from positive samples for a programming language dialect when the standard language grammar is already known [21]. Their algorithm requires the non-terminals in the dialect grammar to be unchanged from the standard grammar, but allows for the terminals and production rules to be extended in the dialect grammar (i.e. new keywords can be added in the dialect along with their associated grammar rules). Their approach has the advantage of being fully automated so the designer simply needs to provide the dialect program samples and the standard language grammar. However, like many current CFG inference techniques, a heuristic is used which cannot guarantee the output grammar converges exactly to the target grammar.

4.2 Inference of Domain Specific Languages

Domain specific languages (DSLs) are languages whose syntax and notation are customized for a specific problem domain, and are often more expressive and declarative compared to general purpose languages. DSLs are intended to be used, and possibly designed, by domain experts who do not necessarily have a strong computer science background. Grammatical inference allows the creation of a grammar for a DSL by only requiring positive (and possibly negative) program samples by the designer.

Črepinšek et al. propose a genetic approach to infer grammars for small DSLs using positive and negative samples [56]. They combine a set of grammar production rules into a *chromosome* representing a complete grammar, then apply crossover and mutation genetic operators that modify a population of chromosomes for the next generation. They use a fitness function that reflects the goal of having the target grammar accept all positive samples and reject all negative samples. Since a single random mutation is more likely to produce a grammar that rejects both positive and negative samples, the authors found that testing a chromosome on only positive samples converges more quickly to the target grammar than testing it on negative samples. Therefore, they chose a fitness value proportional to the total length (in tokens) of the positive samples that can be parsed by a chromosome. Negative samples, used to control overgeneralization, are only included in the fitness value if all positive samples are successfully parsed.

This genetic approach has been shown to accurately infer small DSLs [56], including one discussed by Javed et al. to validate UML class diagrams from use cases [29]. Javed et al. express UML class diagrams in a custom DSL and require a domain expert to provide positive and negative use cases written in that DSL. The system validates these use cases against the given UML diagrams and reports feedback to the user, who can use that feedback to change the UML diagrams to improve use case coverage. In this situation the computer is providing valuable context and information to the human user who is making the important generalization and specialization decisions for the grammar, but in theory UML diagrams can be synthesized entirely from the use case descriptions given a sufficiently powerful grammar inference engine.

Javed et al. extend their genetic algorithm by learning from positive samples only by using beam search and Minimum Description Length (MDL) heuristics [40] in place of negative examples to control overgeneralization of the conjectured grammar [31]. The idea here is to find the simplest grammar at each step and incrementally approach the target grammar. MDL is used as a measure of grammar simplicity, and beam search is used to more efficiently search the solution space of possible grammars. One disadvantage of this approach is it requires the positive samples to be presented in a particular order, from simplest to most complex, which allows the learning algorithm to encode the incremental differences from the samples into the target grammar. The authors' subsequent effort into a grammar inference tool for DSLs, called *MAGIc*, eliminates this need for an order-specific presentation of samples by updating the grammar based on the difference between successive (arbitrary) samples [46,27]. This frees the designer from worrying about the particular order to present their DSL samples to the learning algorithm. Hrnčič et al. demonstrate how *MAGIc* can be adapted to infer DSLs embedded in a general purpose language (GPL) given the GPL's grammar [26]. The GPL's grammar rules are included in the chromosome, but frozen so they cannot mutate. Learning, therefore, occurs strictly on the DSL syntax and the locations in the GPL grammar where the embedded DSL is allowed.

The inference of DSLs can make it easier for non-programmer domain experts to write their own domain-specific languages by simply providing examples of their DSL programs. It can also be used in the migration or maintenance of legacy software whose grammar definitions are lost or unavailable.

4.3 Inference of Graph Grammars and Visual Languages

Unlike one-dimensional strings whose elements are connected linearly, visual languages and graphs are connected in two or more dimensions allowing for arbitrary proximity between elements. Graph grammars define a language of valid graphs by a set of production rules with subgraphs instead of strings on the right-hand side.

Fürst et al. propose a graph grammar inference algorithm based on positive and negative graph samples [23]. The algorithm starts with a grammar that produces exactly the set of positive samples then incrementally generalizes towards a smaller grammar representation, a strategy similar to typical DFA inference algorithms which build a prefix tree acceptor then generalize by merging states. The authors demonstrate their inference algorithm with a flowchart example and a hydrocarbon example, making a convincing case for its applicability to software engineering tasks such as metamodel inference and reverse engineering visual languages.

Another graph grammar inference algorithm is proposed by Ates et al. which repeatedly finds and compresses overlapping identical subgraphs to a single non-terminal node [9]. This system uses only positive samples during the inference process, but validates the resulting grammar by ensuring all the graphs in the training set are parsable and other graphs which are close to but distinct from the training graphs are not parsable. Ates et al. demonstrate their algorithm with two practical inferences: one for the structure of a programming language and one for the structure of an XML data source.

Kong et al. use graph grammar induction to automatically translate a webpage designed for desktop displays into a webpage designed for mobile displays [35]. The inference performed is similar to the aforementioned proposed by Ates et al. [9] because they both use the Spatial Graph Grammar (SGG) formalism and subgraph compression. The induction algorithm consumes webpages, or more accurately their DOM trees, to produce a graph grammar. After a human has verfied this grammar it is used to parse a webpage, and the resulting parse is used to segment the webpage into semantically related subpages suitable for display on mobile devices.

Graph grammar inference algorithms are less common than their text-based counterparts, but provide a powerful mechanism to infer patterns in complex structures. Parsing graphs is NP-hard in general, causing these algorithms to be more computationally expensive than inference from text. Most graph grammar learners overcome this complexity by restricting their graph expressiveness or employing search and parse heuristics to achieve a polynomial runtime.

4.4 Other Uses in Software Engineering

Section 3 describes the difficulty inferring various language classes from positive samples alone, and in particular that only finite languages can be identified in the limit from positive samples [24]. The SEQUITUR algorithm, developed by Nevill-Manning and Witten, is designed to take a single string (long but finite) and produce a context-free grammar that reflects repetitions and hierarchical structure contained in that string [47]. This differs from typical grammar inference algorithms because it does not generalize. Data compression is an obvious use of this algorithm, but it has found other uses in software engineering. For example, Larus uses the SEQUITUR algorithm to concisely represent a program's entire runtime control flow and uses this dynamic control flow information to identify heavily executed paths in the program to focus performance and compiler optimization efforts [41]. It can also be used on the available positive samples as a first step in a generalizing context-free grammar inference algorithm, such as in [46] to seed an initial population of grammars for a genetic approach.

Ahmed Memon proposes using grammatical inference in log files to identify anomalous activity [45]. He treats the contents of log files in a system running normally as a specific language, and any erroneous or anomalous activities reported in the log file are therefore not part of this language. Memon trains a grammar from positive log file samples of a system running normally, then parses subsequent log file entries using this grammar to identify anomalous activity. The inference procedure depends on knowledge of the domain, specifically sixteen text patterns that appear in typical log files: dates, times, IP addresses, session IDs, etc. Once these patterns are identified and normalized, a custom non-terminal merging algorithm is used to generalize the log file grammar.

Another recent use for grammatical inference is in the area of model-driven engineering. The relationship between a grammar and the strings it accepts is analogous to the relationship between a metamodel and the instance models it accepts. Javed et al. describe a method to use grammar inference techniques on a set of instance models to recover a missing or outdated metamodel [30]. The process involves converting the instance models in XML format to a domain-specific language, then performing existing grammar inference techniques on those DSL programs. The authors use their previously developed evolutionary approach [57] to do the actual inference, then recreate a metamodel in XML format from the result so the recovered metamodel can be loaded into a modeling tool. Liu et al. have recently extended this system to handle models with a more complex and segmented organizational structure [44]. The authors refer to these as multi-tiered domains because they support multiple viewpoints into the model.

Two similar problems are grammar convergence [37] and grammar recovery [36], both which involve finding grammars for a variety of software artifacts. The goal of grammar convergence is to establish and maintain a mapping between software artifact structures in different formats that embody the same domain knowledge. Grammatical inference can aid in the early steps of this process to produce a grammar for each knowledge representation by examining available concrete examples. Existing grammar transformation and convergence techniques can then be used on the resulting source grammars to establish a unified grammar.

Grammar recovery can be viewed as a more general version of the grammar inference problem because it seeks to recover a grammar from sources such as compilers, reference manuals and written specifications, in addition to concrete program examples. The effort by Lämmel and Verhoef to recover a VS COBOL II grammar includes leveraging visual syntax diagrams from the manual [36]. These diagrams give clues about the shape of the target grammar's derivation tree, knowledge that is known to greatly improve the accuracy of grammatical inference techniques. For example, reversible context-free and non-counting context-free languages are known to be identifiable from positive examples with these types of structured strings [52,16]. Furthermore, structured strings can be used to identify any context-free language in polynomial time with a membership and equivalence oracle [51]. In the case of grammar recovery, an existing compiler for the language (even without the compiler source code) may be used as a membership oracle.

5 Related Surveys

Many surveys of grammatical inference have been written to introduce newcomers to the field and summarize the state of the art. Most give a thorough overview of grammatical inference in general, but each emphasise different aspects of the literature.

Fu and Booth (1986) give a detailed technical description of some early inference algorithms and heuristics with an emphasis on pattern recognition examples to demonstrate its relevance [22]. This survey is heavy on technical definitions and grammatical notation, suitable for someone with prior knowledge in formal languages who prefers to get right into the algorithms and techniques of grammatical inference.

Vidal (1994) provides a concise but thorough overview of the learning models and language classes in grammatical inference, with ample citations for follow-up investigation [58]. He presents each learning model in the context of fundamental learnability results in the field as well as their practical applications without getting too deeply into the details of each learning model.

Dana Ron's doctoral thesis (1995) on the learning of deterministic and probabilistic finite automata primarily investigates PAC learning as it relates to identifying DFAs, and describes practical applications of its use [50]. Although not exhaustive of grammatical inference in general, this thesis is a good reference for someone specifically interested in DFA inference.

Lee (1996) presents an extensive survey on the learning of context-free languages, including those that have non-grammar formalisms [42]. She discusses approaches that learn from both text and structured data, making it relevant to software engineering induction problems.

Sakakibara (1997) provides an excellent overview of the field with an emphasis on computational complexity, learnability, and decidability [53]. He covers a wide range of grammar classes including DFAs, context-free grammars and their probabilistic counterparts. This survey is roughly organized by the types of language classes being learned.

Colin de la Higuera (2000) gives a high-level and approachable commentary on grammatical inference including its historical progress and achievements [18]. He highlights key issues and unsolved problems, and describes some promising avenues of future research. This commentary is not meant as a technical introduction to inference techniques nor an exhaustive survey, and therefore contains no mathematical or formal notation. It rather serves as a quick and motivational read for anyone interested in learning about grammatical inference. A similar piece on grammatical inference is written by Honavar and de la Higuera (2001) for a special issue of the *Machine Learning* journal (volume 44), emphasizing the cross-disciplinary nature of the field.

Cicchello and Kremer (2003) and Bugalho and Oliveira (2005) survey DFA inference algorithms in depth, with excellent explanations about augmented prefix tree acceptors, state merging, the red-blue framework, search heuristics, and performance comparisons of state of the art DFA inference algorithms [13,10].

de la Higuera (2005) provides an excellent guide to grammatical inference, geared toward people (not necessarily experts in formal languages or computational linguistics) who think grammatical inference may help them solve their particular problem [19]. He gives a general roadmap of the field, examples of how grammatical inference has been used in existing applications, and provides many useful references for further investigation by the reader.

Pieter Adriaans and Menno van Zaanen (2006) compare grammatical inference from three different perspectives: linguistic, empirical, and formal [1]. They introduce the common learning models and broad learnability results in the framework of each perspective, and comment on how these perspectives overlap. This survey is useful for someone who comes from a linguistic, empirical, or formal languages background and wishes to learn about grammatical inference.

6 Future Direction and Challenges

Software engineers are finding a variety of uses for grammatical inference in their work, but grammatical inference is still relatively rare in the field. The theoretical work in grammatical inference is largely disconnected from these practical uses because implementers tend to use domain specific knowledge to craft custom solutions. Such solutions, while successful in some cases, usually ignore the powerful algorithms developed by the theoretical GI community. Domain knowledge should continue to be exploited – we are not advocating otherwise – but domain knowledge needs to be translated into a form general-purpose inference algorithms can use. We believe this is the biggest challenge currently facing software engineers wanting to use grammatical inference in their applications: how to map their domain knowledge to theoretical GI constraints.

Constraints can be imposed in several ways, such as simplifying the grammar class to learn, providing negative samples, adding a membership and/or equivalence oracle where none existed, or partially structuring the input data. Often these constraints are equivalent to some existing structural knowledge or implicit assumptions about the input data, but identifying these equivalences is nontrivial.

```
while limit > a do
    begin
        if a > max then max = a;
        a := a + 1
    end
```

Fig. 2. A sample program in an unknown Pascal-like language

We motivate this approach with a concrete example, inspired by an example from Sakakibara [52]. Suppose you have a collection of computer programs written in an unknown language with a Pascal-like syntax and wish to infer a grammar from the collection. For clarity, Figure 2 shows a sample program in the collection and Figure 3 shows the target grammar to learn (a subset of the full Pascal grammar).

$$\begin{aligned}
Statement &\rightarrow Ident := Expression\\
Statement &\rightarrow \textbf{while}\ Condition\ \textbf{do}\ Statement\\
Statement &\rightarrow \textbf{if}\ Condition\ \textbf{then}\ Statement\\
Statement &\rightarrow \textbf{begin}\ Statementlist\ \textbf{end}\\
Statementlist &\rightarrow Statement; Statementlist\\
Statementlist &\rightarrow Statement\\
Condition &\rightarrow Expression > Expression\\
Expression &\rightarrow Term + Expression\\
Expression &\rightarrow Term\\
Term &\rightarrow Factor\\
Term &\rightarrow Factor \times Term\\
Factor &\rightarrow Ident\\
Factor &\rightarrow (Expression)
\end{aligned}$$

Fig. 3. The target grammar for the Pascal-like language [52]

At first glance this inference problem seems too difficult to solve. It is a context-free grammar with positive samples only, and Gold proved learning a superfinite language in the limit from positive-only samples is impossible [24]. Even with the addition of negative samples there is no known algorithm to efficiently learn a context-free language.

On closer inspection, however, there is additional structural information in the input samples, hidden in a place grammarware authors and parsers are trained to ignore – the whitespace. By taking into account line breaks and indented sections of source code in the input samples, a structured string can be constructed for each program. If we further assume the target grammar is reversible then we can apply a result by Sakakibara, who showed reversible context-free grammars can be learned in polynomial time from positive structured strings [52].

This particular solution depends on two assumptions: (1) all the input samples have meaningful and consistent whitespace formatting, and (2) the target grammar is in the class of reversible context-free grammars. The assumption that the target grammar is reversible context-free is reasonable, as many DSLs would fit this criterion. The Pascal subset grammar in Figure 3 is in fact reversible context-free, but full Pascal is not because adding a production rule like $Factor \rightarrow Number$ to this grammar violates the criteria of reversibility [52].

Leveraging domain knowledge and structural assumptions is quite powerful when inferring grammars from examples and should be encouraged, but at present mapping this domain-specific knowledge to abstract constructs in grammatical inference research requires some creativity and awareness of theoretical results in the field. Allowing the extensive work done in the theoretical grammatical inference community to bear on specific applications of GI would be a great boon to software engineering.

7 Conclusion

In grammatical inference, an inference algorithm must find and use common patterns in example sentences to concisely represent an unknown language in the form of a grammar. This process spans two axes of complexity: the language class to be learned and the learning model employed.

The theoretical foundations of grammatical inference are now well established thanks to contributions by Gold, Angluin and others. The state of the art, however, still has plenty of room to grow, and Colin de la Higuera identifies ten open problems on the theoretical side of grammatical inference that he believes are important to solve going forward [20].

In practice, assumptions can often be made which are not possible in a purely theoretical setting because a specific problem domain has limited scope, allowing for a better outcome than one would expect from simply applying the smallest enclosing theoretical result. Some work has already been done to investigate this relationship deeper, such as that by Kermorvant and de la Higuera [34] and Cano et al [12]. It would be valuable to find a widely applicable technique to equate domain assumptions with either a restriction in the class of language to learn, or an augmentation of the learning model.

Theoretical grammatical inference research continues to advance in many different directions: the language classes being learned, the learning models in use, the criteria for a successful inference, and the efficiency of the inference algorithms. Existing applications for grammatical inference are continually refined and new applications are found in a wide variety of disciplines.

References

1. Adriaans, P., van Zaanen, M.: Computational grammatical inference. STUDFUZZ, vol. 194, pp. 187–203. Springer, Heidelberg (2006)
2. Angluin, D.: Inductive inference of formal languages from positive data. Information and Control 45(2), 117–135 (1980)
3. Angluin, D.: A note on the number of queries needed to identify regular languages. Information and Control 51(1), 76–87 (1981)
4. Angluin, D.: Inference of reversible languages. Journal of the ACM (JACM) 29, 741–765 (1982)
5. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation 75, 87–106 (1987)
6. Angluin, D.: Queries and concept learning. Machine Learning 2(4), 319–342 (1988)
7. Angluin, D.: Negative results for equivalence queries. Machine Learning 5(2), 121–150 (1990)
8. Angluin, D., Kharitonov, M.: When won't membership queries help? In: Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing, STOC 1991, pp. 444–454. ACM, New York (1991)

9. Ates, K., Kukluk, J., Holder, L., Cook, D., Zhang, K.: Graph grammar induction on structural data for visual programming. In: 18th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2006, pp. 232–242 (November 2006)
10. Bugalho, M., Oliveira, A.L.: Inference of regular languages using state merging algorithms with search. Pattern Recogn. 38(9), 1457–1467 (2005)
11. Burago, A.: Learning structurally reversible context-free grammars from queries and counterexamples in polynomial time. In: Proceedings of the Seventh Annual Conference on Computational Learning Theory, COLT 1994, pp. 140–146. ACM, New York (1994)
12. Cano, A., Ruíz, J., García, P.: Inferring Subclasses of Regular Languages Faster Using *RPNI* and Forbidden Configurations. In: Adriaans, P.W., Fernau, H., van Zaanen, M. (eds.) ICGI 2002. LNCS (LNAI), vol. 2484, pp. 28–36. Springer, Heidelberg (2002)
13. Cicchello, O., Kremer, S.C.: Inducing grammars from sparse data sets: a survey of algorithms and results. J. Mach. Learn. Res. 4, 603–632 (2003)
14. Clark, A.: Distributional Learning of Some Context-Free Languages with a Minimally Adequate Teacher. In: Sempere, J.M., García, P. (eds.) ICGI 2010. LNCS, vol. 6339, pp. 24–37. Springer, Heidelberg (2010)
15. Clark, A., Eyraud, R., Habrard, A.: A Polynomial Algorithm for the Inference of Context Free Languages. In: Clark, A., Coste, F., Miclet, L. (eds.) ICGI 2008. LNCS (LNAI), vol. 5278, pp. 29–42. Springer, Heidelberg (2008)
16. Crespi-Reghizzi, S., Guida, G., Mandrioli, D.: Noncounting context-free languages. Journal of the ACM (JACM) 25(4), 571–580 (1978)
17. Crespi-Reghizzi, S., Melkanoff, M.A., Lichten, L.: The use of grammatical inference for designing programming languages. Communications of the ACM 16, 83–90 (1973)
18. de la Higuera, C.: Current Trends in Grammatical Inference. In: Amin, A., Pudil, P., Ferri, F., Iñesta, J.M. (eds.) SSPR&SPR 2000. LNCS, vol. 1876, pp. 28–31. Springer, Heidelberg (2000)
19. de la Higuera, C.: A bibliographical study of grammatical inference. Pattern Recognition 38, 1332–1348 (2005)
20. de la Higuera, C.: Ten Open Problems in Grammatical Inference. In: Sakakibara, Y., Kobayashi, S., Sato, K., Nishino, T., Tomita, E. (eds.) ICGI 2006. LNCS (LNAI), vol. 4201, pp. 32–44. Springer, Heidelberg (2006)
21. Dubey, A., Jalote, P., Aggarwal, S.: Learning context-free grammar rules from a set of programs. Software. IET 2(3), 223–240 (2008)
22. Fu, K.S., Booth, T.L.: Grammatical inference: introduction and survey/part i. IEEE Transactions on Pattern Analysis and Machine Intelligence 8, 343–359 (1986)
23. Fürst, L., Mernik, M., Mahnic, V.: Graph grammar induction as a parser-controlled heuristic search process, Budapest, Hungary (October 2011)
24. Gold, E.M.: Language identification in the limit. Information and Control 10(5), 447–474 (1967)
25. Gold, E.M.: Complexity of automaton identification from given data. Information and Control 37(3), 302–320 (1978)
26. Hrnčič, D., Mernik, M., Bryant, B.R.: Embedding Dsls Into Gpls: A Grammatical Inference Approach. Information Technology and Control 40(4) (December 2011)
27. Hrnčič, D., Mernik, M., Bryant, B.R., Javed, F.: A memetic grammar inference algorithm for language learning. Applied Soft Computing 12(3), 1006–1020 (2012)
28. Ishizaka, H.: Polynomial time learnability of simple deterministic languages. Machine Learning 5(2), 151–164 (1990)

29. Javed, F., Mernik, M., Bryant, B.R., Gray, J.: A grammar-based approach to class diagram validation (2005)
30. Javed, F., Mernik, M., Gray, J., Bryant, B.R.: MARS: a metamodel recovery system using grammar inference. Inf. Softw. Technol. 50(9-10), 948–968 (2008)
31. Javed, F., Mernik, M., Sprague, A., Bryant, B.: Incrementally inferring context-free grammars for domain-specific languages. In: Proceedings of the Eighteenth International Conference on Software Engineering and Knowledge Engineering (SEKE 2006), pp. 363–368 (2006)
32. Juillé, H., Pollack, J.B.: A Stochastic Search Approach to Grammar Induction. In: Honavar, V.G., Slutzki, G. (eds.) ICGI 1998. LNCS (LNAI), vol. 1433, p. 126. Springer, Heidelberg (1998)
33. Kearns, M., Li, M., Pitt, L., Valiant, L.: On the learnability of boolean formulae. In: Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing, STOC 1987, pp. 285–295. ACM, New York (1987)
34. Kermorvant, C., de la Higuera, C.: Learning Languages with Help. In: Adriaans, P.W., Fernau, H., van Zaanen, M. (eds.) ICGI 2002. LNCS (LNAI), vol. 2484, pp. 161–173. Springer, Heidelberg (2002)
35. Kong, J., Ates, K., Zhang, K., Gu, Y.: Adaptive mobile interfaces through grammar induction. In: 20th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2008, vol. 1, pp. 133–140 (November 2008)
36. Lämmel, R., Verhoef, C.: Semi-automatic grammar recovery. Softw. Pract. Exper. 31(15), 1395–1448 (2001)
37. 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)
38. Lang, K.J.: Faster algorithms for finding minimal consistent DFAs. Technical report (1999)
39. Lang, K.J., Pearlmutter, B.A., Price, R.A.: Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In: Proceedings of the 4th International Colloquium on Grammatical Inference, pp. 1–12. Springer, London (1998)
40. Langley, P., Stromsten, S.: Learning Context-Free Grammars with a Simplicity Bias. In: Lopez de Mantaras, R., Plaza, E. (eds.) ECML 2000. LNCS (LNAI), vol. 1810, pp. 220–228. Springer, Heidelberg (2000)
41. Larus, J.R.: Whole program paths. In: ACM SIGPLAN Notices, PLDI 1999, pp. 259–269. ACM, New York (1999)
42. Lee, L.: Learning of context-free languages: A survey of the literature. REP, 12–96 (1996)
43. Li, M., Vitányi, P.M.B.: Learning simple concepts under simple distributions. SIAM Journal of Computing 20, 911–935 (1991)
44. Liu, Q., Bryant, B.R., Mernik, M.: Metamodel recovery from multi-tiered domains using extended MARS. In: Proceedings of the 2010 IEEE 34th Annual Computer Software and Applications Conference, COMPSAC 2010, pp. 279–288. IEEE Computer Society, Washington, DC (2010)
45. Memon, A.U.: Log File Categorization and Anomaly Analysis Using Grammar Inference. Master of science, Queen's University (2008)
46. Mernik, M., Hrnčič, D., Bryant, B., Sprague, A., Gray, J., Liu, Q., Javed, F.: Grammar inference algorithms and applications in software engineering. In: XXII International Symposium on Information, Communication and Automation Technologies, ICAT 2009., pp. 1–7 (October 2009)

47. Nevill-Manning, C.G., Witten, I.H.: Identifying hierarchical structure in sequences: a linear-time algorithm. Journal of Artificial Intelligence Research 7(1), 67–82 (1997)
48. Oncina, J., García, P.: Identifying regular languages in polynomial time. In: Advances in Structural and Syntactic Pattern Recognition - Proceedings of the International Workshop on Structural and Syntactic Pattern Recognition, Bern, Switzerland, pp. 99–108 (1992)
49. Pitt, L., Valiant, L.G.: Computational limitations on learning from examples. Journal of the ACM (JACM) 35(4), 965–984 (1988)
50. Ron, D.: Automata Learning and its Applications. PhD thesis, Hebrew University (1995)
51. Sakakibara, Y.: Learning context-free grammars from structural data in polynomial time. Theoretical Computer Science 76(2-3), 223–242 (1990)
52. Sakakibara, Y.: Efficient learning of context-free grammars from positive structural examples. Information and Computation 97(1), 23–60 (1992)
53. Sakakibara, Y.: Recent advances of grammatical inference. Theoretical Computer Science 185, 15–45 (1997)
54. Trakhtenbrot, B.A., Barzdin, Y.M.: Finite Automata: Behaviour and Synthesis. North-Holland Publishing Company, Amsterdam (1973)
55. Valiant, L.G.: A theory of the learnable. Communications of the ACM 27, 1134–1142 (1984)
56. Črepinšek, M., Mernik, M., Bryant, B.R., Javed, F., Sprague, A.: Inferring context-free grammars for domain-specific languages. Electronic Notes in Theoretical Computer Science 141(4), 99–116 (2005)
57. Črepinšek, M., Mernik, M., Javed, F., Bryant, B.R., Sprague, A.: Extracting grammar from programs: evolutionary approach. ACM SIGPLAN Notices 40, 39–46 (2005)
58. Vidal, E.: Grammatical Inference: An Introductory Survey. In: Carrasco, R.C., Oncina, J. (eds.) ICGI 1994. LNCS, vol. 862, pp. 1–4. Springer, Heidelberg (1994)
59. Yokomori, T.: Polynomial-time learning of very simple grammars from positive data. In: Proceedings of the Fourth Annual Workshop on Computational Learning Theory, pp. 213–227. Morgan Kaufmann Publishers Inc., San Francisco (1991)
60. Yokomori, T.: On polynomial-time learnability in the limit of strictly deterministic automata. Machine Learning 19(2), 153–179 (1995)

Island Grammar-Based Parsing Using GLL and Tom

Ali Afroozeh[1], Jean-Christophe Bach[2,3,4], Mark van den Brand[1], Adrian Johnstone[5], Maarten Manders[1], Pierre-Etienne Moreau[2,3,4], and Elizabeth Scott[5]

[1] Eindhoven University of Technology, NL-5612 AZ Eindhoven, The Netherlands
afroozeh@gmail.com, m.g.j.v.d.brand@tue.nl, m.w.manders@tue.nl
[2] Inria, Villers-lès-Nancy, F-54600, France
[3] Université de Lorraine, LORIA, UMR 7503, Vandœuvre-lès-Nancy, F-54500, France
[4] CNRS, LORIA, UMR 7503, Vandœuvre-lès-Nancy, F-54500, France
jeanchristophe.bach@inria.fr, pierre-etienne.moreau@loria.fr
[5] Royal Holloway, University of London, Egham, Surrey, TW20 0EX, UK
a.johnstone@rhul.ac.uk, eas@cs.rhul.ac.uk

Abstract. Extending a language by embedding within it another language presents significant parsing challenges, especially if the embedding is recursive. The composite grammar is likely to be nondeterministic as a result of tokens that are valid in both the host and the embedded language. In this paper we examine the challenges of embedding the Tom language into a variety of general-purpose high level languages. Tom provides syntax and semantics for advanced pattern matching and tree rewriting facilities. Embedded Tom constructs are translated into the host language by a preprocessor, the output of which is a composite program written purely in the host language. Tom implementations exist for Java, C, C#, Python and Caml. The current parser is complex and difficult to maintain. In this paper, we describe how Tom can be parsed using island grammars implemented with the Generalised LL (GLL) parsing algorithm. The grammar is, as might be expected, ambiguous. Extracting the correct derivation relies on our disambiguation strategy which is based on pattern matching within the parse forest. We describe different classes of ambiguity and propose patterns for resolving them.

Keywords: GLL, Tom, island grammars, parsing, disambiguation.

1 Introduction

Modern software systems are composed of a wide range of programming languages. In many cases there is even a mixture of programming languages within one program. A traditional example is Cobol with CICS or SQL embeddings. It is possible for these embeddings to be realised via strings in which the extension constructs are encoded. In this case the parser of the host language does not need to be aware of the fact that another language is present. The drawback of these string encodings is that syntax errors in the embedded language constructs are not detected until a later phase in which these constructs are processed.

In the last decade language developers have been working on extending general-purpose programming languages with domain-specific languages, referred to in this paper as guest languages. The language embeddings which are not simple string encodings present a challenge from a parsing point of view, especially if the general-purpose programming language and the embeddings can be unboundedly nested. Tom [1] is an example of such an extension which provides general-purpose programming languages, such as Java, C, C#, Python, and Caml, referred to in this paper as the host language, with advanced pattern matching, term rewriting and tree traversal functionality. The Tom compiler processes the Tom constructs and generates corresponding host language code. The host-language compiler can then be used to build the final program from the generated code. The main advantage of this two-phase compiling approach is that the host-language compiler remains unaware of the extension constructs. As a result, the host language can evolve without breaking the extension's compiler, and the guest language can be used to extend other host languages.

If language embeddings are not simple string encodings, the syntax of the host language needs to be modified to accommodate the guest language. This is usually done by using tags which signal the beginning and end of guest constructs. In case of nesting host constructs inside guest constructs, another set of tags may be employed for notifying the parser of the return to the host language [2]. However, the use of tags for switching between languages is not very convenient for developers who use the language. In Tom, only an opening tag is used, while the closing tag is the same as the closing tag of blocks in the host language. Furthermore, switching to the host language inside Tom constructs does not need any special tag, and the host and guest languages may be unboundedly nested. These features make parsing Tom even more challenging.

Parsing the full syntax of the host language is neither desirable nor practical in many cases, especially for Tom, in which we only need to extract the guest constructs. One way to avoid parsing the full syntax of the host language is to use island grammars [3]. An island grammar captures the important constructs, embedded constructs in our case, as "island" and ignores the rest as "water". Two main issues should be taken into consideration while parsing island grammars. First, the class of deterministic context-free languages is not closed under union, meaning that the union of two deterministic languages may no longer be deterministic. Therefore, even if one designs LL(k) or LR(k) syntax for a language extension, there is no guarantee that the resulting language is in the same class. Second, the host and extension languages may share tokens, which may lead to ambiguities. The ambiguities from shared tokens cannot always be resolved by rewriting the grammar, using more lookahead tokens or backtracking, so there is a need for more sophisticated disambiguation schemes.

Attempting to parse an island grammar of a language using standard LL or LR parsing techniques will, at the very least, involve significant modifications in the parser and, in worst case, may not even be possible. For example, the current Tom parser uses multiple parsers, implemented in ANTLR[1], to deal with the host and Tom constructs separately. This implementation is complex, hard to maintain, and does not reflect the grammar of the language. Moreover, having a complex parser implementation makes the changes in and evolution of (both) languages a burden.

During the last 30 years, more efficient implementations of generalised parsers have become available. Since the algorithm formulated by Tomita [4] there have been a number of generalised LR parsing (GLR) implementations, such as GLR [5], a scannerless variant (SGLR) [6] and Dparser[2]. Johnstone and Scott developed a generalised LL (GLL) parser [7] in which the function call stack in a traditional recursive descent parser is replaced by a structure similar to the stack data structure (Graph Structured Stack) used in GLR parsing. GLL parsers are particularly interesting because their implementations are straightforward and efficient.

Our goal is to avoid manipulation within a parser in order to provide a generic, reusable solution for parsing language embeddings. We have chosen Tom as our case study, mainly because of challenges involved in parsing Tom such as recursive embedding and the lack of closing tags. In this paper, we present an island grammar for Tom in EBNF. The fundamental question is how to deal with ambiguities present in island grammars which support recursive embedding. For disambiguation we perform pattern matching within the parse forest. To validate our approach, we conducted parsing experiments using an improved version of our Java implementation of GLL [8].

The rest of this paper is organised as follows. In Section 2 we introduce Tom as a language for term rewriting and give a brief description of the GLL parsing algorithm along with some notes on our GLL implementation. Section 3 describes the idea of island grammars by defining an island grammar for Tom. In Section 4 we illustrate our mechanism for resolving ambiguities in island-based grammars by providing disambiguation rules for different types of ambiguities present in Tom. The results of parsing Tom examples are presented in Section 5. In Section 6 we present other work in the area of parsing embedded languages and compare our approach with them. Finally, in Section 7, a conclusion to this paper and some ideas for future work are given.

2 Preliminaries

In this section we introduce the Tom language which is used for two different purposes in the rest of this paper. First, Tom is used as an example of an embedded syntax with recursive nesting, which poses difficulties for conventional deterministic parsers. Second, Tom is used as a pattern matching and rewriting technology for implementing a disambiguation mechanism for island grammars. The rest of this section gives a brief explanation of the GLL parsing algorithm and our Java-based GLL implementation.

[1] http://www.antlr.org/

[2] http://dparser.sourceforge.net/

2.1 Tom in a Nutshell

Tom [9,1] is a language based on rewriting calculus and is designed to integrate term rewriting and pattern matching facilities into general-purpose programming languages (GPLs) such as Java, C, C#, Python, and Caml. Tom relies on the Formal Island framework [10], meaning that the underlying host language does not need to be parsed in order to compile Tom constructs.

The basic functionality of Tom is pattern matching through the `%match` construct. This construct can be seen as a generalisation of the *switch-case* construct in many GPLs. The `%match` construct is composed of a set of rules where the left-hand side is a *pattern*, *i.e.*, a tree that may contain variables, and the right-hand side an *action*, given by a Java block that may in turn contain Tom constructs.

A second construct provided by Tom is the backquote (`) term. Given an algebraic term, *i.e.*, a tree, a backquote term builds the corresponding data structure by allocating and initialising the required objects in memory.

A third component of the Tom language is a formalism to describe algebraic data structures by means of the `%gom` construct. This corresponds to the definition of inductive types in classical functional programming. There are two main ways of using this formalism: the first one is defining an algebraic data type in Gom and generating Java classes which implement the data type. This is similar to the Eclipse Modeling Framework[3], in which a Java implementation can be generated from a meta-model definition. The second approach assumes that a data structure implementation, for example in Java, already exists. Then, an algebraic data type in Gom can be defined to provide a *mapping* to connect the algebraic type to the existing implementation.

Listing 1 illustrates a simple example of a Tom program. The program starts with a Java class definition. The `%gom` construct defines an algebraic data type with one module, `Peano`. The module defines a sort `Nat` with two constructors: `zero` and `suc`. The constructor `suc` takes a variable `n` as a field. Given a signature, a well-formed and well-typed term can be built using the backquote (`) construct. For example, for `Peano`, `` `zero() `` and `` `suc(zero()) `` are correct terms, while `` `suc(zero(),zero()) `` or `` `suc(3) `` are not well formed and not well typed, respectively.

In the example in Listing 1, the `plus()` and `greaterThan()` methods are implemented by pattern matching. The semantics of pattern matching in Tom is close to the *match* which exists in functional programming languages, but in an imperative context. A `%match` construct is parametrised by a list of subjects, *i.e.*, expressions evaluated to ground terms, and contains a list of rules. The left-hand side of the rules are patterns built upon constructors and new variables, without any restriction on linearity (a variable may appear twice, as in `x,x`). The right-hand side is a Java statement that is executed when the pattern matches the subject. Using the backquote construct (`) a term can be created and returned. Similar to the standard `switch/case` construct, patterns are evaluated from top to bottom. In contrast to the functional *match*, several actions, *i.e.*, right-hand side, may be fired for a given subject as long as no `return` or `break` is executed.

[3] http://www.eclipse.org/modeling/emf/

```
public class PeanoExample {

 %gom {
  module Peano
  Nat = zero() | suc(n: Nat)
 }

 public Nat plus(Nat t1, Nat t2) {
  %match(t1,t2) {
   x,zero() -> { return `x; }
   x,suc(y) -> { return `suc(plus(x,y)); }
 }

boolean greaterThan(Nat t1, Nat t2) {
 %match(t1,t2) {
   x,x          -> { return false; }
   suc(_),zero() -> { return true; }
   zero(),suc(_) -> { return false; }
   suc(x),suc(y) -> { return `greaterThan(x,y); }
  }
}

public final static void main(String[] args) {
  Nat N = `zero();
  for(int i=0 ; i<10 ; i++) { N = `suc(N); }
} }
```

Listing 1. An example of a Java and Tom program. Tom parts are in bold

In addition to the syntactic matching capabilities illustrated above, Tom also supports more complex matching theories such as matching modulo associativity, associativity with neutral element, and associativity-commutativity.

2.2 Generalised LL Parsing

Top down parsers whose execution closely follows the structure of the underlying grammar are attractive, particularly because they make grammar debugging easier. GLL is a top down parsing technique which is fully general, allowing even left recursive grammars, which has worst-case cubic runtime order and which is close to linear on most ‘real’ grammars. In this section we give a basic description of the technique, a full formal description can be found in [7].

A GLL parser effectively traverses the grammar using the input string to guide the traversal. There may be several traversal threads, each of which has a pointer into the grammar and a pointer into the input string. For each nonterminal A there is a block of code corresponding to each alternate of A. At each step of the traversal, (i) if the grammar pointer is immediately before a terminal we attempt to match it to the current input symbol; (ii) if it is immediately before a nonterminal B then the pointer moves to the start of the block of code associated with B; (iii) if it is at the end of an alternate of A then it moves to the position immediately after the instance of A from which it came. This control flow is essentially the same as for a classical recursive descent parser in which the blocks of code for a nonterminal X are collected into a *parse function* for X with traversal steps of type (ii) implemented as a function call to X and traversal steps of type (iii) implemented as function return. In classical recursive descent, we use the runtime stack to manage actions (ii) and (iii) but in a general parser there may be multiple parallel traversals arising from nondeterminism; thus in the GLL algorithm the call stack is handled directly using a Tomita-style graph structured stack (GSS) which allows the potentially infinitely many stacks arising from multiple traversals to be merged and handled efficiently.

Multiple traversal threads are handled using process descriptors, making the algorithm parallel rather than backtracking in nature. Each time a traversal bifurcates, the current grammar and input pointers, together with the top of the current stack and associated portion of the derivation tree, are recorded in a descriptor. The outer loop of a GLL algorithm removes a descriptor from the set of pending descriptors and continues the traversal thread from the point at which the descriptor was created. When the set of pending descriptors is empty all possible traversals will have been explored and all derivations (if there are any) will have been found. Details of the creation and processing of the descriptors by a GLL algorithm can be found in [7] and a more implementation oriented discussion can be found in [11].

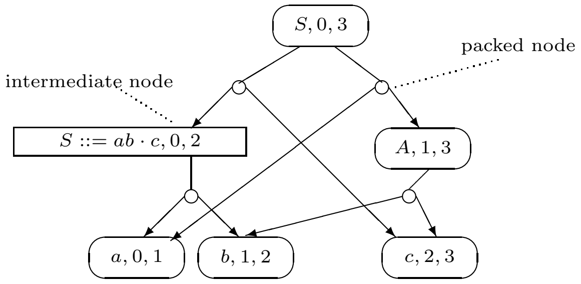


Fig. 1. SPPF

The output of a GLL parser is a representation of all the possible derivations of the input in the form of a *shared packed parse forest* (SPPF), a union of all the derivation trees of the input in which nodes with the same tree below them are shared and nodes which correspond to different derivations of the same substring from the same nonterminal are combined by creating a packed node for each family of children. To make sure that descriptors contain only one tree node, the root of the current subtree, the SPPFs are binarised in the natural left associative manner, with the two left-most children being grouped under an intermediate node, which is in turn then grouped with the next child to the left, etc. It is this binarisation that keeps both the size of the SPPF and the number of descriptors worst-case cubic.

In detail, a binarised SPPF has three types of SPPF nodes: symbol nodes, with labels of the form (x, j, i) where x is a terminal, nonterminal or ϵ and $0 \le j \le i \le n$ (n is the size of the input); intermediate nodes, with labels of the form (t, j, i); and packed nodes, with labels for the form (t, k), where $0 \le k \le n$ and t is a grammar slot. (A grammar slot is essentially an LR(0)-item, a formal definition can be found in [7].) Terminal symbol nodes have no children. Nonterminal symbol nodes, (A, j, i), have packed node children of the form $(A ::= \gamma\cdot, k)$ and intermediate nodes, (t, j, i), have packed node children with labels of the form (t, k), where $j \le k \le i$. A packed node has one or two children, the right child is a symbol node (x, k, i), and the left child, if it exists, is a symbol or intermediate node, (s, j, k). For example, for the grammar $S ::= abc \mid aA, \quad A ::= bc$ we obtain the SPPF as shown in Fig. 1. As is clear from this example, for ambiguous grammars there will be more than one derivation.

2.3 GLL Implementation Notes

Currently, the GLL parsing algorithm does not natively support EBNF; therefore, we convert a grammar described in EBNF to an equivalent BNF grammar prior to the generation of a GLL parser. In our conversion scheme, for example, an EBNF symbol $X*$, the repetition of X, is replaced by a nonterminal named X_*, having two alternates: $X\_* ::= X\ X\_*$ and $X\_* ::= \epsilon$. As the consequence, the resulting SPPF contains symbol nodes associated with the EBNF symbols introduced in the conversion. After parsing, when an SPPF is created, we remove the nodes associated with the EBNF conversion, by replacing them with their children, so that the resulting SPPF corresponds to the initial EBNF grammar.

Our GLL implementation uses a separate lexer. The lexer is driven by the parser and returns all possible token types seen at a particular point of the input. These tokens may overlap. Being a top-down parser, GLL decides at each grammar position whether tokens received from the lexer are relevant. This check is performed by testing the token types against the *first* and *follow* sets of nonterminals at the position. All the relevant tokens at a position are consumed and irrelevant ones are simply ignored. Moreover, the syntax of lexical definitions used by our lexer is inspired by SDF [12]. The difference with SDF is that we only escape characters which have a special meaning in the regular expression definitions. These characters are \, `.`, `^`, `[`, `]`, and `-`.

3 Island Grammars: Tom Syntax as an Example

Island grammars are a method for describing the syntax of a language, concentrating only on relevant constructs. An island grammar comprises two sets of rules: rules for *islands* describing the relevant language constructs which should be fully parsed and rules for *water* describing the rest of the text with which we are not concerned. In parsing embedded languages, the embedded language and host language constructs are captured as island and water, respectively. In this section, we define an island grammar for Tom. The presented approach is general enough to be used in other contexts as well.

```
context-free syntax
Program ::= Chunk*
Chunk ::= Water | Island
```

Listing 2. The starting rules of an island grammar

The starting rules of an island grammar are shown in Listing 2. As can be seen, a program is defined as a list of `Chunk` nonterminals, each being either an island or water. The rules defining water are presented in Listing 3. In island grammars, the overlap between water and island tokens should be recognised. To this end, we define fine-grained `Water` tokens being as small as the building blocks of the supported host languages. In addition, our island grammar is designed to be host-language agnostic, although it may not possible to completely achieve it. For this purpose, the water definition should be flexible and capture the different varieties of tokens appearing in different supported host languages. For example, `SpecialChar` in Listing 3 is defined as the union of all different symbols appearing in the host languages supported by our island grammar.

```
context-free syntax
Water ::= Identifier | Integer | String | Character | SpecialChar

lexical syntax
Identifier ::= [a-zA-Z_]+[a-zA-Z0-9\-_]*
Integer ::= [0-9]+
String ::= ["]([^"\\]|[\\][nrf\\"'tb])*["]
Character ::= [']([^'\\]|[\\][nrf\\"'tb])[']
SpecialChar ::= [; : + \- = & < > * ! % : ? | & @ \[ \] \^ # $ { } , \. ( )]
```

Listing 3. The definition of `Water`

```
context-free syntax
Island ::= TomConstruct | BackQuoteTerm
TomConstruct ::= IncludeConstruct | MatchConstruct | GomConstruct | ...
```

Listing 4. Tom constructs

As can be seen in Listing 4, Tom islands fall into two groups: Tom constructs, prefixed by `%`, and the backquote term. Most of the Tom constructs have the same structure. In this section, we focus on two constructs: the `%include` construct, which is the simplest construct of Tom, and the `%match` construct, which is one of the most used features of Tom. The syntax of these constructs is presented in Listing 5. As can be seen, the production rule of `PatternAction` contains `Chunk*`, meaning that Tom and host-language constructs can be recursively nested inside a `%match` construct.

```
context-free syntax
IncludeConstruct ::= "%include" "{" Water* "}"

MatchConstruct ::= "%match" ( "(" MatchArguments ")" )? "{" PatternAction* "}"
MatchArguments ::= Type? Term ("," Type? Term)*
PatternAction ::= PatternList "->" "{" Chunk* "}"
Term ::= Variable | VariableStar | Identifier "(" ( Term ("," Term)* )? ")"

lexical syntax
Identifier ::= [a-zA-Z_]+[a-zA-Z0-9\-_]*
Variable ::= Identifier
VariableStar ::= Identifier [*]
Type ::= Identifier
```

Listing 5. The `%include` and `%match` constructs

Listing 6 introduces the backquote term. As can be seen, a new water type, `BackQuoteWater`, is introduced in the backquote term definition. The difference between `BackQuoteWater` and `Water` is that the former does not contain parentheses, dot, or commas, while the latter does. These characters should be captured as part of a backquote term's structure and not as water. Examples of backquote terms are `‘x`, `‘x*`, `‘f(1 + x)`, `‘f(1 + h(t), x)`, and `‘(f(x)%b)`. The idea behind backquote terms is to combine classical algebraic notations of term rewriting with host language code such as `"1 +"` or `"%b"`. Note that `x` and `f(x)` inside a backquote term can be interpreted as Tom terms or Java variables and method calls. Based on the defined grammar in Listing 6 and the disambiguation rules in Section 4.3, `x` and `f(x)` are recognized as Tom terms. In the compilation phase, if these terms were not valid Tom terms, they will be printed to the output as they are, thus producing Java variables and method calls.

```
context-free syntax
BackQuoteTerm ::= "‘" CompositeTerm | "‘" "(" CompositeTerm+ ")"
CompositeTerm ::= VariableStar | Variable
               | Identifier "(" (CompositeTerm+ ("," CompositeTerm+)*)? ")"
               | BackQuoteWater
BackQuoteWater ::= Identifier | Integer | String | Character | BackQuoteSpecialChar

lexical syntax
BackQuoteSpecialChar ::= [; : + \- = & < > * ! % : ? | & @ \[ \] \^ # $ { }]
```

Listing 6. The backquote term

4 Disambiguation

In this section we propose a pattern matching technique to resolve the ambiguities present in island grammars. The pattern matching is performed after parsing, when an SPPF is fully built. As discussed in Section 2.2, an SPPF provides efficient means for representing ambiguities by sharing common subtrees. An ambiguity node in an SPPF is a symbol or an intermediate node having more than one packed node as children. For example, consider the grammar of arithmetic expressions in Listing 7. For this grammar, the input string `"1+2+3"` is ambiguous. Its corresponding SPPF is presented in Fig. 2, which contains an ambiguity occurring under the root symbol node.

```
context-free syntax
E ::= E "+" E | Digit
lexical syntax
Digit ::= [1-9]+
```

Listing 7. A simple grammar for arithmetic expressions

For SPPF visualisation, packed nodes are represented by small circles, intermediate nodes by rectangles, and symbol nodes by rounded rectangles in which the name of the symbol node is written. If a symbol node represents a terminal, the matched lexeme is also shown next to the terminal name, *e.g.*, `Digit: 1`. Keywords, such as `"+"`, are represented by themselves without the quotation marks.

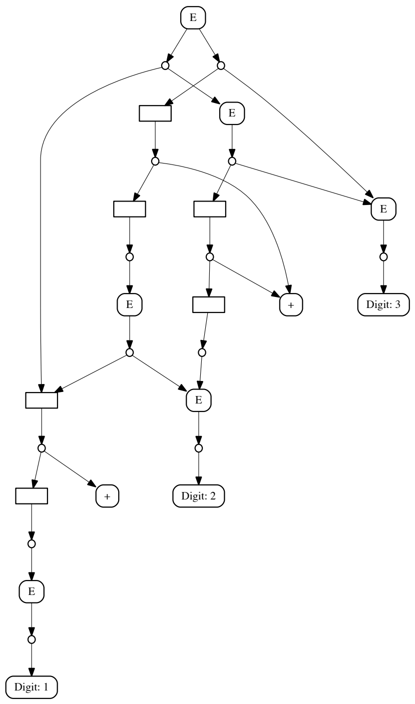


Fig. 2. The complete SPPF

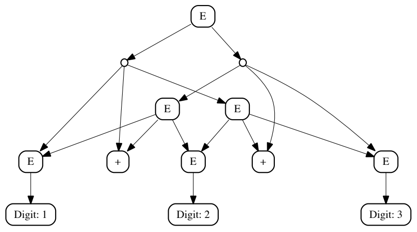


Fig. 3. The reduced SPPF

Furthermore, for a more compact visualisation, nodes related to the recognition of layout, whitespace and comments, are excluded from the figures in this section. In our GLL implementation, layout is recognised after each terminal and before the start symbol of a grammar. Layout is captured through the nonterminal `Layout`, which is defined as `Layout ::= LayoutDef*`, where `LayoutDef` is a lexical definition, *e.g.*, whitespace.

While an SPPF is built, intermediate nodes are added for binarisation, which is essential for controlling the complexity of the GLL parsing algorithm. Furthermore, the version of the GLL algorithm we are using creates packed nodes even when there are no ambiguities. These additional nodes lead to a derivation structure which does not directly correspond to the grammar from which the SPPF has been created.

After the successful construction of an SPPF, one can traverse the SPPF and remove all packed nodes which are not part of an ambiguity, *i.e.*, the packed nodes which are the only child. The intermediate nodes which only have one child, the ones which do not present an ambiguity, can also be removed. When a node is removed, its children are attached to the parent of the node. Care should be taken that the order of children in the parent remains intact. By removing unnecessary packed and intermediate nodes, the SPPF presented in Fig. 2 can be reduced to the SPPF in Fig. 3.

To resolve ambiguities in an SPPF, we traverse a reduced SPPF to find the deepest ambiguity node. Then, the children of this node are checked against a set of disambiguation rules. A disambiguation rule is a rewrite rule that can either (i) remove a packed node matching an "illegal" pattern, or (ii) prefer a packed node over another one. These rules are called *remove* and *prefer* rules, respectively. In *prefer* rules, none of the patterns is illegal, but if both patterns appear under an ambiguity node, one of them should be preferred. A disambiguation rule may apply to more than one packed node under an ambiguity node or may not apply at all.

After applying the set of disambiguation rules on an ambiguity node, if only one packed node remains, the disambiguation is successful. Then, the remaining packed node is replaced by its children. To completely disambiguate an SPPF, disambiguation is performed bottom up. This is because, in many cases, resolving ambiguities in higher level ambiguity nodes in an SPPF depends on first resolving the ambiguities in deeper levels, as subtrees of an ambiguity node in a higher level may be ambiguous themselves. By performing the disambiguation bottom up, it is ensured that the subtrees of an ambiguity node are not ambiguous, thus they can uniquely be specified by tree patterns. If the entire disambiguation procedure is successful, an SPPF is transformed into a parse tree only containing symbol nodes. The syntax of disambiguation rules is as follows:

– A *remove* rule is written as `remove P`, where `P` is a pattern matching a packed node under an ambiguity node.
– A *prefer* rule is written as `prefer` $\mathtt{P_i}$ $\mathtt{P_j}$, where $\mathtt{P_i}$ and $\mathtt{P_j}$ are patterns matching sibling packed nodes under an ambiguity node, in which $\mathtt{P_i}$ should be preferred to $\mathtt{P_j}$. Note that in this notation the list of packed nodes is considered modulo associativity-commutativity (AC), *i.e.*, as a multiset data-structure, and thus the order in which packed nodes appear does not matter.
– A packed node is written as $(\mathtt{s_1}\ \mathtt{s_2}\ \ldots\ \mathtt{s_n})$, where each $\mathtt{s_i}$ is a pattern describing a symbol node.
– A symbol node whose children are not important for us is represented by its label, *e.g.*, `E`. If a symbol node represents a keyword, it should be surrounded by double-quotes, *e.g.*, `"+"`.
– A symbol node in the general form is written as $\mathtt{l}(\mathtt{s_1}\ \mathtt{s_2}\ \ldots\ \mathtt{s_n})$, where `l` is the label of the symbol node, and each $\mathtt{s_i}$ is a pattern describing the symbol node's `ith` child which is a symbol node. In patterns describing symbol nodes, no packed node can appear. This is because in the bottom up disambiguation we always examine the deepest ambiguity node whose subtrees are not ambiguous and, therefore, do not contain packed nodes. Note that additional packed and intermediate nodes which are not part of an ambiguity are already removed before applying patterns.
– `"_"` and `"_*"` match any symbol node and zero or more occurrences of any symbol node, respectively.

It should be mentioned that the user does not need to explicitly specify layout when writing patterns using this syntax. Layout is automatically captured after each terminal symbol, when patterns are translated to Tom rewrite rules, see

Section 4.4. The ambiguity shown in Fig. 3 can be resolved by selecting the left-associative derivation, *i.e.*, `(1+2)+3`, rather than the other derivation, `1+(2+3)`. For this purpose, we define the right-associative derivation as illegal and remove it using the following rule: `remove (E "+" E(E "+" E))`. As can be seen, the disambiguation rule closely follows the grammar rules in Listing 7 and the resulting SPPF in Fig. 3.

4.1 The Island-Water Ambiguity

Tom islands, with the exception of the backquote term, do not have unique opening and closing tags. We define a token as unique if it only appears in either the host or the embedded language. The lack of unique opening and closing tags may mislead the parser into recognising a Tom construct as a collection of successive water tokens. This leads to an ambiguity which we call the island-water ambiguity.

Consider the starting rules of Tom's island grammar in Listing 2. When a GLL parser is about to select the alternates of `Chunk`, if the current input index points to a percentage sign followed by a Tom construct name, such as `include` or `match`, both alternates of `Chunk` are selected. Processing both alternates may lead to an island-water ambiguity. The SPPF corresponding to parsing the input `"%include {file}"` is illustrated in Fig. 4.

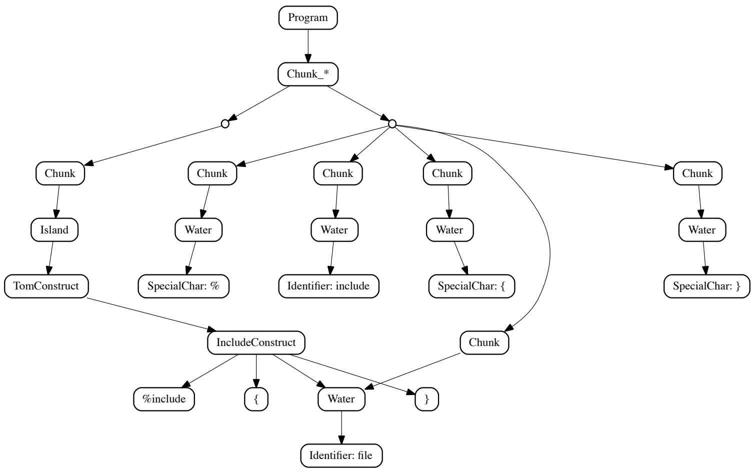


Fig. 4. The Island-Water ambiguity in an `%include` construct

The island-water ambiguity can be resolved by preferring an island to a sequence of water tokens. This can be achieved by the following disambiguation rule:

```
prefer (Chunk(Island)) (Chunk(Water) _*)
```

As for this example, if the ambiguity contains more than one water token, the ambiguity occurs under a node labelled `Chunk_*`, which is in fact a node resulting from the EBNF to BNF conversion, see Section 2.3. The conversion, however, does not affect the writing of patterns, as patterns describe tree structures under packed nodes and not their parent, the ambiguity node.

4.2 The Island-Island Ambiguity

Detecting the end of a Tom construct is significantly more difficult than its beginning. Tom constructs end with a closing brace, which is also the end-of-block indicator in the supported host languages of Tom. Detecting the end of a Tom construct may lead to what we call the island-island ambiguity, which happens between islands with different lengths. In this kind of ambiguity, some closing braces have been wrongly interpreted as water.

```
1 %match(s) {
2   x -> {
3      { System.out.println(‘x); }
4   }
5 }
```

Listing 8. A `%match` construct containing an Island-Island ambiguity

An example of a Tom program containing an island-island ambiguity is given in Listing 8. Parsing this example produces two different match constructs, and hence an ambiguity. A match construct needs at least two opening and closing braces to be successfully recognised, and the rest of the tokens, including any other closing braces, may be treated as water. For this example, every closing brace, after the second closing brace on line 4, may be interpreted as the closing brace of the match construct.

Resolving the island-island ambiguity is difficult, mainly because it depends on the semantics of the embedded and host languages. We disambiguate this case by choosing the match construct which has well-balanced braces in its inner water fragments. There are two ways to check the well-balancedness of braces. First, we can use a manual traversal function which counts the opening and closing braces inside water fragments of a Tom construct. This check is, however, expensive for large islands. Second, we can prevent islands with unbalanced braces to be created in the first place by modifying the lexical definition of `SpecialChar` in Listing 3. We remove braces from the lexical definition and add `"{" Chunk* "}"` as a new alternate to `Water`. With this modification, opening and closing braces can only be recognised as part of a match construct or pairs of braces surrounding water tokens. We chose the second option for resolving the island-island ambiguities occurring in Tom's island grammar.

4.3 The Backquote Term Ambiguities

Identifying the beginning of a backquote term is straightforward because the backquote character does not exist in Tom's supported host languages. Therefore, no ambiguity occurs at the beginning of a backquote term. However, a number of ambiguities may occur while detecting the end of a backquote term. The simplest example of an ambiguous backquote term is `‘x`, in which `x` can either be recognised as `BackQuoteWater` or `Variable`. This ambiguity is depicted in Fig. 5. For disambiguation, we prefer a `Variable` over `BackQuoteWater` by the following disambiguation rule:

```
prefer (Variable) (BackQuoteWater)
```

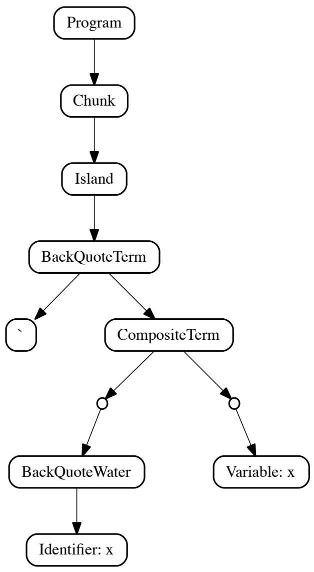


Fig. 5. `BackQuoteWater/Variable`

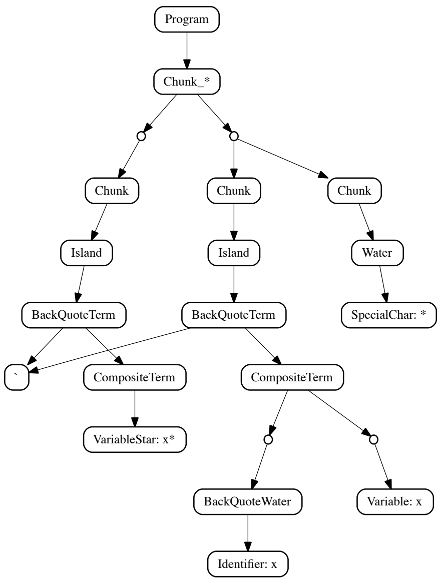


Fig. 6. `Variable/VariableStar`

Another example of ambiguity in the backquote term can be observed in `‘x*`, which can be recognised as a backquote term containing `VariableStar`, denoting a list of terms, or a backquote term containing the variable `x` after which a water token (the asterisk character) follows. The disambiguation in this case depends on the typing information of the host language. For example, `‘x* y`, considering Java as the host language, should be recognised as `‘x` multiplied by `y`, and not `‘x*` followed by `y`, provided that `y` is an integer. We do not, however, deal with the typing complexities and currently follow a simple rule: if an asterisk character directly follows a variable, the recognition of `VariableStar` should be preferred. The SPPF corresponding to the parsing of `‘x*` is shown in Fig. 6.

In the SPPF in Fig. 6 two ambiguities are present: one between the recognition of `x` as `BackQuoteWater` or `Variable`, which is already discussed. The new ambiguity, caused by the presence of the asterisk character is under `Chunk_*`, can be resolved by the following rule:

```
prefer (Chunk(Island(BackQuoteTerm))
       (Chunk(Island(BackQuoteTerm)) Chunk(Water))
```

An ambiguity similar to the one present in `‘x*` occurs in backquote terms ending in parentheses, see Listing 6. For example, in `‘x()`, parentheses can be recognised as water and not as part of the backquote term. We use a disambiguation rule in which a backquote term containing the parentheses as part of the backquote term should be preferred to the one in which parentheses are recognised as water.

4.4 Implementation

So far, we described our disambiguation mechanism without exploring the implementation details. To implement disambiguation rules, we use the pattern matching and rewriting facilities of Tom. Listing 9 defines an algebraic data type for an SPPF using a `%gom` construct. The algebraic type defines a sort `SPPFNode` with three constructors, corresponding to three node types present in an SPPF, and the sort `NodeList` defining a list of nodes. The `SymbolNode` constructor creates a symbol node with its label and its list of the children. The `PackeNode` and `IntermediateNode` constructors create a packed or intermediate node by their list of children. Finally, `concNode` is the constructor for creating a list of `SPPNode` terms using the `*` operator.

```
%gom {
  SPPFNode = SymbolNode(label:String, children:NodeList)
           | PackedNode(children:NodeList)
           | IntermediateNode(children:NodeList)
  NodeList = concNode(SPPFNode*)
}
```

Listing 9. The algebraic type definition for an SPPF

In addition to the algebraic signature in Listing 9, we use a Tom *mapping* which connects the Java implementation of an SPPF, used by the parser, to the algebraic view. For example, using the mapping, an instance of the `SymbolNode` class, from the Java implementation, is viewed as a `SymbolNode` algebraic constructor. The subterms of this constructor, which are of sort `String` and `NodeList`, are then automatically retrieved by the mapping. More importantly, through this algebraic view, the Java implementation can be automatically traversed using Tom strategies, without the need to provide hand-written visitors. All this efficient and statically-typed machinery is generated and optimised by the Tom compiler.

```
SymbolNode("E", concNode(z1*,
  PackedNode(concNode(
        SymbolNode("E",_)
        SymbolNode("+",_),_,
        SymbolNode("E", concNode(SymbolNode("E",_), SymbolNode("+",_), _, SymbolNode("E",_)))
  )),z2*)) -> SymbolNode("E", concNode(z1*, z2*));
```

Listing 10. A rewrite rule in Tom for disambiguating binary operators

Listing 10 shows how the rule for removing a right-associative derivation, *i.e.*, `remove (E "+" E(E "+" E))`, for the grammar in Listing 7, is translated to a Tom rewrite rule. The notation `concNode(z1*, PackedNode(...), z2*)` denotes *associative* matching with *neutral element*, meaning that the subterm `PackedNode` is searched into the list of packed nodes, and the *context*, possibly empty, being captured by variables `z1*` and `z2*`. Furthermore, the anonymous variable `"_"` is automatically added after each terminal to capture layout.

5 Results

Using the island grammar and the disambiguation rules presented in sections 3 and 4, respectively, we were able to parse all the Tom programs from the tom-examples package. This package, which is shipped with the source distribution of Tom, contains more than 400 examples (for a total of 70,000 lines of code) showing how Tom is used in practice. The size of these examples varies from 24 lines of code (679 characters) to 1,103 lines of code (30,453 characters).

Based on our findings, the examples having a size of about 10,000 characters could be parsed and disambiguated in less than one second, and the disambiguation time of an instance was always lower than its parsing time. Moreover, from what we observed, the parsing time for Tom examples was linear.

Table 1. Parsing times for the selected Tom examples (in milliseconds)

Example file	#Lines	#Tokens	Parsing time	Disambiguation time	Total time
Peano.t	84	340	362	64	426
Compiler.t	169	1,365	718	304	1,022
Analysis.t	253	1,519	667	358	1,025
Langton.t	490	3,430	1,169	524	1,693
TypeInference.t	909	6,503	1,632	850	2,482
BigExample.t	1,378	11,682	2,917	739	3,656

In order to evaluate the efficiency of our approach, we selected five representative examples, see Table 1. Peano is a simple example manipulating Peano integers. Compiler is a compiler for a Lazy-ML implementation. Analysis is a bytecode static analyser based on CTL formulae. Langton is a cellular automata simulator, which contains many algebraic patterns. TypeInference is a type-inference algorithm for patterns, in presence of sub-typing. This example contains many nested `%match` constructs. We also considered an artificial example, BigExample, that could be made as big as needed, by concatenating pieces of code.

We compared our implementation with the current implementation of the Tom parser, which also uses an island-grammar approach, implemented using the ANTLR parser generator. Currently, the ANTLR 3-based implementation is approximatively twice as fast on a few examples than the GLL implementation. However, our GLL implementation is not yet thoroughly optimised. In particular, the scanner can be made more efficient. Therefore, we expect to obtain better results in the future.

6 Related Work

The name "island grammar" was coined in [13] and later elaborated on in [3]. Moonen showed in [3] how to utilise island grammars, written in SDF, to generate robust parsers for reverse engineering purposes. Robust parsers should not fail when parsing incomplete code, syntax errors, or embedded languages. However, the focus of [3] is more on reverse engineering and information extraction from source code rather than parsing embedded languages.

There has been considerable effort on embedding domain-specific languages in general-purpose programming languages. For example, Bravenboer and Visser [14] presented MetaBorg, a method for providing concrete syntax for object-oriented libraries, such as Swing and XML, in Java. The authors demonstrated how to express the concrete syntax of language compositions in SDF [12], and transform the parsed embedded constructs to Java by applying rewrite rules using Stratego[4]. In MetaBorg the full grammar of Java is parsed, while we use an island-grammar based approach. Furthermore, MetaBorg uses distinct opening and closing tags for transition between the host and embedded languages which prevents many ambiguities to happen in the first place.

The ASF+SDF Meta-Environment[5] has been used in a number of cases for developing island grammars. SDF, the underlying syntax formalism of the ASF+SDF Meta-Enviornment, provides the *prefer* and *avoid* mechanisms to mark nodes in the parse forest which should be kept or removed [15]. These mechanisms, however, do not work in all cases and may produce unpredictable results. For example, [16,17] show examples of island grammars which cannot be disambiguated using SDF. The problem with *prefer* and *avoid* mechanisms is that the decision for selecting the desired parse tree under an ambiguity node is made by only considering the highest level production rule, which are marked with *prefer* or *avoid*. In many cases, disambiguation cannot be done by merely checking the highest level rule, and there is a need to explore the subtrees in more detail. Our disambiguation syntax allows the user to express the trees under ambiguity nodes using patterns in as much detail as needed.

The use of term rewriting and tree matching for resolving ambiguities is not new. For example, in [18], the authors demonstrated how to use rewrite rules written in ASF+SDF to resolve semantic ambiguities present in programming languages such as Cobol and C. Our approach in defining disambiguation rules is very similar to the one presented in [18]. However, our focus is on resolving context-free ambiguities in island grammars, rather than semantic ambiguities. In addition, we provide a simple pattern matching notation, whereas in [18] the disambiguation rules are expressed using a complete term rewriting language.

An example of using island grammars in parsing embedded languages which does not use a generalised parsing technique is presented by Synytskyy et al. in [19]. The authors presented an approach for robust multilingual parsing using island grammars. They demonstrated how to parse Microsoft's Active Server

[4] `http://strategoxt.org/`

[5] `http://www.meta-environment.org/Meta-Environment/ASF+SDF`

Pages (ASP) documents, a mix of HTML, Javascript, and VBScript, using island grammars. For parsing, they used TXL[6], which is a hybrid of functional and rule-based programming styles. TXL uses top-down backtracking to recognise the input. Due to too much backtracking, for some grammars, the parsing might be very slow or impractical [20]. Moreover, a grammar expressed in TXL is disambiguated by ordering the alternates. In contrast to this work, we used a generalised parser which has the worst case complexity of $O(n^3)$. Moreover, our disambiguation mechanism covers a wider range of ambiguities than alternate ordering.

Another related work which does not use a generalised parsing technique is presented by Schwerdfeger and Van Wyk in [21]. The authors described a mechanism for verifying the composability of deterministic grammars. In their approach, grammar extensions are checked for composability, and if all extensions pass the check, an LR parser and a context-aware scanner is generated for the composition. The context-aware scanner is used to detect the overlaps between tokens. The presented approach in [21] parses the full grammar of the host language instead of an island grammar. Furthermore, as mentioned by the authors, not all extensions pass the composiblitiy check. Our approach is more generic compared to [21] since by using GLL no restriction exists on composing grammars. More importantly, as explained in Section 2.3, a GLL parser with a separate lexer only consumes token types which are relevant at the parsing position, thus effectively providing context-sensitive lexing, without the need to modify the parser or change the parsing algorithm. Finally, using GLL and our disambiguation mechanism, we are able to deal with complex, ambiguous grammars.

For island grammar-based parsing, it is essential to deal with tokens which overlap or have different types. Aycock and Nigel Horspool in [22] proposed the Schrödinger's token method, in which the lexer reports the type of a token with multiple interpretations as the Schrödinger type, which is in fact a meta type representing all the actual types. When the parser receives a Schrödinger's token, for each of its actual types, parsing will be continued in parallel. The authors then demonstrated how to modify a GLR parser to support the Schrödinger's token.

An alternative to Schrödinger's token is to use scannerless parsing [6]. In scannerless parsing there is no explicit lexer present, so the parser directly works on the character level. This has the advantage of giving the parser the opportunity to decide on the type of a token based on the context in which the token's characters appear. Scannerless parsing has mainly two disadvantages. First, because every character in the language is a token, the size of the parse forest is larger than parsing with a scanner. Second, ambiguities happening at the character level, *e.g.*, the longest match for keywords, should be explicitly resolved in the parse forest. A GLL parser with a separate scanner provides the same power as a scannerless parser, without having to deal with character level ambiguities, as they are already resolved by the longest match at the lexical level.

[6] `http://www.txl.ca/`

7 Conclusions and Future Work

In this paper we have shown how languages based on island grammars can be parsed using existing technologies such as GLL and Tom. We developed a host-agnostic island grammar for Tom which only fully parses the Tom constructs and ignores the host language constructs. The primary challenge of island grammar-based parsing is the ambiguities resulting from the overlap between host and embedded language tokens. We analysed different ambiguity classes of Tom's island grammar and provided patterns for resolving them.

Our main objective was to avoid modification within the parser in order to provide a generic solution for parsing embedded languages. We proposed a method for disambiguating island grammars using pattern matching and rewriting the parse forest. Our disambiguation mechanism is implemented by the pattern matching mechanism of Tom. In addition, in one case, the island-island ambiguity, we rewrote the grammar to resolve the ambiguity. Using our approach, one can express a modular island grammar in EBNF, independent of the complexity or the number of embedded languages.

We performed a number of experiments to validate our findings. In these experiments, we parsed all the Tom files from the tom-examples package, with different characteristics such as different sizes and number of islands. Moreover, we compared our GLL implementation with the current ANTLR implementation. The GLL implementation has a parsing speed comparable to the current implementation, but in a few cases the ANTLR implementation is twice as fast.

As future work, there are a number of paths we shall explore. First, we will investigate how we can improve our GLL implementation as it is not yet thoroughly optimised. Second, we shall work on replacing the current ANTLR implementation with the GLL implementation in the Tom compiler. Third, we will investigate the possibility of pattern matching while parsing to increase the efficiency of the disambiguation process. Finally, we plan to apply our approach to other language compositions, *e.g.*, Java+XML or Java+SQL, to determine how generic our method is.

Acknowledgements. We would like to thank Alexander Serebrenik who has provided us with feedback on an early draft of this work and also helped us in measuring the complexity of parsing Tom examples. This work was partly funded by the Engineering and Physical Sciences Research Council within project EP/I032509/1.

References

1. Moreau, P.-E., Ringeissen, C., Vittek, M.: A Pattern Matching Compiler for Multiple Target Languages. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 61–76. Springer, Heidelberg (2003)
2. Bravenboer, M., Dolstra, E., Visser, E.: Preventing injection attacks with syntax embeddings. Science of Computer Programming 75(7), 473–495 (2010)

3. Moonen, L.: Generating robust parsers using island grammars. In: Proceedings of the 8th Working Conference on Reverse Engineering, pp. 13–22. IEEE (2001)
4. Tomita, M.: Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems. Kluwer Academic Publishers, Norwell (1985)
5. Rekers, J.: Parser Generation for Interactive Environments. PhD thesis, University of Amsterdam, The Netherlands (1992), `http://homepages.cwi.nl/~paulk/dissertations/Rekers.pdf`
6. Visser, E.: Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam (1997)
7. Scott, E., Johnstone, A.: GLL parse-tree generation. Science of Computer Programming (to appear, 2012)
8. Manders, M.W.: mlBNF - a syntax formalism for domain specific languages. Master's thesis, Eindhoven University of Technology, The Netherlands (2011), `http://alexandria.tue.nl/extra1/afstversl/wsk-i/manders2011.pdf`
9. Balland, E., Brauner, P., Kopetz, R., Moreau, P.-E., Reilles, A.: Tom: Piggybacking Rewriting on Java. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, pp. 36–47. Springer, Heidelberg (2007)
10. Balland, E., Kirchner, C., Moreau, P.-E.: Formal Islands. In: Johnson, M., Vene, V. (eds.) AMAST 2006. LNCS, vol. 4019, pp. 51–65. Springer, Heidelberg (2006)
11. Johnstone, A., Scott, E.: Modelling GLL Parser Implementations. In: Malloy, B., Staab, S., van den Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 42–61. Springer, Heidelberg (2011)
12. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The syntax definition formalism SDF-reference manual-. SIGPLAN Not. 24(11), 43–75 (1989)
13. van Deursen, A., Kuipers, T.: Building documentation generators. In: Proceedings of the IEEE International Conference on Software Maintenance, pp. 40–49 (1999)
14. Bravenboer, M., Visser, E.: Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. SIGPLAN Not. 39(10), 365–383 (2004)
15. den van Brand, M.G.J., Scheerder, J., Vinju, J.J., Visser, E.: Disambiguation Filters for Scannerless Generalized LR Parsers. In: Nigel Horspool, R. (ed.) CC 2002. LNCS, vol. 2304, pp. 143–158. Springer, Heidelberg (2002)
16. Post, E.: Island grammars in ASF+SDF. Master's thesis, University of Amsterdam, The Netherlands (2007), `http://homepages.cwi.nl/~paulk/theses/ErikPost.pdf`
17. van der Leek, R.: Implementation Strategies for Island Grammars. Master's thesis, Delft University of Technology, The Netherlands (2005), `http://swerl.tudelft.nl/twiki/pub/Main/RobVanDerLeek/robvanderleek.pdf`
18. van den Brand, M.G.J., Klusener, S., Moonen, L., Vinju, J.J.: Generalized parsing and term rewriting: Semantics driven disambiguation. Electr. Notes Theor. Comput. Sci. 82(3), 575–591 (2003)
19. Synytskyy, N., Cordy, J.R., Dean, T.R.: Robust multilingual parsing using island grammars. In: Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research, CASCON 2003, pp. 266–278. IBM Press (2003)
20. Cordy, J.R.: TXL - A Language for Programming Language Tools and Applications. Electronic Notes in Theoretical Computer Science 110, 3–31 (2004)
21. Schwerdfeger, A.C., Van Wyk, E.R.: Verifiable composition of deterministic grammars. SIGPLAN Not. 44(6), 199–210 (2009)
22. Aycock, J., Nigel Horspool, R.: Schrödinger's Token. Software, Practice & Experience 31, 803–814 (2001)

Layout-Sensitive Generalized Parsing

Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann

University of Marburg, Germany

Abstract. The theory of context-free languages is well-understood and context-free parsers can be used as off-the-shelf tools in practice. In particular, to use a context-free parser framework, a user does not need to understand its internals but can specify a language *declaratively* as a grammar. However, many languages in practice are not context-free. One particularly important class of such languages is layout-sensitive languages, in which the structure of code depends on indentation and whitespace. For example, Python, Haskell, F#, and Markdown use indentation instead of curly braces to determine the block structure of code. Their parsers (and lexers) are not declaratively specified but hand-tuned to account for layout-sensitivity.

To support *declarative* specifications of layout-sensitive languages, we propose a parsing framework in which a user can annotate layout in a grammar. Annotations take the form of constraints on the relative positioning of tokens in the parsed subtrees. For example, a user can declare that a block consists of statements that all start on the same column. We have integrated layout constraints into SDF and implemented a layout-sensitive generalized parser as an extension of generalized LR parsing. We evaluate the correctness and performance of our parser by parsing 33 290 open-source Haskell files. Layout-sensitive generalized parsing is easy to use, and its performance overhead compared to layout-insensitive parsing is small enough for practical application.

1 Introduction

Most computer languages prescribe a textual syntax. A parser translates from such textual representation into a structured one and constitutes the first step in processing a document. Due to the development of parser frameworks such as lex/yacc [15], ANTLR [18,17], PEGs [6,7], parsec [13], or SDF [8], parsers can be considered off-the-shelf tools nowadays: Non-experts can use parsers, because language specifications are declarative. Although many parser frameworks support some form of context-sensitive parsing (such as via semantic predicates in ANTLR [18]), one particularly relevant class of languages is not supported declaratively by any existing parser framework: layout-sensitive languages.

Layout-sensitive languages were proposed by Landin in 1966 [12]. In layout-sensitive languages, the translation from a textual representation to a structural one depends on the code's layout and its indentation. Most prominently, the *off-side rule* prescribes that all non-whitespace tokens of a structure must be further to the right than the token that starts the structure. In other words, a token

```
if x != y:
   if x > 0:
      y = x
else:
   y = -x
```

(a) Python: Indentation resolves the dangling else problem.

```
do input <- readInput
   case input of
     Just txt -> do putStrLn "thank you"
                    sendToServer txt
                    return True
     Nothing  -> fail "no input"
```

(b) Haskell: Nested block structure.

Fig. 1. Layout-sensitive languages use indentation instead of curly braces

is offside if it occurs further to the left than the starting token of a structure; an offside token must denote the start of the next structure. In languages that employ the offside rule, the block structure of code is determined by indentation and layout alone, whose use is considered good style anyway.

The offside rule has been applied in a number of computer languages including Python, Haskell, F#, and Markdown. The Wikipedia page for the off-side rule[1] lists 20 different languages that apply the offside rule. For illustration, Figure 1 shows a Python and a Haskell program that use layout to declare the code's block structure. The layout of the Python program specifies that the *else* branch belongs to the outer *if* statement. Similarly, the layout of the Haskell program specifies to which *do*-block each statement belongs. Unfortunately, current declarative parser frameworks do not support layout-sensitive languages such as Python or Haskell, which means that often the manually crafted parsers in compilers are the only working parsers. This makes it unnecessarily hard to extend these languages with new syntax or to create tools for them, such as refactoring engines or IDEs.

Our core idea is to declare layout as constraints on the shape and relative positioning of syntax trees. These *layout constraints* occur as annotations of productions in the grammar and restrict the applicability of annotated productions to text with valid layout. For example, for conditional expressions in Python, we annotate (among other things) that the *if* keyword must start on the same column as the *else* keyword and that all statements of a *then* or *else* branch must be further indented than the *if* keyword. These latter requirements are context-sensitive, because statements are rejected based on their appearance within a conditional statement. Thus, layout constraints cannot be fully enforced during the execution of a context-free parser.

We developed an extension of SDF [8] that supports layout constraints. The standard parsing algorithm for SDF is scannerless generalized LR parsing [21]. In a generalized parsing algorithm, all possible parse trees for an input string are processed in parallel. One approach to supporting layout would be to parse the input irrespective of layout in a first step (generating every possible parse tree), and then in a second step discard all syntax trees that violate layout

[1] http://en.wikipedia.org/w/index.php?title=Off-side_rule&oldid=517733101

constraints. However, we found that this approach is not efficient enough for practical applications: For many programs, the parser fails to terminate within 30 seconds. To improve performance, we identified a subset of layout constraints that in fact does not rely on context-sensitive information and therefore can be enforced at parse time. We found that enforcing these constraints at parse time and the remaining constraints at disambiguation time is sufficiently efficient.

To validate the correctness and to evaluate the performance of our layout-sensitive parser, we have build layout-sensitive SDF grammars for Python and Haskell. In particular, we applied our Haskell parser to all 33 290 Haskell files in the open-source repository Hackage. We compare the result of applying our parser to applying a traditional generalized parser to the same Haskell files where block structure has been made explicit through curly braces. Our study empirically validates the correctness of our parser and shows that our layout-sensitive parser can compete with parsers that requires explicit block structure.

We make the following contributions:

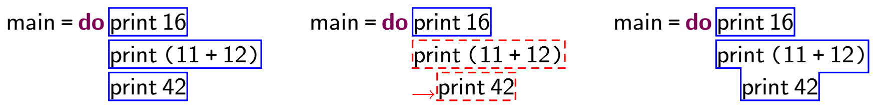
– We identify common idioms in existing layout-sensitive languages. Based on these idioms, we design a constraint language for specifying layout-sensitive languages declaratively.
– We identify context-free layout constraints that can be enforced at parse time to avoid excessive ambiguities.
– We implement a parser for layout-sensitive languages based on an existing scannerless generalized LR parser implementation in Java.
– We extended existing layout-insensitive SDF grammars for Python and Haskell with layout constraints.
– We evaluate the correctness and performance of our parser by parsing 33 290 open-source Haskell files and comparing the results against parse trees produced for Haskell files with explicit block structure. Our evaluation suggests that our parser is correct and fast enough for practical application.

Our parser, grammars, and raw evaluation data are open-source and available online at `http://github.com/seba--/layout-parsing`. While our parser implementation is based on a scannerless parser, the ideas presented in this paper are applicable to parsers with separate lexers as well.

2 Layout in the Wild

Many syntactic constructs in the programming language Haskell use layout to encode program structure. For example, the *do*-Block in the simple Haskell program in Figure 2(a) contains three statements which are horizontally aligned at the same column in the source code. We visualize the alignment by enclosing the tokens that belong to a statement in a box. More generally, a box encloses code corresponding to a subtree of the parse tree. The exact meaning of these boxes will become clear in the next section, where they form the basis of our constraint language.

A Haskell parser needs to check the alignment of statements to produce correct parse trees. For example, Figure 2(b) displays an incorrect parse tree that



```
main = do print 16
          print (11 + 12)
          print 42
```

(a) Three statements with correct vertical alignment.

```
main = do print 16
          print (11 + 12)
          → print 42
```

(b) Wrong parse: Statements have to begin in the same column, hence print 42 cannot be a statement.

```
main = do print 16
          print (11 + 12)
             print 42
```

(c) Correct parse: Only two statements, where the second print is applied to three arguments.

Fig. 2. Simple Haskell programs

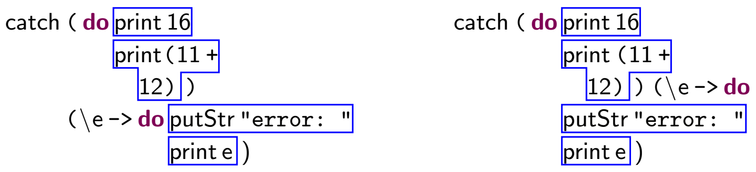
wrongly identifies print 42 as a separate statement, even though it is further indented than the other statements. Figure 2(c) visualizes the correct parse tree for this example: A *do*-Block with two statements. The second statement spans two lines and is parsed as an application of the function print to three arguments. In order to recognize program structure correctly, a parser for a layout-sensitive language like Haskell needs to distinguish programs as in Figure 2(a) from programs as in Figure 2(c).

It is not possible to encode this difference in a context-free grammar, because that would require counting the number of whitespace characters in addition to keeping track of nesting. Instead, many parsers for layout-sensitive languages contain a handwritten component that keeps track of layout and informs a standard parser for context-free languages about relevant aspects of layout, for instance, by inserting special tokens into the token stream. For example, the Python language specification[2] describes an algorithm that preprocesses the token stream to delete some *newline* tokens and insert *indent* and *dedent* tokens when the indentation level changes. Python's context-free grammar assumes that this preprocessing step has already been performed, and uses the additional tokens to recognize layout-sensitive program structure.

This approach has the advantage that a standard parser for context-free languages can be used to parse the preprocessed token stream, but it has the disadvantage that the overall syntax of the programming language is not defined in a declarative, human-readable way. Instead, the syntax is only defined in terms of a somewhat obscure algorithm that explicitly manipulates token streams. This is in contrast to the success story of declarative grammar and parsing technology [11].

Furthermore, a simple algorithm for layout-handling that informs a standard parser for context-free languages is not even enough to parse Haskell. The Haskell language specification describes that a statement ends earlier than visible from the layout if this is the only way to continue parsing [14]. For example, the Haskell program in Figure 3(a) is valid: The statement print (11 + 12) only includes one closing parenthesis, because the second closing parenthesis cannot be consumed inside the statement. An algorithm for layout handling could not decide where

[2] `http://docs.python.org/py3k/reference/`



```
catch ( do print 16
           print (11 +
             12) )
      (\e -> do putStr "error: "
                print e )
```

(a) Exception handler.

```
catch ( do print 16
           print (11 +
             12) ) (\e -> do
           putStr "error: "
           print e )
```

(b) Means the same as (a).

Fig. 3. Both variants of this more complicated Haskell program have valid layout

to end the statement by counting whitespace characters only. Instead, additional information from the context-free parser is needed to decide that the statement needs to end because the next token cannot be consumed. As a second and more extreme example, consider the program in Figure 3(b) that has the same parse tree as the program in Figure 3(a). In particular, the statements belong to different *do*-blocks even though they line up horizontally. These two programs can only be parsed correctly by close cooperation between the context-free part of the parser and the layout-sensitive part of the parser, which therefore have to be tightly integrated. This need for tight integration further complicates the picture with low-level, algorithmic specifications of layout rules prevalent in existing language specifications and implementations.

In this section, we have focused our investigation of layout-sensitive languages on Haskell and Python, but we believe our box model is general enough to explain layout in other languages as well.

3 Declaring Layout with Constraints

Our goal is to provide a high-level, declarative language for specifying and implementing layout-sensitive parsers. In the previous section, we have discussed layout informally. We have visualized layout by boxes around the tokens that belong to a subtree in Figures 2 and 3. We propose (i) to express layout rules formally as constraints on the shape and relative positioning of boxes and (ii) to annotate productions in a grammar with these constraints. The idea of layout constraints is that a production is only applicable if the parsed text adheres to the annotated constraint.

For example, Figure 4 displays an excerpt from our grammar for Haskell that specifies the layout of Haskell *do*-blocks with implicit (layout-based) as well as explicit block structure. This is a standard SDF grammar except that some productions are annotated with layout constraints. For example, the nonterminal Impl stands for implicit-layout statements, that is, statements of the form ⌜□ (but not □ or ◱). The layout constraint `layout("1.first.col < 1.left.col")` formally expresses the required shape ⌜□ for subtree number `1`.

```
context-free syntax
  Stm             ->  Impl    {layout("1.first.col < 1.left.col")}
  Impl            ->  Impls
  Impl Impls      ->  Impls   {cons("StmSeq"), layout("1.first.col == 2.first.col")}
  Stm             ->  Expls
  Stm ";" Expls   ->  Expls   {cons("StmSeq")}
  Impls           ->  Stms    {cons("Stms")}
  "{" Expls "}"   ->  Stms    {cons("Stms"), ignore-layout}
  "do" Stms       ->  Exp     {cons("Do"), longest-match}
```

Fig. 4. Excerpt of our layout-sensitive Haskell grammar. Statements with implicit layout (Impl) have to follow the offside rule. Multiple statements have to align horizontally. Statements with explicit layout (Expl) are not layout-sensitive.

$$\begin{array}{rcl}
tree & ::= & number \\
tok & ::= & tree.\mathsf{first} \mid tree.\mathsf{left} \mid tree.\mathsf{right} \mid tree.\mathsf{last} \\
ne & ::= & tok.\mathsf{line} \mid tok.\mathtt{col} \mid ne \mathbin{\texttt{+}} ne \mid ne \mathbin{\texttt{-}} ne \\
be & ::= & ne \mathbin{\texttt{==}} ne \mid ne \mathbin{\texttt{<}} ne \mid ne \mathbin{\texttt{>}} ne \mid be \mathbin{\texttt{\&\&}} be \mid be \mathbin{\texttt{||}} be \mid \texttt{!}be \\
c & ::= & \mathsf{layout}(be) \mid \mathsf{ignore\text{-}layout}
\end{array}$$

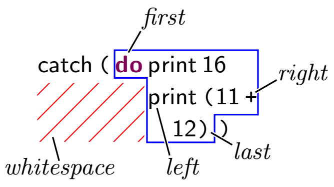
Fig. 5. Syntax of layout constraints c that can annotate SDF productions

We provide the full grammar of layout constraints in Figure 5. Layout constraints can refer to direct subtrees (including terminals) of the annotated production through numerical indexes.

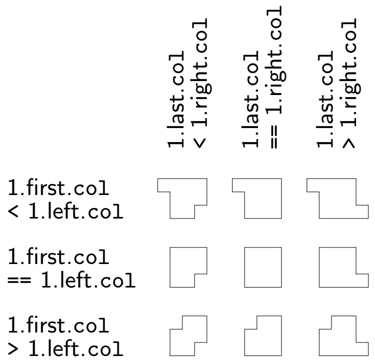
Each subtree exposes its shape via the source location of four tokens in the subtree, which describe the relevant positions in the token stream. Layout constraints use *token selectors* to access these tokens: first selects the first non-whitespace token, last selects the last non-whitespace token, left selects the leftmost non-whitespace token that is not on the same line as the first token, and right selects the rightmost non-whitespace token that is not on the same line as the last token. Figure 6(a) shows how the positions of these tokens describe the shape of a subtree.

It is essential in our design that layout rules can be described in terms of the locations of these four tokens, because this provides a declarative abstraction over the exact shape of the source code. As is apparent from their definition, the token selectors left and right fail if all tokens occur in a single line. Since a single line of input satisfies any box shape, we do not consider this a constraint violation.

For each selected token, the *position selectors* line and `col` yield the token's line and column offset, respectively. Hence the constraint `1.first.col < 1.left.col` specifies that the left border of the shape of subtree 1 must look like □. In other words, the constraint `1.first.col < 1.left.col` corresponds to Landin's offside rule. Consider the following example:

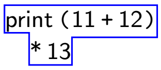


(a) The source locations of four tokens induce (an abstraction of) the shape of a subtree.



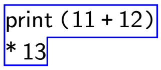
(b) Layout constraints to restrict the shape of a box.

Fig. 6. Example layout constraints and the corresponding boxes



```
print (11 + 12)
   * 13
```

Here, the constraint 1.first selects the first token of the function application, yielding the character p for scannerless parsers, or the token print otherwise. 1.left selects the left-most token not on the first line, that is, the operator symbol *. This statement is valid according to the Impl production because the layout constraint is satisfied: The column in which print appears is to the left of the column in which * appears. Conversely, the following statement does not adhere to the shape requirement of Impl because the layout constraint fails:



```
print (11 + 12)
* 13
```

Consequently, the Impl production is not applicable to this statement.

The layout constraint 1.first.col < 1.left.col mentions only a single subtree of the annotated production and therefore restricts the shape of that subtree. Figure 6(b) shows other examples for layout constraints that restrict the shape of a subtree. In addition to these shapes, layout constraints can also prescribe the vertical structure of a subtree. For example, the constraint 1.first.line == 1.last.line prohibits line breaks within the subtree 1 and 1.first.line + num(2) == 1.last.line requires exactly two line breaks.

If a layout constraint mentions multiple subtrees of the annotated production, it specifies the relative positioning of these subtrees. For example, the nonterminal Impls in Figure 4 stands for a list of statements that can be used with implicit layout. In such lists, all statements must start on the same column. This horizontal alignment is specified by the layout constraint 1.first.col == 2.first.col. This constraint naturally composes with the constraint in the Impl production:

A successful parse includes applications of both productions and hence checks both layout constraints.

The anti-constraint ignore-layout can be used to deactivate layout validation locally. In some languages such as Haskell and Python, this is necessary to support explicit-layout structures within implicit-layout structures. For example, the Haskell grammar in Figure 4 declares explicit-layout statement lists. Since these lists use explicit layout {stmt;...;stmt}, no additional constraints are needed. Haskell allows code within an explicit-layout list to violate layout constraints imposed by surrounding constructs. Correspondingly, we annotate explicit-layout lists with ignore-layout, which enables us to parse the following valid Haskell program:

```
do print (11 + 12)
      print 13
   do { print 14 ;
 print 15 }
   print 16
```

Our Haskell parser successfully parses this program even though the second statement seemingly violates the shape requirement on Impl. However, since the nested explicit statement list uses ignore-layout, we skip all its tokens when applying the left or right token selector. Therefore, the left selector in the constraint of Impl fails to find a leftmost token that is not on the first line, and the constraint succeeds by default.

We deliberately kept the design of our layout-constraint language simple to avoid distraction. For example, we left out language support for abstracting over repeating patterns in layout constraints. However, such facilities can easily be added on top of our core language. Instead, we focus on the integration of layout constraints into generalized parsing.

4 Layout-Sensitive Parsing with SGLR

We implemented a layout-sensitive parser based on our extension of SDF [8] with layout constraints. Our parser implementation builds on an existing Java implementation [10] of scannerless generalized LR (SGLR) parsing [19,21]. A SGLR parser processes all possible interpretations of the input stream in parallel and produces multiple potential parse results. Invalid parse results can be filtered out in an additional disambiguation phase.

We have modified the SGLR parser to take layout constraints into account.[3] As a first naive but correct strategy, we defer all validation of layout constraints until disambiguation time. As an optimization of this strategy, we then identify layout constraints that can be safely checked at parse time.

[3] We can reuse the parse-table generator without modification, because it automatically forwards layout constraints from the grammar to the corresponding reduce-actions in the parse table.

4.1 Disambiguation-Time Rejection of Invalid Layout

SDF distinguishes two execution phases: parse time and disambiguation time. At parse time, the SGLR parser processes the input stream to construct a *parse forest* of multiple potential parser results. This parse forest is input to the disambiguation phase, where additional information (e.g., precedence information) specified together with the context-free grammar is used to discard as many of the trees in the parse forest as possible. Ideally, only a single tree remains, which means that the given SDF grammar is unambiguous for the given input.

While conceptually layout constraints restrict the applicability of annotated productions, we can still defer the validation of layout constraints to disambiguation time. Accordingly, we first parse the input ignoring layout constraints and produce all possible trees. However, to enable later checking of token positions, during parsing we store line and column offsets in the leaves of parse trees.

After parsing, we disambiguate the resulting parse forest by traversing it. Whenever we encounter the application of a layout-constrained production, we check that the layout constraint is satisfied. For violated constraints, we reject the corresponding subtree that used the production. If a layout violation occurs within an ambiguity node, we select the alternative result (if it is layout-correct).

The approach described so far is a generic technique that can be used to integrate any context-sensitive validation into context-free parsing. For instance, Bravenboer et al. [1] integrate type checking into generalized parsing to disambiguate metaprograms. However, layout-sensitive parsing is particularly hard because of the large number of ambiguities even in small programs.

For example, in the following Haskell programs, the number of ambiguities grows exponentially with the number of statements:

```
foo = do print 1
```

```
foo = do print 1
         print 2
```

```
foo = do print 1
         print 2
         print 3
```

For the first program, the context-free parser results in a parse forest with one ambiguity node that distinguishes whether the number 1 is a separate statement or an argument to print. The second example already results in a parse forest with 7 ambiguity nodes; the third example has 31 ambiguity nodes. The number of ambiguities roughly quadruples with each additional statement.

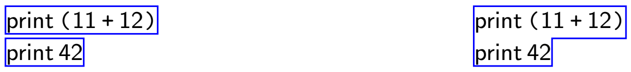
Despite sharing between ambiguous parse trees, disambiguation-time layout validation can handle programs of limited size only. For example, consider the Haskell program that contains 30 repetitions of the statement print 1 2 3 4 5 6 7 8 9. After parsing, the number of layout-related ambiguities in this program is so big that it takes more than 20 seconds to disambiguate it. A more scalable solution to layout-sensitive parsing is needed.

4.2 Parse-Time Rejection of Invalid Layout

The main scalability problem in layout validation is that ambiguities are not local. Without explicit block structure, it is not clear how to confine layout-based ambiguities to a single statement, a single function declaration, or a single class declaration. For example, in the print examples from the previous subsection, a number on the last line can be argument to the print function on the first line. Similarly, when using indentation to define the span of if-then-else branches as in Python, every statement following the if-then-else can be either part the else branch or not. It would be good to restrict the extent of ambiguities to more fine-grained regions at parse time to avoid excessive ambiguities.

Internally, SGLR represents intermediate parser results as states in a graph-structured stack [19]. Each state describes (i) a region in the input stream, (ii) a nonterminal that can generate this input, and (iii) a list of links to the states of subtrees. When parsing can continue in different ways from a single state, the parser splits the state and follows all alternatives. For efficiency, SGLR uses local ambiguity packing [19] to later join such states if they describe the same region of the input and the same nonterminal (the links to subtrees may differ). For instance, in the ambiguous input print (1 + 2 + 3), the arithmetic expression is described by a single state that corresponds to both (1+2)+3 and 1+(2+3). Thus, the parser can ignore the local ambiguity while parsing the remainder of the input.

Due to this sharing, we cannot check context-sensitive constraints at parse time. Such checks would require us to analyze and possibly resplit parse states that were joined before: Two parse states that can be treated equally from a context-free perspective may behave differently with respect to a context-sensitive property. For example, the context-free parser joins the states of the following two parse trees representing different Haskell statement lists:



```
print (11 + 12)                    print (11 + 12)
print 42                           print 42
```

The left-hand parse tree represents a statement list with two statements. The right-hand parse tree represents a statement list with a single statement that spans two lines. This statement violates the layout constraint from the Haskell grammar in Figure 4 because it does not adhere to the offside rule (shape ⊐). Since the context-free parser disregards layout constraints, it produces both statement lists nonetheless.

The two statement lists describe the same region in the input: They start and end at the same position, and both parse trees can be generated by the Impls nonterminal (Figure 4). Therefore, SGLR joins the parse states that correspond to the shown parse trees. This is a concrete example of two parse trees that differ due to a context-sensitive property, but are treated identically by SGLR.

Technically, context-sensitive properties require us to analyze and possibly split parse states that are not root in the graph-structured stack. Such a split deep in the stack would force us to duplicate all paths from root states to the split state. This not only entails a serious technical undertaking but likely degrades the parser's runtime and memory performance significantly.

To avoid these technical difficulties, we would like to enforce only those layout constraints at parse time that do not interact with sharing. Such constraints must satisfy the following invariant: If a constraint rejects a parse tree, it must also reject all parse trees that the parser might represent through the same parse state. For constraints that satisfy this invariant, it cannot happen that we prematurely reject a parse state that should have been split instead: Each tree represented by such state would be rejected by the constraint. In particular, such constraints only use information that is encoded in the parse state itself, namely the input region and the nonterminal. This information is the same for all represented trees and we can use it at parse time to reject states without influencing splitting or joining.

In our constraint language, the input region of a tree is described by the token selectors first and last. Since the input region is the same for all trees that share a parse state, constraints that only use the first and last token selectors (but not left or right) can be *enforced at parse time* without influencing sharing: If such a constraint rejects any random tree of a parse state, the constraint also rejects all other trees because they describe the same input region.

One particularly useful constraint that only requires the token selectors first and last is `1.first.col == 2.first.col`, which denotes that trees `1` and `2` need to be horizontally aligned. Such constraint is needed for statement lists of Haskell and Python. Effectively, the constraint reduces the number of potential statements to those that start on the same column. This confines many ambiguities to a single statement. For example, the constraint allows us to reject the program shown in Figure 2(b) at parse time because the statements are not aligned. However, it does not allow us to reject or distinguish the programs shown in Figure 2(a) and 2(c); we retain an ambiguity that we resolve at disambiguation time.

Technically, we enforce constraints at parse time when executing reduce actions. Specifically, in the function DO-REDUCTIONS [21], for each list of subtrees, we validate that the applied production permits the layout of the subtrees. We perform the regular reduce action if the production does not specify a layout constraint, or the constraint is satisfied, or the constraint cannot be checked at parse time. If a layout constraint is violated, the reduce action is skipped.

The remaining challenge is to validate that we in fact reduce ambiguity to a level that allows acceptable performance in practice.

5 Evaluation

We evaluate *correctness* and *performance* of our layout-sensitive generalized parsing approach with an implementation of a Haskell parser. Correctness is interesting because we reject potential parser results based on layout constraints; we expect that layout should not affect correctness. Performance is critical because our approach relies on storing additional position information and creating additional ambiguity nodes that are later resolved, which we expect to have a negative influence on performance. We want to assess whether the performance penalty of our approach is acceptable for practical use (e.g., in an IDE). Specifically, we evaluate the following research questions:

RQ1: Can a layout-sensitive generalized Haskell parser parse the same files and produce equivalent parse trees as a layout-insensitive Haskell parser that requires explicit layout?

RQ2: What is the performance penalty of the layout-sensitive Haskell parser compared to a layout-insensitive Haskell parser that requires explicit layout?

5.1 Research Method

In a controlled setting, we quantitatively compare the results and performance of different Haskell parsers on a large set of representative Haskell files.

Parsers and parse results. We have implemented the layout-sensitive parser as discussed above by modifying the original SGLR parser written in Java.[4] We have extended an existing SDF grammar for Haskell that required explicit layout[5] with layout constraints. We want to compare our parser to a reimplementation of GHC's hand-tuned LALR(1) parser that has been developed by others and is deployed as part of the haskell-src-exts package.[6] Here, we refer to it simply as GHC parser. However, comparing the performance of our layout-sensitive SGLR parser to the hand-optimized GHC parser would be unfair since completely different parsing technologies are used. Also comparing the produced abstract syntax trees of both parsers is not trivial, because differently structured abstract syntax trees are generated. Therefore, we primarily compare our layout-sensitive parser to the original SGLR parser that did not support layout.

However, the original SGLR parser is layout-insensitive and therefore not able to parse Haskell files that use implicit layout (which almost all Haskell files do). Therefore, we also used the pretty printer of the haskell-src-exts package to translate Haskell files with arbitrary combinations of explicit and implicit layout into a representation with only explicit layout. Since the pretty printer also removes comments, the files may be smaller and hence faster to parse. Therefore, we use the same pretty printer to create a file that uses only implicit layout and contains no comments either.

Overall, we have three parsers (GHC, the original SGLR parser, and our layout-sensitive SGLR parser) which we can use to parse three different files (original layout, explicit-only layout, implicit-only layout). We are interested in the parser result and parse time of four combinations:

`GHC`. Parsing the file with *original layout* using the GHC parser.

`SGLR-Orig`. Parsing the file with *original layout* (possible mixture of explicit and implicit layout) with our layout-sensitive SGLR parser.

`SGLR-Expl`. Parsing the file after pretty printing with *explicit layout only* and without comments with the original SGLR parser.

`SGLR-Impl`. Parsing the file after pretty printing with *implicit layout only* and without comments with our layout-sensitive SGLR parser.

We illustrate the process, the parsers, and the results in Figure 7. All SGLR-based parsers use the same Haskell grammar of which the original SGLR parser

[4] Actually, we improved the original implementation by eliminating recursion to avoid stack overflows when parsing files with long comments or long literal strings.

[5] `http://strategoxt.org/Stratego/HSX`

[6] `http://hackage.haskell.org/package/haskell-src-exts`

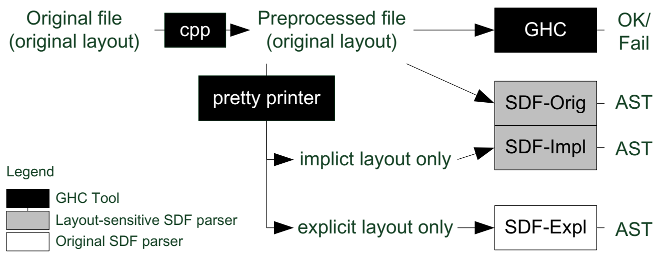


Fig. 7. Evaluation setup

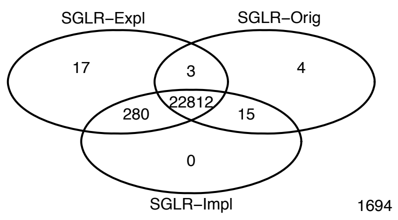
ignores the layout constraints. Our Haskell grammar implements the Haskell 2010 language report [14], but additionally supports the following extensions to increase coverage of supported files: *HierarchicalModules*, *MagicHash*, *FlexibleInstances*, *FlexibleContexts*, *GeneralizedNewtypeDeriving*. We configured the GHC parser accordingly and, in addition, deactivated its precedence resolution of infix operators, which is a context-sensitive mechanism that can be implemented as a post-processing step. Running the C preprocessor is necessary in many files and performed in all cases. Note that `SGLR-Orig` and `SGLR-Impl` use the same parser, but execute it on different files.

Subjects. To evaluate performance and correctness on realistic files, we selected a large representative collection of Haskell files. We attempt to parse all Haskell files collected in the open-source Haskell repository Hackage.[7] We extracted the latest version of all 3081 packages that contain Haskell source code on May 15, 2012. In total, these packages contain 33 290 Haskell files that amount to 258 megabytes and 5 773 273 lines of code (original layout after running cpp).

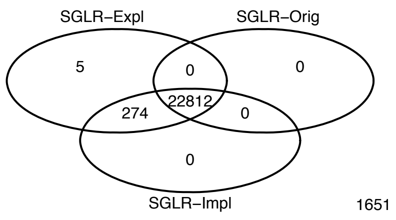
Data collection. We perform measurements by repeating the following for each file in Hackage: We run the C preprocessor and the pretty printer to create the files with original, explicit-only, and implicit-only layout. We measure the wall-clock time of executing the GHC parser and the SGLR-based parsers on the prepared files as illustrated in Figure 7. We stop parsers after a timeout of 30 seconds and interpret longer parsing runs as failure. We parse all files in a single invocation of the Java virtual machine and invoke the garbage collector between each parser execution. After starting the virtual machine, we first parse 20 packages (215 files) and discard the results to account for warmup time of Java's JIT compiler. A whole run takes about 6 hours. We repeat the entire process with all measurements three times after system reboots and use the arithmetic mean of each file and parser over all runs.

We run all performance measurements on the same 3 GHz, dual-core machine with 4GB memory and Java Hotspot VM version 1.7.0_04. We specified a maximum heap size of 512MB and a maximum stack size of 16MB.

[7] `http://hackage.haskell.org`



(a) Number of files each parser produces the correct AST for.



(b) Ignoring files that timeout with at least one parser.

Fig. 8. Correctness of layout-sensitive parsing

Analysis procedure. We discard all files that cannot be parsed by the GHC parser configured as described above. On the remaining files, for research question RQ1 (correctness), we evaluate that the three abstract syntax trees produced by SGLR parsers are the same (that is, we perform a form of differential testing).

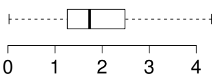
For research question RQ2 (performance penalty), we determine the relative slow down between `SGLR-Expl` and `SGLR-Impl`. We calculate the relative performance penalty between parsers separately for each file that can be parsed by all three parsers. We report the geometric mean and the distribution of the relative performance of all these files.

5.2 Results

Correctness. Of all 33 290 files, 9071 files (27 percent) could not be parsed by the GHC parser (we suspect the high failure rate is due to the small number of activated language extensions). Of the remaining 24 219 files, 22 812 files (94 percent) files could be parsed correctly with all three SGLR-based parsers (resulting in the same abstract syntax tree). We show the remaining numbers in the Venn diagram in Figure 8(a). Some differences are due to timeouts; the diagram in Figure 8(b) shows those results that do not time out in any parser.

Performance. The median parse times per file of all parsers are given in Figure 9(b). Note that the results for `GHC` are not directly comparable, since they include a process invocation, which corresponds to an almost constant overhead of 15 ms. On average `SGLR-Impl` is 1.8 times slower than `SGLR-Expl`. We show the distribution of performance penalties as box plot in Figure 9(a) (without outliers). The difference between `SGLR-Orig` and `SGLR-Impl` is negligible; `SGLR-Impl` is slightly faster on average because pretty printing removes comments.

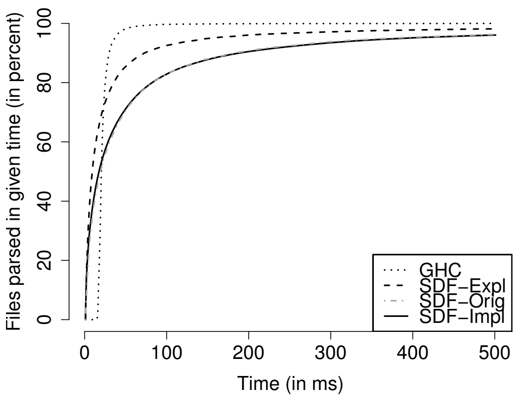
In Figure 9(c), we show the parse times for all four parsers (the graph shows how many percent of all files can be parsed within a given time). We see that, as to be expected, `SGLR-Expl` is slower than the hand-optimized `GHC`, and `SGLR-Impl` is slower than `SGLR-Expl`. The parsers `SGLR-Impl` and `SGLR-Orig` perform similarly and are essentially not distinguishable in this figure.



(a) Distribution of relative performance penalty (SGLR-Impl/SGLR-Expl).

	median time/file
GHC	(<)19 ms
SGLR-Expl	8 ms
SGLR-Orig	18 ms
SGLR-Impl	17 ms

(b) Median parse times.



(c) Distribution of parsing times.

Fig. 9. Performance of layout-sensitive parsing

5.3 Interpretation and Discussion

As shown in Figure 8(a), SGLR-Orig and SGLR-Impl do not always produce the same result as SGLR-Expl. Of these differences, 40 can be ascribed to timeouts, which occur in SGLR-Expl as well as in SGLR-Orig and SGLR-Impl. The remaining differences are shown in Figure 8(b). We investigated these differences and found that the five files that only SGLR-Expl can parse are due to Haskell statements that start with a pragma comment, for example:

```
{-# SCC "Channel_Write" #-} liftIO . atomically $ writeTChan pmc m
```

Since our SGLR-based parsers ignore such pragma comments, the statement appears to be indented too far. We did not further investigate due to the low number of occurrences of this pattern.

For the 274 files that only SGLR-Expl and SGLR-Impl can parse, we took samples and found that SGLR-Orig failed because of code that uses a GHC extension called *NondecreasingIndentation*, which is not part of the Haskell 2010 language report but cannot be deactivated in the GHC parser. The extension allows programs to violate the offside rule for nested layout blocks:

```
foo = do                                  foo = do
print 16                                  print 16
do                pretty-prints to        do
print 17                                  print 17
print 18                                  print 18
```

None of the SGLR-based parsers can handle such programs. However, the GHC pretty printer always produces code that complies with the offside rule. Thus, SGLR-Expl and SGLR-Impl can parse the pretty-printed code, whereas SGLR-Orig fails on the original code. We consider this a bug of the reimplementation of the

GHC parser, which does not implement the Haskell 2010 language report even when configured accordingly.

Finally, GHC accepts 1651 files that none of the SGLR-based parsers accepts. Since not even the layout-insensitive parser `SGLR-Expl` accepts these files, we suspect inaccuracies in the original Haskell grammar independent of layout.

Regarding performance, layout-sensitive parsing with `SGLR-Impl` entails an average slow down of 1.8 compared to layout-insensitive parsing with `SGLR-Expl`. Given the median parse times per file (Figure 9(b)), this slow down is still in the realm of a few milliseconds and suggests that layout-sensitive parsing can be applied in practice. In particular, this slow down seems acceptable given the benefits of declarative specifications of layout as in our approach, as opposed to low-level implementation of layout within a lexer or the parser itself. Furthermore, we expect room for improving the performance of our implementation of layout-sensitive parsing, as we discuss in Section 6.

Overall, regarding correctness (RQ1), we have shown that layout-sensitive parsing can parse almost all files that the layout-insensitive `SGLR-Expl` can parse. In fact, we did not find a single actual difference that would indicate an incorrect parse. Regarding performance penalty (RQ2), we believe that the given slow down does not inhibit practical application of our parser.

5.4 Threats to Validity

A key threat to external validity (generalizability of the results) is that we have analyzed only Haskell files and parse only files from the Hackage repository. We believe that the layout mechanisms of Haskell are representative for other languages, but our evaluation cannot generalize beyond Haskell. Furthermore, files in Hackage have a bias toward open-source libraries. However, we believe that our sample is large enough and the files in Hackage are diverse enough to present a general picture.

An important threat to internal validity (factors that allow alternative explanations) is the pretty printing necessary for parser `SGLR-Expl`. Pretty printing removes comments but possibly adds whitespace. The pretty-printed files with explicit layout have a 45 percent larger overall byte size compared to original layout, whereas the pretty-printed files with implicit layout have a 15 percent smaller byte size. Unfortunately, we have no direct influence on the pretty printer. We believe that the influence of pretty printing is largely negligible, because whitespace and comments should not trigger ambiguities (the similarity of the performance of `SGLR-Orig` and `SGLR-Impl` can be seen as support).

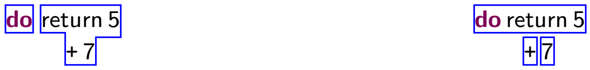
It may be surprising that GHC (and also `SGLR-Orig`) fail to parse over one quarter of all files. We have sampled some of these files and found that they require more language extensions than we currently support. For example, the GADTs and TypeFamilies extensions seem to be popular, but we did not implement their syntax in our grammar and deactivated them in the GHC parser. In future work, we would like to support Haskell more completely, which should increase the number of supported Hackage files.

Regarding construct validity (suitability of metrics for evaluation goal), we measured performance using wall-clock time only. For the SGLR-based parsers, we control JIT compilation with a warmup phase. By running the garbage collector between parser runs and monitoring the available memory, we ensured that all parsers have a similar amount of memory available. However, the layout-aware parser stores additional information and may perform different in scenarios with less memory available. Furthermore, we can, of course, not entirely eliminate background noise. Although we have repeated all measurements only three times, we believe the measurements are sufficiently clear and we have checked that variations between the three measurements are comparably minor for all parsers (for over 95 percent of all files, the standard deviation of these measurements was less than 10 percent of the mean).

6 Discussion and Future Work

We modified an SGLR parser to support validation of layout constraints at parse time and disambiguation time. Here, we summarize some technical implications, potential improvements, and limitations of our parser.

Technical implications. Layout-sensitive parsing interacts with traditional disambiguation methods such as priorities or follow restrictions. For example, consider the following Haskell program, which can be parsed into two layout-correct parse trees (boxes indicate the toplevel structure of the trees):



In both parse trees, the *do*-block consists of a single statement that adheres to the offside rule. However, the Haskell language report specifies that the left-hand parse tree is correct: For *do*-blocks the *longest match* needs to be selected.

SDF provides a longest-match disambiguation filter for lexical syntax, called follow restrictions [20]. A typical use of follow restrictions is to ensure that identifiers are not followed by any letters, which should be part of the identifier instead. Since, in fact, both of the above parse trees correspond to some valid Haskell program (dependent on layout), not even context-free follow restrictions enable us to disambiguate correctly because they ignore layout. Similarly, a priority filter would reject the same parse tree irrespective of layout.

For this reason, we added a disambiguation filter to SDF called `longest-match`. We use it to declare that, in case of ambiguity, a production should extend as far to the right as possible. We annotated the production for *do*-blocks in Figure 4 accordingly. Since our parser stores position information in parse trees anyway, the implementation of the longest-match filtering is simple: For ambiguous applications of a longest-match production we compare the position of the `last` tokens and choose the tree that extends further.

More generally, it should be noted that due to position information in parse trees, our parser supports less sharing than traditional GLR parsers do. Our parser can only share parse trees that describe the same region in the input stream. We have not yet investigated the implications on memory consumption, but our empirical study indicates that the performance penalty is acceptable.

Performance improvements. In our implementation of layout-sensitive generalized parsing, we mostly focused on correctness and only addressed performance in so far as it influences the feasibility of our approach. Therefore, in our current implementation, we suspect two significant performance improvements are still possible. First, we *interpret* layout constraints by recursive-descent with dynamic type checking. We have profiled the performance of our parser and found that about 25 percent of parse time and disambiguation time are spent on interpreting layout constraints. We expect that a significant improvement is possible by *compiling* layout constraints when loading the parse table. Second, our current implementation validates *all* layout constraints at disambiguation time. However, we validate many constraints at parse time already (as described in Section 4.2). We suspect that avoiding the repeated evaluation of those constraints represents another significant performance improvement.

Limitations. In general, context-sensitive properties can be validated after parsing at disambiguation time without restriction. However, the expressivity of our constraint language is limited in multiple ways. First, layout constraints in our language are *compositional*, that is, a constraint can only refer to the direct subtrees of a production. It might be useful to extend our constraint language with pattern-matching facilities as known, for example, from XPath. However, it is not obvious how such pattern matching influences the performance of parsing and disambiguation; we leave this question open. A second limitation is that we focus on one-dimensional layout-sensitive languages only. However, a few layout-sensitive languages employ a two-dimensional syntax, for example, for type rules as in Epigram [16]. We would like to investigate whether our approach to layout-sensitivity generalizes to two-dimensional parsers.

7 Related Work

We have significantly extended SDF's frontend [8] and its SGLR backend [19,21] to support layout-sensitive languages declaratively. We are not aware of any other parser framework that provides a *declarative* mechanism for layout-sensitive languages. Instead, existing implementations of parsers for layout-sensitive languages are handwritten and require separate layout-sensitive lexing.

For example, the standard Python lexer and parser are handwritten C programs.[8] While parsing, the lexer checks for changes of the indentation level in the input, and marks them with special *indent* and *dedent* tokens. The parser then consumes these tokens to process layout-sensitive program structures. This implementation is non-declarative.

[8] `http://svn.python.org/projects/python/trunk/Modules/parsermodule.c`

As another example, the GHC Haskell compiler employs a layout-sensitive lexer that uses the Lexer generator Alex[9] in combination with manual Haskell code. The generated layout-sensitive lexer manages a stack of layout contexts that stores the beginning of each layout block. When the parser queries the lexer for layout-relevant tokens (such as curly braces), the lexer adapts the layout context accordingly. These interactions between parser and lexer are non-trivial and require virtual tokens for implicit layout. Since the layout rules of Haskell are hard-coded into the lexer, it is also not easy to adapt the parser and lexer for other languages. The same holds for the Utrecht Haskell Compiler [2].

Data-dependent grammars [9] support the declaration of constraints to restrict the applicability of a production. However, constraints in data-dependent grammars must be context-insensitive [9, Lemma 4], and therefore cannot be used to describe languages with context-sensitive layout such as Haskell.

8 Conclusion

We have presented a parser framework that allows the declaration of layout constraints within a context-free grammar. Our generalized parser enforces constraints at parse time when possible but fully validates parse trees at disambiguation time. We have empirically shown that our parser is correct and the performance penalty is acceptable compared to layout-insensitive generalized parsing. We believe that this work will enable language implementors to specify the grammar of their layout-sensitive languages in a high-level, declarative way.

Our original motivation for this work was to develop a syntactically extensible variant of Haskell in the style of SugarJ [4], where regular programmers write syntactic language extensions. This requires a declarative and composable syntax formalism as provided by SDF [8,3]. Based on the work presented here, we have been able to implement SugarHaskell [5], an extensible preprocessor and IDE for Haskell.

Acknowledgments. We thank Doaitse Swierstra for discussion and challenging Haskell examples, and the anonymous reviewers for their feedback. This work is supported in part by the European Research Council, grant No. 203099.

References

1. 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)
2. Dijkstra, A., Fokker, J., Swierstra, S.D.: The architecture of the Utrecht Haskell compiler. In: Proceedings of Haskell Symposium, pp. 93–104. ACM (2009)
3. Erdweg, S., Giarrusso, P.G., Rendel, T.: Language composition untangled. In: Proceedings of Workshop on Language Descriptions, Tools and Applications, LDTA (to appear, 2012)

[9] http://www.haskell.org/alex/

4. Erdweg, S., Rendel, T., Kästner, C., Ostermann, K.: SugarJ: Library-based syntactic language extensibility. In: Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 391–406. ACM (2011)
5. Erdweg, S., Rieger, F., Rendel, T., Ostermann, K.: Layout-sensitive language extensibility with SugarHaskell. In: Proceedings of Haskell Symposium, pp. 149–160. ACM (2012)
6. Ford, B.: Packrat parsing: Simple, powerful, lazy, linear time, functional pearl. In: Proceedings of International Conference on Functional Programming (ICFP), pp. 36–47. ACM (2002)
7. Ford, B.: Parsing expression grammars: A recognition-based syntactic foundation. In: Proceedings of Symposium on Principles of Programming Languages (POPL), pp. 111–122. ACM (2004)
8. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The syntax definition formalism SDF – reference manual. SIGPLAN Notices 24(11), 43–75 (1989)
9. Jim, T., Mandelbaum, Y., Walker, D.: Semantics and algorithms for data-dependent grammars. In: Proceedings of Symposium on Principles of Programming Languages (POPL), pp. 417–430. ACM (2010)
10. Kats, L.C.L., de Jonge, M., Nilsson-Nyman, E., Visser, E.: Providing rapid feedback in generated modular language environments: Adding error recovery to scannerless generalized-LR parsing. In: Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 445–464. ACM (2009)
11. Kats, L.C.L., Visser, E., Wachsmuth, G.: Pure and declarative syntax definition: Paradise lost and regained. In: Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 918–932. ACM (2010)
12. Landin, P.J.: The next 700 programming languages. Communication of the ACM 9(3), 157–166 (1966)
13. Leijen, D., Meijer, E.: Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Universiteit Utrecht (2001)
14. Marlow, S. (ed.): Haskell 2010 language report (2010), `http://www.haskell.org/onlinereport/haskell2010`
15. Mason, T., Brown, D.: Lex & yacc. O'Reilly (1990)
16. McBride, C.: Epigram: Practical Programming with Dependent Types. In: Vene, V., Uustalu, T. (eds.) AFP 2004. LNCS, vol. 3622, pp. 130–170. Springer, Heidelberg (2005)
17. Parr, T., Fisher, K.: LL(*): The foundation of the ANTLR parser generator. In: Proceedings of Conference on Programming Language Design and Implementation (PLDI), pp. 425–436. ACM (2011)
18. Parr, T., Quong, R.W.: ANTLR: A predicated-LL(k) parser generator. Software Practice and Experience 25, 789–810 (1994)
19. Tomita, M.: An efficient augmented-context-free parsing algorithm. Computational Linguistics 13(1-2), 31–46 (1987)
20. den van Brand, M.G.J., Scheerder, J., Vinju, J.J., Visser, E.: Disambiguation Filters for Scannerless Generalized LR Parsers. In: Nigel Horspool, R. (ed.) CC 2002. LNCS, vol. 2304, pp. 143–158. Springer, Heidelberg (2002)
21. Visser, E.: Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam (1997)

PAPAGENO: A Parallel Parser Generator for Operator Precedence Grammars

Alessandro Barenghi, Ermes Viviani, Stefano Crespi Reghizzi, Dino Mandrioli, and Matteo Pradella

Dipartimento di Elettronica e Informazione - Politecnico di Milano
{barenghi,viviani,crespi,mandrioli,pradella}@elet.polimi.it

Abstract. In almost all language processing applications, languages are parsed employing classical algorithms (such as the LR(1) parsers generated by Bison), which are sequential due to their left-to-right state-dependent nature. Although early theoretical studies on parallel parsing algorithms delineated potential speedups on abstract parallel machines using a data-parallel approach, practical developments have not materialized, except in recent experiments on ad hoc parsers for large XML files. We describe a general-purpose practical generator (PAPAGENO) able to produce efficient deterministic parallel parsers, which exhibit significant speedups when parsing large texts on modern multi-core machines, while not penalizing sequential operation. The generated parser relies on the properties of Floyd's operator precedence grammars, to provide a naturally parallel implementation of the parsing process. Parsing of each text portion proceeds in parallel and independently, without communication and synchronization, until all partial parse stacks are recombined into the final result. Since Floyd's grammars can express most syntaxes with little adaptation, we have performed extensive experiments, on both synthetically generated texts and real JSON documents. The effective parallel code portion in the generated parsers exceeds 80% for most of the tested scenarios.

Keywords: Parser generation, Parallel Parsing, Floyd Operator Precedence Grammars.

1 Introduction

Language parsing, also known as syntactic analysis, occurs in many situations: compilation, natural language processing, document browsing, genome sequencing, program analysis targeted at detecting malicious behaviours, and others. Although syntactic analysis is less computationally demanding than semantic analysis, it is frequently applied to very large data sets in contexts where speedups and related energy saving are often important. The common linear-time left-to-right LR(1) and LL(1) algorithms used for deterministic context-free (or BNF) languages are an important milestone of algorithmic research. Due to their ability to recognise a wider class of formal languages, they superseded earlier algorithms such as the ones employing Floyd's *operator-precedence grammars* (FG), which rely on only local information to decide parsing steps (for an introduction see e.g. [1]). The LR and LL algorithms are amenable to efficient implementations on serial computing machines, such as the ones provided in popular parser generators (e.g. Bison), but their structure hinders efficient parallelization: early attempts at it have not been successful, and appear to be almost abandoned.

In this work we describe a general-purpose optimized FG-based parallel-parser generator. As mentioned, FGs have already been used successfully to define early programming languages; they have a very fast sequential parser, which is still used by modern compilation platforms (gcc for instance) to parse expressions with multiple operator precedences. Recently, research on formal methods has renewed the interest for FGs, thanks to their nice closure and decidability properties [2]. In particular, the relevant feature distinguishing FG languages from LR and LL ones is the closure under the substring extraction; this property enables independent parallel parsing of substrings. Starting from a straightforward sequential parser, we have implemented a parser generator producing two versions of an associative reduction scheme, and measured consistent speedups on both synthetic benchmarks and large real JSON files. Measurements indicate good scalability on different multi-core architectures, leading to significant reductions of parsing time with respect to state-of-the-art sequential parsers.

Related research. The straightforward idea of splitting a long text into chunks, to be parsed in parallel and subsequently recombined by further parsing actions, has motivated several studies in the period 1965-1990. Early theoretical studies on data-parallel algorithms such as the ones reported in [3,4] determined the computational complexity that can be obtained in principle on certain abstract parallel machines by using a data-parallel approach.

Parallel parsing requires an algorithm that, unlike a classical parser, is able to process *substrings* which are not syntactically legal, although they occur in legal strings. A substring parser operates on a substring without any knowledge of the outcome of parsing left and right contexts. Incidentally, substring parsing algorithms have been also studied for different purposes, such as processing damaged or faulty texts. Notable examples of substring parsing research have adapted shift-reduce LR(1) parsers, by dropping the condition that the text to the left of the chunk under examination should have been parsed already. The first, and simpler approach due to Mickunas and Schell [5], modifies an LR(1) parser in order to start in several possible states and to scan the chunk as far as deterministically possible. However, a significant amount of the computation is typically left to the chunk recombination phase as this may propagate changes on all the chunks preceding the one being recombined. In other approaches, such as [6] and [7], each chunk parser carries on all possible alternative parses and subsequently, it recombines the parsed chunks: assuming the original grammar to be LR(1), this chunk parsing can be done in linear time although with constants greater than the ones for shift-reduce parsing.

The few people who have performed some limited experimentation on such algorithms have generally found that performances critically depend on the cut points between chunks: if a chunk starts, say, with begin, the parser can recognize almost completely a full language block. On the contrary, starting a chunk on an identifier opens too many syntactic alternatives. As a consequence such parsers have been typically combined with language-dependent heuristics for splitting the source text into chunks that start on keywords announcing a splitting friendly construct.

After a long intermission, research in the field of parallel parsing has recently resumed with more practical goals, such as to parse XML documents on multicore machines, both servers and clients. Some published works, e.g., [8] and [9], rely explicitly on the assumption that the parsed language is either XML or a subset of it, in order to devise ad-hoc strategies to extract parallelism from the parsing process. In other words, such projects no longer qualify as general-purpose parsers. Although specialized parsers, say, for XML, may be of interest to particular communities, our project is the first to develop a general-purpose language-independent parallel parser generator, and validate it with experiments on real-world benchmarks on modern architectures. In addition to this, we provide guidelines for the implementation and optimization of the algorithm, such as the design of a unified data structure for the abstract syntax tree and parser stack representation, which has proved instrumental for obtaining the high code fraction executed in parallel, reported in the experimental results.

The paper is organized as follows: Section 2 provides the background and the definitions regarding FGs, Section 3 proposes our parallel parsing algorithm, Section 4 delineates the implementation strategies employed and reports the results of the experimental validation campaign. Finally, Section 5 draws our conclusions.

2 Definitions and Background on Operator Precedence Grammars

Since *Floyd*'s operator precedence *Grammars* (FG) and parsers are a classical technique for syntax definition and analysis, it suffices to recall the main relevant concepts from e.g. [1]. Let Σ denote the *terminal alphabet* of the language. A BNF grammar in *operator form* consists of a set of productions P of the form $A \to \alpha$ where A is a *nonterminal* symbol and α, called the right-hand side (rhs) of the production, is a nonempty string made of terminal and nonterminal symbols, such that if nonterminals occur in α, they are separated by at least one terminal symbol. The set of nonterminals is denoted by V_N. It is well known that any BNF grammar can be recast into operator form. To qualify as FG, an operator grammar has to satisfy a condition, known as absence of precedence conflicts. We will now introduce informally the concept of *precedence relation*, a partial binary relation over the terminal alphabet, which can take one of three values: $\lessdot$ *(yields precedence)* , $\gtrdot$ *(takes precedence)* , $\doteq$ *(equal in precedence).* For a given FG, the precedence relations are easily computed and represented in the *operator precedence matrix* (OPM). A grammar for simple arithmetic expressions and the corresponding OPM are in Fig. 1. Entries like $+ \lessdot a$ and $a \gtrdot +$ indicate that, when parsing a string containing the pattern $\ldots + a + \ldots$, the rhs a of rule $F \to a$ has to be reduced to the nonterminal F. Similarly the pattern $\ldots + (E) \times \ldots$ is reduced by

Grammar G consists of $\Sigma = \{a, +, \times, (,)\}, V_N = \{E, T, F\},$ axiom $= E$ and

$$P = \{E \to E + T \mid T, \qquad T \to T \times F \mid F, \qquad F \to (E) \mid a\}$$

Operator precedence matrix:

$$M_2 = \begin{array}{c|c|c|c|c|c|} & a & + & \times & (&) \\ \hline a & & \gtrdot & \gtrdot & & \\ \hline + & \lessdot & \gtrdot & \lessdot & \lessdot & \gtrdot \\ \hline \times & \lessdot & \gtrdot & \gtrdot & \lessdot & \gtrdot \\ \hline (& \lessdot & \lessdot & \lessdot & \lessdot & \doteq \\ \hline) & & \gtrdot & \gtrdot & & \gtrdot \\ \hline \end{array}$$

Fig. 1. Example of FG for arithmetic expressions

rule $F \to (E)$ to $\ldots + F \times \ldots$ since the relations are $\ldots + \lessdot (\overset{\doteq}{E}) \gtrdot \times \ldots$. There is no relation between terminals a and (because they never occur as adjacent or separated by a nonterminal. A grammar is FG if for any two terminals, at most one precedence relation holds. In sequential parsers it is customary to enclose the input string between two special characters $\perp$, such that $\perp$ yields precedence to any other character and any character takes precedence over $\perp$.

Precedence relations precisely determine if a substring matching a rhs should be reduced to a nonterminal. This test is very efficient, based on local properties of the text, and does not need long distance information (unlike the tests performed by LR(1) parsers). In case the grammar includes two productions such as $A \to x$ and $B \to x$ with the same rhs, the reduction of string x leaves the choice between A and B open. The uncertainty could be propagated until just one choice remains open, but, to avoid this minor complication, we assume without loss of generality, that the grammar does not have repeated rhs's [2].

The mentioned local properties suggest that FG are an attractive choice for data-parallel parsing, but even for sequential parsing, they are very efficient [1]: "Operator-precedence parsers are very easy to construct and very efficient to use, operator-precedence is the method of choice for all parsing problems that are simple enough to allow it". In practice, even when the language reference grammar is not a FG, small changes permit to obtain an equivalent FG, except for languages of utmost syntactic complexity. This is witnessed by the JSON grammar employed as our benchmark, described in Section 4.

3 PAPAGENO

3.1 Parallel Parsing Algorithm

As the parallel algorithm implemented by our generator stems directly from sequential FG parser, we first describe the latter, providing the extension to the parallel technique afterwards. As far as we know, this is the first design and realization of a parallel parsing algorithm for FGs. We also note that this algorithm performs an effective parse of the token stream and is thus different from parallel bracket matching algorithms such as the one presented by Cole [10]. The key idea driving Algorithm 1 is that, wherever a series of $\doteq$ precedence relations enclosed by a pair of $\lessdot, \gtrdot$ is found between adjacent tokens, the enclosed symbol string is the handle of a reduction. To find handles, the parser uses the operator precedence matrix OPM and a (pushdown) stack S to keep track of the tokens to be reduced when the next $\gtrdot$ relation is found. As the parsing algorithm needs to recognise a particular grammar rule (e.g. for building the Abstract Syntax Tree AST representation) upon reduction, the list of productions P is needed to detect the actual production to be applied. Provided that the algorithm is adapted to always shift nonterminal symbols, it is possible to reuse it for all the stages of parallel parsing with no modifications.

In the case of a serial parsing the algorithm takes an empty stack S as a parameter, and the list of input tokens as I, which can be thought as delimited by two special symbols marking the beginning and the end of the token stream. Algorithm 1 operates as follows: it obtains the symbol under the cursor and checks its precedence relation with the terminal symbol occupying the highest place on the parsing stack (lines 3–4). If the precedence relation is not $\gtrdot$ or the symbol is not a terminal, the symbol is pushed

Algorithm 1. Floyd Grammar Parser

```
Globals: $OPM$: Precedence matrix of $G$, $P$: List of productions of $G$
Input: $I$: Input symbol list, $S$: Parsing stack
Output: $S$: the parsing stack after the parsing action
1  begin
2      while $I \neq \emptyset$ do
3          token ← READ_CURSOR($I$)
4          prec ← $OPM$(token,TOP_TERMINAL($S$))
5          if prec $\neq \gtrdot$ or IS_NONTERMINAL(token) then
6              PUSH($S$ , (token, prec))
7              MOVE_CURSOR_FORWARD($I$)
8          else
9              repeat
10                 (token , prec) ← POP($S$)
11                 PUSH(rhs_rebuild , token)
12             until prec= $\lessdot$
13             if $\exists p \in P \mid$ RHS($p$) = rhs_rebuild then
14                 SEMANTIC_ACTION(LHS($p$))
15                 PUSH($S$ , (LHS($p$),$\perp$))
16             else
17                 return NIL
18     return $S$
```

on the top of the stack and the cursor is moved forward by one position (lines 5–7). If the parsing algorithm meets a $\gtrdot$ precedence relation, it needs to rebuild the rhs of the corresponding rule to perform the proper reduction action: this is performed through an auxiliary stack, rhs_rebuild, where the algorithm stores the elements from the top of the stack, until it finds a $\lessdot$ precedence relation. Upon finding the $\lessdot$ precedence relation, the algorithm checks if the rebuilt rhs is a valid production of the grammar and performs the reduction action if this is the case. If the rebuilt rhs is not a valid production the algorithm terminates abnormally signalling that the input string is not valid through returning NIL. If the serial parsing procedure terminates correctly, Algorithm 1 is expected to return a parsing stack containing only the axiom of G.

Exploiting the fact that the parser makes the decision whether to shift or to reduce only on the basis of the precedence matrix OPM, the current token, and the top of the stack, it is possible to divide the token stream into different chunks or substrings, and perform a substantial amount of the parsing with different workers. The essential quality of this algorithm is that all the parsing actions performed on a chunk are final, i.e. no parsing work is ever undone on a substring, thus all the parsing actions performed by the worker threads are correct and useful. To this end, the input is split into as many chunks as the desired number of workers w, and all the workers run the aforementioned algorithm returning their stacks at the end.

Since the splitting of the token stream is not constrained in any way, nor it depends on the language grammar, the result returned by a worker will likely be a nonempty stack S, since there are no warranties that an arbitrary substring of a sentence is a valid

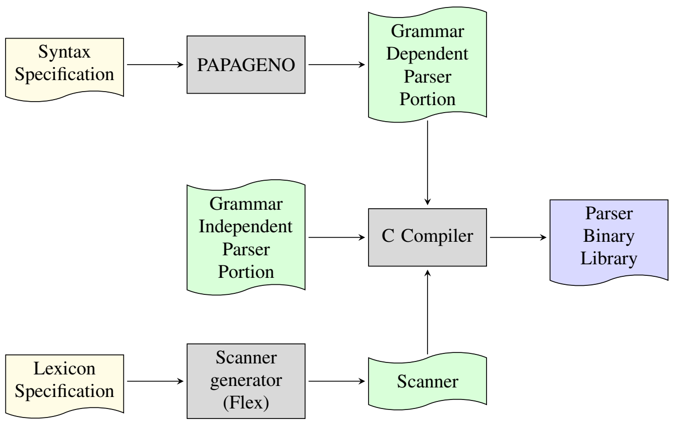


Fig. 2. Typical usage of the PAPAGENO toolchain, starting from a grammar and lexical specification, obtaining the parsing library. Specifications provided by the user are marked in yellow, generated C source code is marked in green.

sentence of the language. Such nonempty stack can be partitioned in two parts: one containing only $\doteq$ and $\gtrdot$ relations (i.e. the lower one), and one containing $\lessdot$ and $\doteq$ (i.e. the upper one).

We thus combine the results of two adjacent parsers employing as the parsing stack of the new parser the upper part of the stack of the left worker, and as input stream the lower part of the right one. As the parsing algorithm is able to shift the nonterminal symbols on the stack, it will be able to handle the input stream even if it is not composed by tokens only.

3.2 The PAPAGENO Parser Generator

PAPAGENO, the parallel parser generator tool, produces a C implementation of the parallel parsing algorithm described before, from the specification of a grammar in operator precedence (Floyd) form. The implementation of the parser is combined with a lexical scanner obtained from the de-facto standard scanner generator Flex, to obtain a fully functional parser library, which can be linked with the main application being developed. We chose to employ serial lexers generated by Flex to enhance the ease of use of the tool, however designing a parallel lexer is a rather easy task, as lexical analysis is inherently local and provides a mean to split the input data properly. Moreover the performance loss is negligible as the lexing process is very fast. The parsing process is started by invoking the `parse` call, which receives two parameters: a reference to the input character stream, and the number of workers onto which the parsing process should be split. As depicted in Figure 2, the typical workflow to employ PAPAGENO is analogous to the one of common parser generators such as Yacc/Bison. The user writes two files: a grammar specification, describing the grammar rules and any semantic actions to be performed jointly with reductions, and a lexical specification, describing the terminals or tokens used by the grammar. PAPAGENO can thus be employed as a drop-in replacement for common parser generators, provided that the user checks the form of the language grammar, and removes the possible precedence conflicts in the rules.

For the sake of clarity, and to ease the study of the implementation and possible modifications, the C parser implementation is split into a grammar-independent part (support for data structures, parsing algorithm implementation) and a grammar dependent part (token and productions representation), residing in different compilation units. The choice of the C language as the target for the implementation was driven by the need to produce highly performing parser implementations while retaining the largest portability. To ease the adoption of PAPAGENO, the syntax of the grammar specification file follows closely the one used by Yacc/Bison, thus requiring little or no effort to port an existing grammar definition. In particular, semantic attributes of the terminals and nonterminals may be conveniently accessed through the same syntax as Bison, and the semantic actions to be triggered upon a reduction are specified in the same way. To guide the user during the process of representing the grammar in Floyd form, PAPAGENO offers diagnostic messages pinpointing any existing precedence conflicts between terminal symbols, and it outputs a printable form of the precedence matrix.

3.3 Performance Tuning Strategies

It would be impossible to achieve the potential advantages of parallel parsing without a careful choice of efficient programming techniques, of which we report here the most significant ones. We have found that memory access represents the bottleneck of our parsing technique, due to the computational lightweight nature of FG parsing, therefore the parsers generated by PAPAGENO exploit various techniques to relieve as much as possible the memory pressure on the target architecture. First, the terminal and nonterminal symbols are represented as integers, taking care to use the most significant bit as a flag to separate terminals or non-. In this way, it is possible to decide whether a list node should be shifted on the stack or not, through a simple check on the first bit of the value, avoiding the use of a lookup table.

In addition to this, since the precedence value can only assume four different values (namely, $\lessdot, \doteq, \gtrdot$ and $\perp$), we use a bit-packed representation of the OPM effectively reducing its size by four times against a straightforward character based representation, which allows to achieve low latency access to the table thanks to the fact that it is small enough to fit in the processor cache memories. To prevent performance losses from fragmented memory allocation, typical of pointer based structures such as the AST, we manage a preallocated userspace memory pool, wrapping the common memory allocation function (`malloc`). This technique both increases the data locality and prevents the workers, implemented as POSIX threads, from being serialized during the calls to the `malloc` function. As far as the size of the memory pool goes, PAPAGENO preallocates half of the estimated size of the AST, through computing the average branching factor from the length of the right hand sides of the grammar rules, and increases the allocated pool size by one fifth of this quantity, if the parser needs more memory.

Another performance tuning technique concerns a smart representation for the rhs of the rules, to ease the checks upon reduction. The rhs are stored in a prefix tree (*trie*), thus allowing the parsing process to find the matching production in linear time w.r.t. the length of the rhs of the productions. In order to further compress the trie, we use the technique described by Germann et al. in [11], which represents the trie as an array, both

$\text{S} \rightarrow \text{OBJECT}$
$\text{OBJECT} \rightarrow \{\} \mid \{\, \text{MEMBERS} \,\}$
$\text{MEMBERS} \rightarrow \text{PAIR} \mid \text{PAIR}\, ,\text{MEMBERS}$
$\text{PAIR} \rightarrow \text{STRING} \,:\, \text{VALUE}$
$\text{VALUE} \rightarrow \text{STRING} \mid \boldsymbol{number} \mid \text{OBJECT} \mid \text{ARRAY} \mid \boldsymbol{bool}$
$\text{STRING} \rightarrow {''}{''} \mid {''}\,\text{CHARS}\,{''}$
$\text{ARRAY} \rightarrow [\,] \mid [\,\text{ELEMENTS}\,]$
$\text{ELEMENTS} \rightarrow \text{VALUE} \mid \text{VALUE}\, ,\text{ELEMENTS}$
$\text{CHARS} \rightarrow \text{CHAR} \mid \text{CHAR CHARS} \quad \| \quad \underline{\text{CHARS} \rightarrow \boldsymbol{char} \mid \boldsymbol{char}\ \text{CHARS}}$
$\text{CHAR} \rightarrow \boldsymbol{char}$

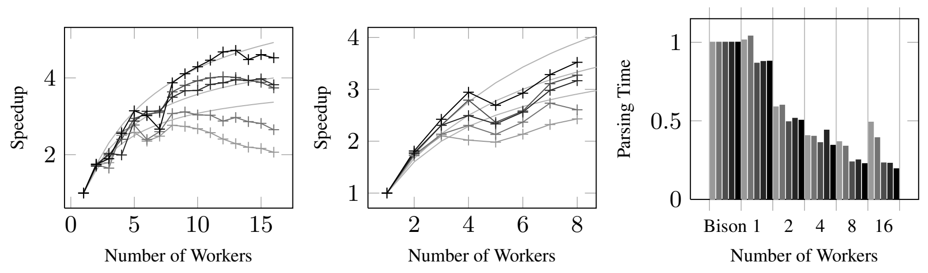
Fig. 3. Official JSON grammar. Uppercase symbols denote nonterminals, while lowercase ones are tokens; the only one modified rule needed to transform the grammar in operator precedence form is underlined.

improving the data locality and reducing the size of the data structure, while retaining the same cost in the lookup operations. This technique involves the representation of the pointers of the trie structure as indexes stored in the same array as the trie values.

4 Benchmark Application: The JSON Language

We chose the *JavaScript Object Notation* (JSON) language as a case study to evaluate the performances of the generated parser. JSON is a data representation language described in the Internet Engineering Task Force document RFC4627, and widely employed in web applications as a less verbose substitute for XML. We also picked JSON as the average size of a JSON file is larger than a compilation unit of a common programming language. The official grammar of JSON, listed in Fig. 3 required only a trivial change to be put into Floyd compliant form, thus confirming the expressive power of this family of languages. The generated parser for the JSON grammar was tested on two different x86-64 Linux hosts to evaluate the achieved speedups: the first host is an Intel Core i7 920, a high end desktop CPU endowed with Simultaneous Multi-Threading (SMT) capabilities, while the second one is a quad-Opteron 8378, (16 physical cores in total, 4 cores per socket), a typical server grade platform. All the hosts were running a Linux 2.6 series kernel and were equipped with enough RAM to contain the whole AST and token list.

As a testbench, we chose real world JSON files of different sizes, in order to evaluate the speedup obtainable. The set of chosen files encompasses the configuration file of AdBlocker, a common browser plugin (80 kB), the Gospel of John (150kB), a statistic data-bank on food consumption provided by the Italian Institute of Statistics (1.6MB), a file containing statistics on n-grams present in English in Google Books (10MB), and the index of all the documents available on the UK Comprehensive Knowledge Archive Network (75MB). Figure 4(a) and 4(b) report the speedup factors over a serial parsing process achieved on the aforementioned testbench by the 16 core and 4-core-SMT platforms respectively while raising the number of workers. The tone of grey in the plot indicates the length of the string, with the lighter greys representing shorter strings.



(a) Speedups w.r.t input string length - 16 Core Platform (b) Speedups w.r.t input string length - 4 Core Platform (c) Comparison with Bison - 16 Core Platform

Fig. 4. Speedups obtained with respect to the string length (Figures (a) and (b))and comparison with Bison generated LALR(1) parsers (Figure (c)). Running times have been normalized taking as time unit the execution time of the Bison generated parser.

Moreover, the theoretical speedups predicted by Amdahl's law for a parallel code portion of 75%-80%-85% are plotted in background as a reference gauge. Already for small string lengths (80kB) the algorithm yields a speedup higher than 2×, but the full advantages of parallel parsing become evident from strings as small as 150kB (where the parallel execution portion reaches 75%) and are fully exploited starting from the 1.6 MB dataset. These results show that our approach is already effective and profitable for text sizes in the range of the average webpage (Google reports in its web metrics report in 2010 an average page size of 320 kB [12]). We also report that, without employing the aforementioned performance tuning techniques to optimize memory accesses and reduce inter-worker serializations (packed structures and memory pooling), the portion of code effectively executed in parallel by the architecture was significantly lower.

Overall, the parallel parsing strategy shows a promising exploitation of the multiple cores available on modern platforms. In particular, this strategy effectively exploits the advantages of simultaneous multi-threading enabled architectures, as raising the number of workers beyond the one of the physical cores of the architecture (i.e. above four in our case) still yields significant speedups. A quantitative measure of the parallel code portion shows that 95% of the instructions are performed in parallel on the four cores thanks to the interleaving of the instructions from two workers on a single core performed by the architecture. This tight instruction interleaving exploits the stalls caused by the memory load and store actions to perform computations from another worker effectively obtaining a parallel code portion close to the theoretical maximum. Finally, Figure 4(c) provides a comparison of the overall parsing times against the ones of a serial LR parser generated by Bison. The depicted data show how our approach behaves consistently better than Bison when employing at least two workers, while showing comparable running times even when employed in serial mode. In particular, the parsing time for the longest string (75MB in size) is effectively cut down from 10.51s to 2.07s, a roughly five-fold improvement.

As there is no recent open literature report on the performances of a general purpose parallel parser generator, we report the work of Lu et al. [8], who built an XML specific parser. The authors report that, exploiting selected features of the language identified via a preparsing phase, it is possible to obtain speedups ranging from 2.35× to 2.55×

on a quad core machine, depending on the target XML structure. To this approach we compare favourably as we achieve higher speedups without the use of a preparsing pass or any specific knowledge about the points where the input token stream is split.

5 Conclusion

We have presented a new parallel parser generator which exploits the properties of operator precedence grammars, also known as Floyd grammars. Such grammars are expressive enough to be used a real world programming language (for instance Algol68 has a fully specified FG). The tool generates automatically a C implementation of the parser given a grammar description provided in Bison compatible syntax, and an additional parameter indicating how many threads are desired. The experimental validation of the effectiveness of the generated parsers, employing the grammar of JSON as a practical test case shows that the code portion running in parallel reaches 85% on common multicore architectures and scales well up to 16 cores.

As a future direction to enhance our tool, we foresee the development of a parallel lexer generator. This will allow a complete parallelization of the lexing-scanning process, thus allowing an easier distribution of the workload among different execution units. Also, the same distinguishing FG property of closure under substring extraction will allow us to couple parallel parsing with incremental techniques.

Acknowledgements. We thank Brad Chen of Google Inc. for practical indications on the benchmarks, and in particular for suggesting to use JSON. Valerio Ponte has participated in the early development of the tool.

References

1. Grune, D., Jacobs, C.J.H.: Parsing techniques a practical guide. Ellis Horwood Limited, Chichester (1990)
2. Crespi Reghizzi, S., Mandrioli, D.: Operator precedence and the visibly push-down property. JCSS, Journ. Computer and System Science 78(6), 1837–1867 (2012)
3. Cohen, J., Kolodner, S.: Estimating the speedup in parallel parsing. IEEE Transactions on Software Engineering 11(1), 114–124 (1985)
4. Sarkar, D., Deo, N.: Estimating the speedup in parallel parsing. IEEE Trans. on Softw. Eng. 16(7), 677 (1990)
5. Mickunas, M.D., Schell, R.M.: Parallel compilation in a multiprocessor environment (extended abstract). In: Proceedings of the 1978 Annual Conference, ACM 1978, pp. 241–246. ACM, New York (1978)
6. Goeman, H.: On parsing and condensing substrings of LR languages in linear time. Theor. Comput. Sci. 267, 61–82 (2001)
7. Bates, J., Lavie, A.: Recognizing substrings of LR(k) languages in linear time. ACM Trans. Program. Lang. Syst. 16, 1051–1077 (1994)
8. Lu, W., Chiu, K., Pan, Y.: A parallel approach to XML parsing. In: GRID, pp. 223–230. IEEE (2006)
9. Pan, Y., Zhang, Y., Chiu, K.: Hybrid Parallelism for XML SAX Parsing. In: IEEE International Conference on Web Services, ICWS 2008, pp. 505–512. IEEE Computer Society (2008)

10. Cole, M.: Parallel programming, list homomorphisms and the maximum segment sum problem. In: Proceedings of ParCo., vol. 93, pp. 211–230 (1993)
11. Germann, U., Joanis, E., Larkin, S.: Tightly packed tries: How to fit large models into memory, and make them load fast, too. In: Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing, pp. 31–39 (2009)
12. Ramachandran, S.: Web metrics: Size and number of resources. Technical report, Google (2010)

TouchRAM: A Multitouch-Enabled Tool for Aspect-Oriented Software Design

Wisam Al Abed, Valentin Bonnet, Matthias Schöttle, Engin Yildirim, Omar Alam, and Jörg Kienzle

School of Computer Science, McGill University, Montreal, QC H3A E09, Canada
`Omar.Alam@mail.mcgill.ca, Joerg.Kienzle@mcgill.ca`

Abstract. This paper presents TouchRAM, a multitouch-enabled tool for agile software design modeling aimed at developing scalable and reusable software design models. The tool gives the designer access to a vast library of reusable design models encoding essential recurring design concerns. It exploits model interfaces and aspect-oriented model weaving techniques as defined by the Reusable Aspect Models (RAM) approach to enable the designer to rapidly apply reusable design concerns within the design model of the software under development. The paper highlights the user interface features of the tool specifically designed for ease of use, reuse and agility (multiple ways of input, tool-assisted reuse, multitouch), gives an overview of the library of reusable design models available to the user, and points out how the current state-of-the-art in model weaving had to be extended to support seamless model reuse.

1 Introduction

Model-Driven Engineering (MDE) [12] is a unified conceptual framework in which the whole software life cycle is seen as a process of *model production*, *refinement,* and *integration*. High-level specification models are refined or combined with other models using *model transformations* to include more and more solution details and to ultimately produce a model that can be executed.

In practice, MDE faces several important challenges that prevent the widespread adoption of modeling as a means to improving the software development process. In the context of this work, the two relevant challenges are *scalability* and *reusability* of models. Models of complex applications tend to grow in size, to a point where even individual views are not readily understood or analyzable anymore. Furthermore, building complex models is very time consuming: models are often created from scratch, as opposed to reusing existing models.

Aspect-orientation modeling (AOM) techniques define special kinds of model transformations called *model weavers* that have been successfully used to separate and compose crosscutting concerns within software models, focussing in particular on the intricacies of concern interactions and conflicts. AOM makes it possible to package models of generic concerns in such a way that they are easy to reuse within other models. Furthermore, by providing weaver support for model hierarchies, complex models can be build by composing existing ones.

This paper presents TouchRAM, a multitouch-enabled tool for agile software design modeling aimed at developing scalable and reusable software design models. The tool gives the designer access to a vast library of reusable design models encoding essential recurring design concerns and provides support to rapidly apply these concerns within the design of the software under development. This is enabled by exploiting model interfaces and aspect-oriented model weaving techniques as defined by the Reusable Aspect Models (RAM) approach [10].

The paper is structured as follows: Section 2 gives an overview of the tool implementation and background on RAM. Section 3 highlights the tool features specifically targeted at agile software design modeling: subsection 3.1 presents how the GUI streamlines model manipulation; subsection 3.2 outlines the library of reusable design concern models available to the user and explains how they can be applied within an application model, and subsection 3.3 explains how the tool supports working with model hierarchies. Section 4 points out how the current state-of-the-art in model weaving had to be extended to support seamless model reuse and model hierarchies, and the last section draws some conclusions.

2 Background

This section presents background information on the tool, i.e., what technologies and frameworks it is built on and other implementation details. In order to make the tool accessible to a wide audience, cross-platform compatibility was one of the major design concerns. We therefore opted to do our development entirely in Java. As a result, all libraries and frameworks we considered to use to build our user interface and model transformation backend had to be implemented in Java as well.

2.1 Architecture

TouchRAM consists of the front end, i.e., the graphical user interface (GUI) and the backend, which contains the RAM meta-model and the RAM model weaver.

The GUI of the TouchRAM tool is realized using the open source Java framework *Multitouch for Java* (MT4j) [3]. MT4j is a framework for creating visual applications in 2D or 3D using OpenGL for software or hardware accelerated graphics rendering. An event stack that allows for different kinds of input events is also provided; the tool ships with support of mouse and keyboard input as well as multitouch input through the TUIO protocol [4]. TouchRAM has been tested for 32 and 64 bit architectures on the Mac OSX, Windows and Linux (Ubuntu) platforms, however, its dependence on Java and TUIO means it should work on any environment where both these are supported.

While the user interface of TouchRAM relies on MT4j, all other components of the tool are decoupled from the GUI based on a Model-View-Controller design. This makes separate evolution of the GUI and the backend possible.

The foundation of the backend is the RAM meta-model that defines the abstract syntax for RAM models created with the tool. The meta-model is defined using the Eclipse Modeling Framework (EMF) [14]. The provided facility allows us to define the structured data model and generate the required Java code that is used by TouchRAM. Furthermore, we are able to serialize our models in XMI (XML Metadata Interchange) format corresponding to that meta-model. Command-based editing provided by EMF.Edit is used in order to offer the user undo/redo functionality. The User Interface is notified of changes to the model through EMFs built-in notification mechanism. The Object Constraint Language (OCL) is used for constraints on the meta-model, specification of derived properties and implementation of operations defined in the meta-model.

The RAM weaver, which is invoked by the GUI on command of the user, is capable of composing multiple models together. Instead of directly implementing the weaving with Java, TouchRAM uses the Kermeta workbench [1]. Kermeta provides an object-oriented model transformation language based on lambda expressions similar to OCL and works with EMF-based meta-models. Furthermore, Kermeta includes support for aspect-orientation. Transformations written in Kermeta are compiled into Scala code, which runs on a standard Java VM.

2.2 Reusable Aspect Models

TouchRAM is based on Reusable Aspect Models (RAM), an aspect-oriented multi-view modeling approach that integrates class diagram, sequence diagram and state diagram AOM techniques [10]. As a result, RAM aspect models can describe the structure and the behavior of a concern under study. Currently, however, TouchRAM only supports structural modeling.

RAM aspect models define an aspect interface that clearly designate the functionality provided by the aspect, as well as its mandatory instantiation parameters [5]. When an aspect model is applied, all mandatory instantiation parameters must be mapped to compatible model elements in the application model. Flexibility is achieved by allowing any model element to optionally be composed or extended. RAM supports the creation of elaborate aspect dependency chains. This makes it possible to model an aspect that provides complex functionality by decomposing it into aspects that provide simpler functionality. At the same time, aspects providing simpler functionality can be reused in several aspects of complex functionality. As a result, scattering and tangling of models can be prevented at all complexity levels.

3 Tool Features Supporting Agile Software Design

Modeling raises the level of abstraction in comparison to source code, and therefore has the potential for enabling fast exploration of software designs. To make this possible, though, a modeling tool must be designed accordingly. In this section we report on three key features of the TouchRAM tool specifically targeted at agile software design: the streamlined model manipulation capabilities offered by the TouchRAM GUI, the reusable design concern library, and navigation through different levels of abstraction.

3.1 Streamlined Model Manipulation

Exploiting Platform Capabilities. The GUI of TouchRAM has been designed to provide intuitive and fast model manipulation capabilities to ensure that the user can focus entirely on the task of modeling. This starts by maximally exploiting the input and output hardware of the platform that the tool is running on. On the input side, TouchRAM supports multitouch and gesture-based input as well as standard mouse and keyboard. The tool was designed without modes, i.e., at any time, the modeler can use gestures or the mouse/keyboard to manipulate the model, depending on what input is most efficient. On the output side, TouchRAM is designed to support heterogeneous screen sizes, mainly by providing high performance support for panning and zooming. The user can change the zoom level at any time by either using the mouse wheel or a two-finger scale gesture. To assure that a model created on one screen can be opened on a different screen without losing the overall overview of the model, TouchRAM scales models according to the current screen dimensions when opening.

Intuitive Editing. General tasks are performed using simple gestures (i.e., tap, double-tap and tap-and-hold) that work with both mouse and touch input. For example, tap-and-hold is used to create or edit model elements. When performed on the background, a class is created. Tapping-and-holding on a class puts the class in edit mode which allows to delete or add attributes or operations. Double-tapping allows to edit an element or one of its properties. When done on the visibility field or return type of an operation or the type of an attribute, a selector menu pops up displaying semantically correct choices for that element.

Certain manipulations can be done very efficiently using multitouch. For instance, the tool can recognize advanced gesture commands, i.e., drawing a rectangle to create a class, performing a zig-zag movement to delete a model element, or drawing a line to create an association. For users that do not have access to multitouch input, TouchRAM offers (less efficient) mouse equivalents for these commands, for instance, double-clicking on one class to put it into edit mode, and then double-clicking another class to create an association between the two or clicking and holding to inherit from that class.

Of course there are also manipulations that are more efficiently done with the keyboard and mouse, e.g., writing text or detailed positioning of model elements. For users that do not have access to a keyboard, TouchRAM displays a popup touch-keyboard whenever a text input is expected.

Some manipulations can also be accomplished in multiple ways. For instance, when adding a new operation to a class and a keyboard is available, the whole signature can be written at once. The given signature is then parsed and checked for conformance to the meta-model. For example, entering "+ String getName()" creates an operation called getName which is public, has no parameters and returns a String. If no keyboard is available, only the model element names need to be provided using the touch-keyboard. The tool displays the potential semantically valid visibility, parameters and return types choices to the modeler, who selects the desired elements with a simple tap.

Tool-Assisted Layout. Currently TouchRAM does not provide support for automated layout of class diagrams. However, the modeler can rearrange classes one by one, or multiple classes simultaneously using multitouch gestures. The relationships between classes, i.e., associations and inheritance, are repositioned automatically by the tool using a sophisticated algorithm. The relationship lines are attached to the class angle depending on the relative position of the related classes, minimizing line crossings.

3.2 Library of Reusable Design Concern Models

The success of modern programming languages such as Java is partially due to the fact that they ship with a significant library of reusable code. The Java Class Library as an example offers a programmer thousands of classes that provide solutions for common implementation concerns, including classes for all common data structures, such as lists, trees, and maps. That way, a programmer does not need to code these classes herself, but simply reuses their behavior by instantiating them and calling the appropriate methods.

The idea of the reusable design concern model library (RDCML) that ships with TouchRAM is similar, but is applied to modeling. Its purpose is to increase modeling productivity by providing models for common design concerns that a modeler can use within an application model with minimal effort when appropriate[1]. Each model in the library is self-contained, i.e., it contains all the structural and behavioral model elements pertaining to a design concern, and is typically relatively small (1 - 6 classes). The *model interface* clearly designates all the classes, associations and operations that are visible, i.e., that can be instantiated / called at runtime to invoke the functionality provided by the model. The current models in the RDCML are organized into the following categories:

– The *design patterns* category contains aspects for the basic structural, behavioral and creational design patterns (e.g., *Singleton*, *Observer*, *Command*, etc.)
– The *utility* category contains aspects that provide basic functionalities like copying *(Copyable aspect)* and naming *(Named aspect)*, as well as data structures involving multiple objects, such as *Map*.
– The *networking* category contains aspects relevant to networking, such as *Serializer*, *SocketCommunication* and *NetworkedCommand*.
– The *workflow* category contains aspects that are useful whenever the application needs to define and execute flexible workflows. For example, the current library supports sequential, conditional, timed, nested and parallel execution of activities.

[1] The RDML is not meant to replace standard class libraries. On the contrary, in order to access functionality provided by standard programming language libraries, TouchRAM currently provides support for importing Java classes into design models using reflection.

– The *transactions* category contains aspects that provide state checkpointing and recovery support, as well as design solutions for isolating concurrently running activities.

Applying a Reusable Design Concern. When elaborating an application model, a modeler typically starts by defining application-specific structure and behavior. When appropriate, she can also choose to apply aspect models from the RDCML to complete her design. In RAM terminology this is called *instantiation*. Aspect models are reusable, because they can be instantiated multiple times in the same application model, or in several distinct application models.

When the modeler applies an aspect from the library, TouchRAM displays an instantiation view, which allows the modeler to establish a mapping from classes and methods in the library model (also called the lower-level aspect) to the application model (also called the higher-level aspect). The instantiation view is divided into two parts: the higher-level aspect is viewed in the top and the lower-level aspect is viewed in the bottom of the view. Tapping on a class in the bottom view highlights the classes that it can be mapped to in the higher-level aspect. Just like for standard model manipulations, the tool assists the modeler in creating semantically correct mappings. This is illustrated in Fig. 1. It depicts a situation where a modeler applies the *Observer* design pattern model to a *StockExchange* application model. She has already mapped the *|Subject* class of the lower-level aspect to the *Stock* class, and the *|Observer* class to the *StockWindow* class, and is now in the process of mapping the *|modify* operation of *|Subject*. Since *|modify* is part of class *|Subject* in the lower-level aspect, and *|Subject* was already mapped to *Stock* in the higher-level aspect, TouchRAM marks only the methods of *Stock* with matching parameters as selectable.

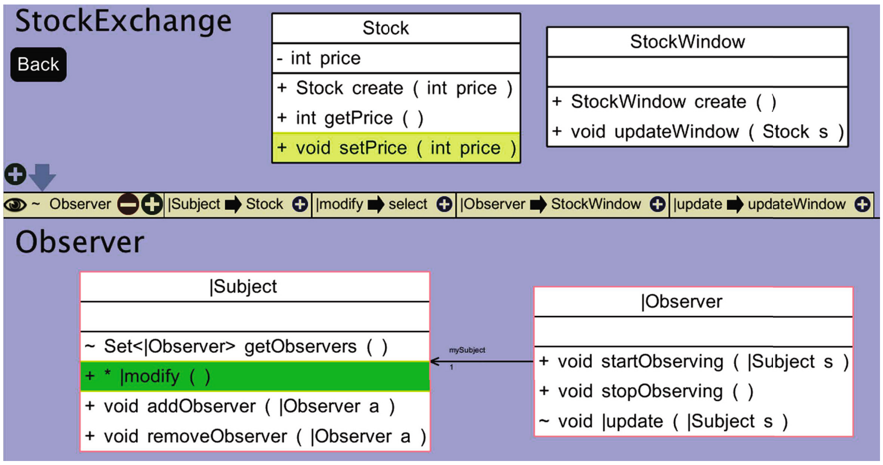

StockExchange
Back
Stock
- int price
+ Stock create (int price)
+ int getPrice ( )
+ void setPrice (int price)
StockWindow
+ StockWindow create ( )
+ void updateWindow (Stock s)
~ Observer
|Subject ➡ Stock
|modify ➡ select
|Observer ➡ StockWindow
|update ➡ updateWindow
Observer
|Subject
~ Set<|Observer> getObservers ( )
+ * |modify ( )
+ void addObserver (|Observer a)
+ void removeObserver (|Observer a)
mySubject
1
|Observer
+ void startObserving (|Subject s)
+ void stopObserving ( )
~ void |update (|Subject s)

Fig. 1. Instantiation View of TouchRAM

The *Observer* example also nicely illustrates why reusable design concern models are more powerful than OO programming language libraries. Class libraries can only encapsulate implementation concerns if the structural and behavioral interface for the concern is contained in a single class. This is not the case for the *Observer* design pattern, since it involves two classes (*Subject* and *Observer*) with distinct behavioral responsibilities (*modify*, *notify*, *update*, etc.).

3.3 Navigating Levels of Abstraction

Instantiations can not only be used to reuse models of the RDCML. A modeler can define her own reusable aspect models, which allows her to decompose a big application model into several smaller inter-dependent models that describe different application-specific design concerns. TouchRAM supports complex model dependency chains, and hence big models can be built by combining many small aspect models that describe the design at different levels of abstraction. For example, the low-level aspect *Observer* shown in Fig. 1 actually depends on an even lower-level aspect called *ZeroToManyAssociation*, which uses the Java implementation class `java.util.Set` to link an instance of a *|Data* class to many instances of the *|Associated* class. The *Observer* aspect uses this low-level design concern to link a *|Subject* instance with many *|Observers*, as shown in the instantiation directives on the bottom line of Fig. 1.

When using TouchRAM, it is not uncommon to create software designs with model hierarchies with many layers of abstraction/models. A modeler can proceed in a top-down manner, starting at high-level application-specific models, and incrementally adding lower-level design details, either self-modeled or from the RDCML. Conversely, the modeler can also start by designing lower-level models first, and then raise their level of abstraction by adding higher-level models that depend on them.

TouchRAM allows the modeler to easily navigate through the model hierarchy. When she opens an aspect model, a list of its instantiations is shown in the bottom of the editing view. She can view the instantiated aspect simply by tapping on the eye icon of an instantiation: a new view opens to display the instantiated aspect. This allows the modeler to focus on each individual design concern at each level of abstraction in isolation.

TouchRAM also allows the modeler to visualize how a higher-level aspect interacts with a lower-level aspect. Tap-and-hold on an instantiation instructs the RAM weaver to combine the lower-level aspect with the higher-level one according to the instantiation mapping to yield a woven model that displays the model elements from both models, i.e., from both levels of abstraction. Using this feature, the modeler can selectively visualize specific lower-level design details, for instance for analysis reasons.

To experiment with different designs, the modeler can exchange design models at a given level of abstraction with other ones providing similar functionality. Typically this simply involves replacing an instantiation in the higher-level model, and then asking the weaver to compose the models again.

Finally, with the “weave all” command, the modeler can instruct the weaver to “flatten all levels of abstraction” and produce a woven model that contains all the specified design details. The details on how the weaver handles model hierarchies are presented in the next section.

4 Hierarchical Model Weaving

The core of the class diagram weaver in TouchRAM is based on the symmetric composition technique proposed by France et al. [11] that was implemented in a tool called *Kompose* [8,2]. In essence, Kompose merges two class diagrams into one by looking at the signature of the model elements in each diagram, and then combining those with matching signatures. After the merge, post-merge directives can be used in order to make changes to the resulting model, if necessary. In order to support reuse, pre-merge directives can be used to prepare a generic model for a specific use, e.g., to change general model element names to names used in the application model so that the signature-based weaving algorithm will merge them.

Kompose does not directly support aspect hierarchies as defined in RAM, and therefore can not be used as such to support incremental top-down or bottom-up modeling as described in subsection 3.3. We had to considerably modify and extend the approach to fit our needs.

4.1 Instantiation Types

To use a design concern model within another design model, the modeler specifies an instantiation mapping between the two models as described in subsection 3.2. Elements that can be mapped are classes, operations, associations, attributes, and parameters. Currently there are two types of instantiations that can be created: *depends* and *extends*. The two types correspond to the two different ways of using TouchRAM to incrementally build models of significant size.

The *depends* instantiation is used when the two models are modeling different levels of abstraction of the software design, as it is the case for top-down or bottom-up development. With *depends*, the modeler is required to provide mappings for all lower-level model elements that she wants to expose at the higher level. The visibility of all unmapped elements is by default switched from *public* to *aspect-private* [5] by the weaver to encapsulate the low-level details.

The *extends* instantiation is used when the designer’s intent is to increment a current design model with additional functionality. Since in this case both design models are at the same level of abstraction, they often refer to the same model elements. Therefore, with *extends*, default mappings are created for all model elements that have the same signature, and all model elements from the lower-level model maintain their visibility properties during the weaving process.

4.2 Weaving Instantiations

In order to allow design exploration across levels of abstractions and increments in a flexible and agile way, our weaver must be capable of weaving any two directly dependent design concern models within a hierarchy together. The resulting model must be one that correctly replaces the two original models within the hierarchy. This is achieved by updating the instantiation directives from the lower level.

For example, in Fig. 2 A depends on B, which in turn depends on C and D. When weaving B into A to yield a new model A+B, our algorithm needs to update the instantiations made in B that originally mapped elements from C and D to elements in B to now map the elements from C and D to the corresponding elements in A. In our example, aspect A mapped *|B->|A* and *Y->X*, but aspect B mapped *|C->|B* and *|D->Y*. After weaving B into A, the updated mappings are *|C->|A* and *|D->X*.

The general rule to update instantiations at weave time is as follows: Given two aspects A and B where A depends on B, for each mapping m1 in A where a left hand side element appears on the right hand side of a mapping m2 in B (text colored in red linked by dotted lines in Fig. 2), create a new mapping in A between the left hand side element of m2 and the right hand side elements of m1 (text colored in blue linked by dashed lines in Fig. 2).

4.3 Weaving Algorithm

The following list summarizes the steps that our weaver executes to weave a reusable design concern model B into model A:

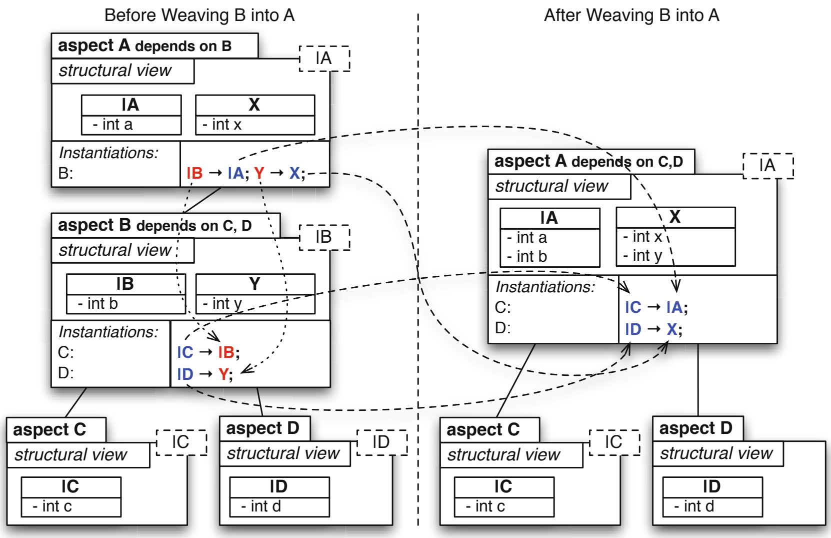


Fig. 2. Updating Instantiation Directives During Weaving

1. Process *extends* instantiations: If the instantiation is of type *extends*, create default mappings for all model elements in B that have corresponding model elements in A. For classes, the names must match. For operations, the signature must match.
2. Check for name clashes: In the woven model, two classes representing different design classes can not have the same name. Therefore, if at this point there exists a class in B with a name that matches the name of a class in A, but there is no mapping defined between the classes and the signatures of the operations in the classes do not match, the weaver terminates with an exception. The modeler is prompted to resolve the name conflict by either renaming the class in A or by defining an explicit mapping.
3. Weave: Merge model elements from B with model elements from A according to the instantiation mapping. Elements in B that are not mapped explicitly are simply copied into A. This yields the woven model A+B.
4. Update instantiations: Update the instantiations in B according to the rule described above and add them to A+B.

5 Conclusion

This paper presented TouchRAM, a multitouch tool for agile software design modeling aimed at developing scalable and reusable software design models. TouchRAM is available at `http://www.cs.mcgill.ca/~joerg/SEL/TouchRAM.html`. The tool highlights are: 1) a streamlined user interface that exploits mouse and touch-based input to enable intuitive and fast model editing, 2) a library of reusable design concern models, and 3) support for model interfaces and elaborate hierarchical model dependencies. With TouchRAM, a modeler can rapidly build complex software designs following either a top-down, bottom-up, or incremental design approach. The tool provides facilities to inspect different levels of abstraction of the design being modeled by navigating the model dependencies, to combine individual models in order to provide insight on how different models interact, as well as generate a complete woven model, if desired.

To the best of our knowledge, TouchRAM is currently the only AOM tool supporting aspect hierarchies. Unfortunately we were not able to verify that claim, since the other existing AOM tools are not readily available for the general public. These include: the Motorola WEAVR [7], which is a proprietary tool for modeling with the SDL notation, MATA [15], an AOM plugin for Rational Architect, and the UML/Theme tool [6].

We are currently working on providing export/import functionality to/from standard UML to integrate TouchRAM with other MDE tools used for software development. We are also planning on adding support for sequence diagrams and state diagrams to specify the behavior of software designs as described in [10]. Finally, we want to integrate some of the ideas of researchers from the HCI community that have worked on touch-based manipulations of diagrams in order to improve the TouchRAM interface. For instance, Frisch et al. [9] present a way for handling complex and large models by introducing off-screen visualization techniques in order to effectively navigate software models. The basic premise of their work is to represent model elements that are clipped from the current viewable area by proxies. Schmidt et al. [13] present several interesting multitouch interaction techniques designed for the exploration of node-link diagrams.

References

1. Kermeta, http://www.kermeta.org
2. Kompose, http://www.kermeta.org/mdk/kompose/
3. MT4j - Multitouch for Java, http://www.mt4j.org
4. Tangible User Interface Objects, http://www.tuio.org
5. Al Abed, W., Kienzle, J.: Information Hiding and Aspect-Oriented Modeling. In: 14th AOM Workshop, Denver, CO, USA, October 4, pp. 1–6 (October 2009)
6. Carton, A., Driver, C., Jackson, A., Clarke, S.: Model-Driven Theme/UML. In: Katz, S., Ossher, H., France, R., Jézéquel, J.-M. (eds.) Transactions on AOSD VI. LNCS, vol. 5560, pp. 238–266. Springer, Heidelberg (2009)
7. Cottenier, T., Berg, A.V.D., Elrad, T.: The motoroal weavr: Model weaving in a large industrial context. In: Industry Track of the 5th International Conference on Aspect-Oriented Software Development (AOSD 2006). ACM, Bonn (2006)
8. Fleurey, F., Baudry, B., France, R., Ghosh, S.: A generic approach for automatic model composition. In: 11th AOM Workshop, Nashville, TN (2007)
9. Frisch, M., Dachselt, R.: Off-screen visualization techniques for class diagrams. In: 5th International Symposium on Software Visualization, pp. 163–172. ACM (2010)
10. Kienzle, J., Al Abed, W., Klein, J.: Aspect-Oriented Multi-View Modeling. In: AOSD 2009, March 1-6, pp. 87–98. ACM Press (2009)
11. Reddy, Y.R., Ghosh, S., France, R.B., Straw, G., Bieman, J.M., McEachen, N., Song, E., Georg, G.: Directives for Composing Aspect-Oriented Design Class Models. In: Rashid, A., Akşit, M. (eds.) Transactions on AOSD I. LNCS, vol. 3880, pp. 75–105. Springer, Heidelberg (2006)
12. Schmidt, D.C.: Model-driven engineering. IEEE Computer 39, 41–47 (2006)
13. Schmidt, S., Nacenta, M., Dachselt, R., Carpendale, S.: A set of multi-touch graph interaction techniques. In: ITS 2010, pp. 113–116. ACM (2010)
14. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley Professional (2009)
15. Whittle, J., Jayaraman, P.: MATA: A Tool for Aspect-Oriented Modeling Based on Graph Transformation. In: Giese, H. (ed.) MODELS 2008. LNCS, vol. 5002, pp. 16–27. Springer, Heidelberg (2008)

A Common Foundational Theory for Bridging Two Levels in Ontology-Driven Conceptual Modeling

Giancarlo Guizzardi and Veruska Zamborlini

Ontology and Conceptual Modeling Research Group (NEMO)
Computer Science Department,
Federal University of Espírito Santo (UFES), Brazil
{gguizzardi,vczamborlini}@inf.ufes.br

Abstract. In recent years, there has been a growing interest in the use of Foundational Ontologies, i.e., ontological theories in the philosophical sense to provide real-world semantics and principled modeling guidelines for conceptual domain modeling languages. In this paper, we demonstrate how a philosophically sound and cognitively-oriented ontological theory of *objects* and *moments* (property-instances) has been used to: (i) (re)design a system of modeling primitives underlying the conceptual domain modeling language OntoUML; (ii) derive supporting technology for mapping these conceptual domain models to less-expressive computationally-oriented codification languages. In particular, we address here a mapping strategy to OWL (Web Ontology Language) which addresses the issue of temporally changing information.

Keywords: Ontological Foundations, Conceptual Domain Modeling, Temporally Changing Information, UFO, OntoUML, OWL.

1 Introduction

In December 2011, the Object Management Group (OMG) released a new Request for Proposal (RFP) entitled SIMF (Semantic Information Modeling Federation) [1]. The SIMF initiative is aimed at developing a *"standard that addresses the federation of information across different representations, levels of abstraction, communities, organizations, viewpoints, and authorities. Federation, in this context, means using independently conceived information sets together for purposes beyond those for which the individual information sets were originally defined"*. Moreover, the proposal should *"define, adopt and/or adapt languages to express the conceptual domain models, logical information models and model bridging relationships needed to achieve this federation"*.

Information Federation is inherently a semantic interoperability problem and underlying this RFP there is the recognition that current modeling technologies fall short in suitably supporting this task of semantic interoperability. At first, at the conceptual domain modeling level, we need a language which is truthful to the subtleties of the subject domains being represented. Moreover, this language should be expressive enough to make explicit the ontological commitment of the different worldviews underlying different models to be federated.

In a seminal paper [2], John Mylopoulos defines conceptual modeling as *"the activity of representing aspects of the physical and social world for the purpose of communication, learning and problem solving among human users"* and states that *"the adequacy of a conceptual modeling notation rests in its ability to promote understanding and problem solving regarding these domains among these human users...not machines"*. In summary, conceptual modeling is about representing in diagrammatic notations, conceptualizations of reality to be shared among human users. For this reason, as defended by a number of authors over the years [3,4], a conceptual modeling notation should have its primitives grounded in the categories of a Foundational Ontology. Moreover, following the aforementioned desiderata, this Foundational Ontology should be one that takes both Cognition and Linguistic Competence seriously into consideration [5,6].

The expressivity in a modeling language needed to make explicit the ontological commitments of a complex domain tends to make this language prohibitive from a computational point of view in tasks such as automated reasoning. Conversely, computationally tractable logical languages tend to lack the expressivity to handle this essential aspect of semantic interoperability. For this reason, as defended in [5,6], we need a two level approach for domain modeling: (i) firstly, we should develop conceptual models as rich as possible to efficiently support the tasks of meaning negotiation and semantic interoperability across "communities, organizations, viewpoints, and authorities"; (ii) once the proper relationship between different information models is establish, we can generate (perhaps several different) implementations in different logical languages addressing different sets of non-functional design requirements.

In this paper, we illustrate a number of the aforementioned aspects. Firstly, we present a fragment of a Cognitive Foundational Ontology which has been employed over the years to analyze, re-design and integrate a number of conceptual modeling languages and reference models (section 2). Secondly, we illustrate how this Foundational Ontology has been used to redesign an Ontologically and Cognitively well-founded Conceptual Domain Modeling Language (CDML) (section 3). Finally, we show that these theory's ontological categories (which define the ontological semantics of this CDML) can be directly employed for creating transformations between models in this language and computationally-oriented representations (section 4). In particular, we address here the issue of devising transformation strategies for representing the important modal (temporal) aspects of this CDML in OWL (Web Ontology Language), given the limitations of the latter language in representing this sort of information. And at last, section 5 of the article presents some final considerations.

2 Ontological Background

In this section, we discuss the Unified Foundational Ontology (UFO). UFO is a reference ontology of endurants based on a number of theories from Formal Ontology, Philosophical Logics, Philosophy of Language, Linguistics and Cognitive Psychology. In the sequel, we restrict ourselves to a fragment of this ontology, depicted in Figure 1. Moreover, due to space limitations and the focus of the paper, we present the ontological categories comprising UFO superficially. For an in depth presentation and corresponding formalization, one should refer to [7].

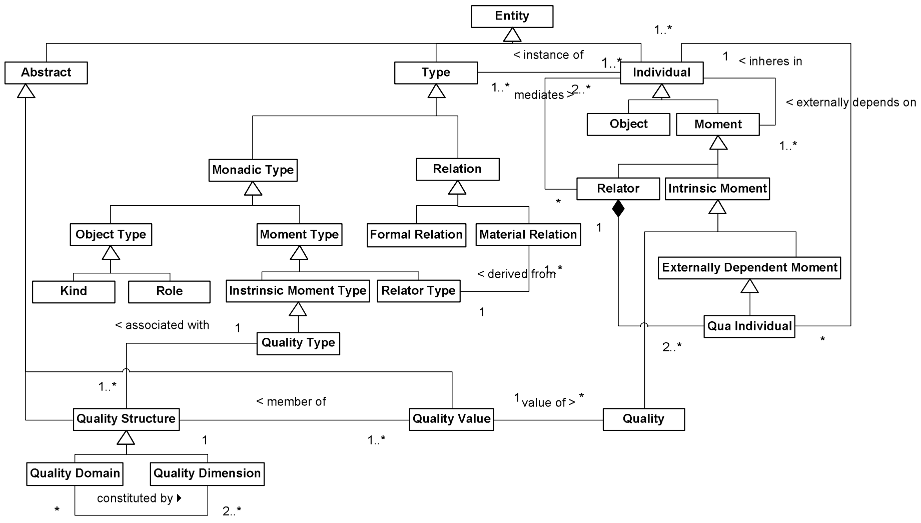


Fig. 1. A Fragment of the Unified Foundational Ontology (UFO)

2.1 Objects and Moments

A fundamental distinction in this ontology is between the categories of *Individual* and *Universal*. Individuals are entities that exist in reality possessing a unique identity. Universals, conversely, are pattern of features which can be realized in a number of different individuals. The core of this ontology exemplifies the so-called *Aristotelian ontological square* or what is termed a "*Four-Category Ontology*" [8] comprising the category pairs *Object-Object Universal*, *Moment-Moment Universal*. From a metaphysical point of view, this choice allows for the construction of a parsimonious ontology, based on the primitive and formally defined notion of *existential dependence*: We have that a particular *x* is *existentially dependent* (*ed*) on another particular *y* iff, as a matter of necessity, y must exist whenever x exists. Existential dependence is a modally constant relation, i.e., if x is dependent on y, this relation holds between these two specific particulars in all possible worlds in which x exists.

The word *Moment* is derived from the german *Momente* in the writings of E. Husserl and it denotes, in general terms, what is sometimes named *trope*, *abstract particular*, *individual accident*, *mode* or *property instance*. Thus, in the scope of this work, the term bears no relation to the notion of time instant in colloquial language. Typical examples of moments are: a color, a connection, an electric charge, a social commitment. An important feature that characterizes all *moments* is that they can only exist in other particulars (in the way in which, for example, electrical charge can exist only in some conductor). To put it more technically, we say that moments are *existentially dependent* on other individuals (named their *bearers*). Existential dependence can also be used to differentiate intrinsic and relational moments: *intrinsic moments* are dependent of one single particular (e.g., color, a headache, a temperature); *relational moments* (or *relators*) depend on a plurality of individuals (e.g., an employment, a medical treatment, a marriage). A special type of existential dependence relation that holds between a moment *x* and the particular *y* of which *x* depends is the relation of *inherence* (*i*). Thus, for a particular *x* to be a moment of another particular *y*, the relation *i(x,y)* must hold between the two. For example, inherence glues your smile to your face, or the charge in a specific conductor to the conductor itself. Here, we admit that moments can inhere in other moments. Examples include the individualized time extension, or the graveness of a particular symptom. The infinite regress in the inherence chain is prevented by the fact that there are individuals that cannot inhere in other individuals, namely, *Objects*.

Examples of *objects* include ordinary entities of everyday experience such as an individual person, a dog, a house, a hammer, a car, Alan Turing and The Rolling Stones but also the so-called *Fiat Objects* such as the North-Sea and its proper-parts and a non-smoking area of a restaurant. In contrast with moments, objects do not inhere in anything and, as a consequence, they enjoy a higher degree of independence. To state this precisely we say that: an object *x* is *independent* of all other objects which are disjoint from *x*, i.e., that do not share a common part with *x*. This definition excludes the dependence between an object and its *essential* and *inseparable parts* [7], and the obvious dependence between an object and its essential moments.

To complete the Aristotelian Square, depicted in Figure 2, we consider here the categories of *object universal* and *moment universal*. We use the term universal here in a broader sense without making any *a priori* commitment to a specific theory of universals. A universal thus can be considered here simply as something (i) which can be predicated of other entities and (ii) that can potentially be represented in language by *predicative terms*. We also use the relation of *instantiation (or classification)* between individuals and universals. Object universals classify objects and moment universals classify moments. Examples of the former include Apple, Planet and Person. Examples of the latter include Color, Electric Charge and Headache. Finally, we define the relation of *characterization* between moment universals and the universals instantiated by the individuals that exemplify them: a moment universal *M* characterizes a universal *U* iff every instance of *U* bears and instance of *M*.

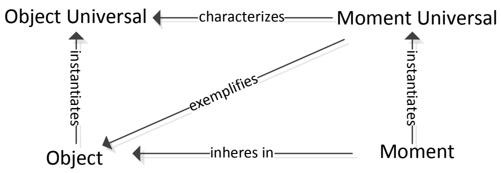


Fig. 2. The ontological Square

2.2 Object Universals

Within the category of object universals, we make a fundamental distinction based on the formal notions of *rigidity* and *anti-rigidity*: A universal *U* is rigid if for every instance *x* of *U*, *x* is necessarily (in the modal sense) an instance of *U*. In other words, if *x* instantiates *U* in a given world *w*, then *x* must instantiate *U* in every possible world *w'*. In contrast, a universal *U* is anti-rigid if for every instance *x* of *U*, *x* is *possibly* (in the modal sense) not an instance of *U*. In other words, if *x* instantiates *U* in a given world *w*, then there must be a possible world *w'* in which *x* does not instantiate *U*. An object universal which is rigid is named here a ***Kind***. In contrast, an anti-rigid object universal is termed here a ***Phased-Sortal*** [7]. The prototypical example highlighting the modal distinction between these two categories is the difference between the Kind Person and the Phase-Sortals Student and Adolescent instantiated by the individual John in a given circumstance. Whilst John can cease to be a Student and Adolescent (and there were circumstances in which John was not one), he cannot cease to be a Person. In other words, while the instantiation of the phase-sortals Student and Adolescent has no impact on the identity of a particular, if an individual ceases to instantiate the universal Person, then she ceases to exist as the same individual.

In the example above, John can move in and out of the Student universal, while being the same individual, i.e. without losing his identity. This is because the principle of identity that applies to instances of Student and, in particular, that can be applied to John, is the one which is supplied by the Kind Person of which the Phase-Sortal Student is a subtype. This is always the case with Phased-Sortals, i.e., for every Phased-Sortal *PS*, there is a unique ultimate Kind *K*, such that: (i) *PS* is a specialization of *K*; (ii) *K* supplies the unique principle of identity obeyed by the instances of *PS*. If *PS* is a Phased-Sortal and *K* is the Kind specialized by *PS*, there is a *specialization condition* φ such that *x* is an instance of *PS* iff *x* is an instance of *K* that satisfies condition φ.

A particular type of Phased-Sortal emphasized in this article is what is named in the literature a ***Role***. A role *Rl* is an anti-rigid object type whose specialization condition φ is an extrinsic (relational) one. For example, one might say that if John is a Student then John is a Person who is enrolled in some educational institution, if Peter is a Customer then Peter is a Person who buys a Product *x* from a Supplier *y*, or if Mary is a Patient than she is a Person who is treated in a certain medical unit. In other words, an entity plays a role in a certain context, demarcated by its relation with other entities. This meta-property of Roles is named *Relational Dependence* and can be formally characterized as follows: A universal *T* is relationally dependent on another universal *P* via relation *R* iff for every instance *x* of *T* there is an instance *y* of *P* such that *x* and *y* are related via *R* [7].

2.3 Qualities and Quality Structures

An attempt to model the relation between intrinsic moments and their representation in human cognitive structures is presented in the theory of *conceptual spaces* introduced in [9]. The theory is based on the notion of *quality dimension*. The idea is that for several perceivable or conceivable quality universals there are associated quality dimensions in human cognition. For example, height and mass are associated with one-dimensional structures with a zero point isomorphic to the half-line of nonnegative numbers. Other properties such as color and taste are represented by multi-dimensional structures. Moreover, the author distinguishes between *integral* and *separable* quality dimensions: *"certain quality dimensions are integral in the sense that one cannot assign an object a value on one dimension without giving it a value on the other. For example, an object cannot be given a hue without giving it a brightness value (…) Dimensions that are not integral are said to be separable, as for example the size and hue dimensions."* He then defines a *quality domain* as "a set of integral dimensions that are separable from all other dimensions" [9]. Furthermore, he defends that the notion of conceptual space should be understood literally, i.e., quality domains are endowed with certain geometrical structures (topological or ordering structures) that constrain the relations between its constituting dimensions. Finally, the perception or conception of an intrinsic aspect can be represented as a point in a quality domain. This point is named here a *quality value*.

Once more, an example of a quality domain is the set of integral dimensions related to color perception. A color quality *c* of an apple *a* takes its value in a three-dimensional color domain constituted of the dimensions hue, saturation and brightness. The geometric structure of this space (the *color spindle* [9]) constrains the relation between some of these dimensions. In particular, saturation and brightness are not totally independent, since the possible variation of saturation decreases as brightness approaches the extreme points of black and white, i.e., for almost black or almost white, there can be very little variation in saturation. A similar constraint could be postulated for the relation between saturation and hue. When saturation is very low, all hues become similarly approximate to grey.

We adopt in this work the term *quality structures* to refer to quality dimensions and quality domains, and we define the formal relation of *association* between a quality structure and an intrinsic aspect universal. Additionally, we use the terms *quality universals* for those intrinsic moment universals that are directly *associated with* a quality structure, and the term *quality* for an aspect classified under a quality universal. Furthermore, we define the relation of *valueOf* connecting a quality to its quality value in a given quality structure. Finally, we also have that quality structures are always associated with a unique quality universal, i.e., a quality structure associated with the universal Weight cannot be associated with the universal Color. This is not to say, however, that different quality structures cannot be associated with the same quality universal. For instance, with the quality universal color, we can have both the HSB (Hue-Saturation-Brightness) structure and the RGB (Red-Green-Blue) structure. In Figure 3 below, we illustrate an entity, its intrinsic color quality and the value of this quality mapped to into two different quality structures, hence, producing two different (albeit comparable) quality values.

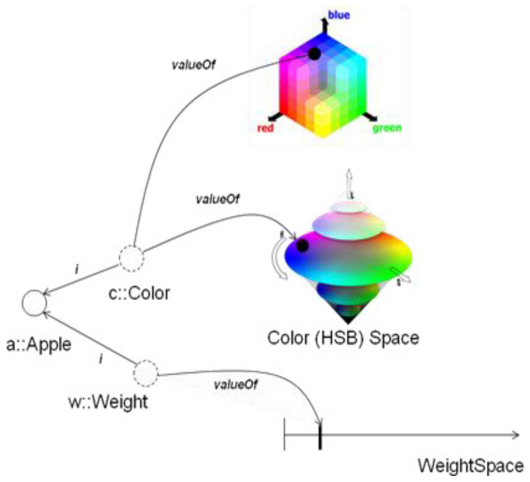


Fig. 3. An object, some of its inhering qualities and the associated quality structures

The view of qualities defended here assumes that change is substitution (as opposed to variation) of moments, i.e. "the color of *x* turned from red to brown" is represented by a red-quality of *x* that temporally precedes a brown-quality of *x*. As a consequence, we have that although a quality q can have different quality values in different quality spaces, their values in each of these structures cannot be changed. Taking this view into consideration, we elaborate further in two orthogonal partitions capturing specific characteristics of qualities which are related to aspects of temporal change of their bearers.

In the first partition, we distinguish between **necessary** (mandatory) versus **contingent** (optional) qualities; in the second, we distinguish between **immutable** versus **mutable** ones. The former distinction refers to the need for an entity to bear that property, regardless of its value. For instance, suppose that in a given conceptualization of (legal) Person, both *name* and *age* are mandatory characteristics of people (every person has a name and an age) whilst *ssn* (*social security number*) and *nickname* (*alias*) are, in contrast, optional characteristics of people. Now, notice that the relation between a person and age is a relation of *generic dependence*, i.e., the same person can bear different age qualities in different situations as long as they are all instances of the quality universal *age*. This brings us to the second of these partitions: a quality *q* is immutable to a bearer *x* of type *T* iff *x* must bear that very same quality in all possible situations in which *x* instantiates *T*. In this case, the relation between *x* and *q* is a relation of *specific dependence* (as opposed to a generic one). Again, let us suppose that, in a given conceptualization, (legal) persons cannot change their proper names. In this situation, a name would not only be a necessary but also an immutable characteristic of people. Suppose now that, in this conceptualization, that although *ssn* is an optional characteristic of people, once an *ssn* is assigned to a person, it cannot be changed. In this case, *ssn* would be an immutable and contingent quality. Finally, in this conceptualization, we assume that nicknames are both optional to people and, once assigned, can always be changed. In this case, nickname would be an example of a contingent and mutable quality.

2.4 Relators, Relations, Roles and Qua Individuals

Following the philosophical literature, we recognize here two broad categories of relations, namely, *material* and *formal* relations [7]. Formal relations hold between two or more entities directly, without any further intervening individual. Examples include the relations of *existential dependence (ed), subtype*, *instantiation*, *parthood, inherence (i)*, among many others not discussed here [7]. Domain relations such as *working at*, *being enrolled at*, and *being the husband of* are of a completely different nature. These relations, exemplifying the category of *Material relations*, have material structure of their own. Whilst a formal relation such as the one between Paul and his headache x holds directly and as soon as Paul and x exist, for a material relation of *being treated in* between Paul and the medical unit MU_1 to exist, another entity must exist which *mediates* Paul and MU_1. These entities are termed *relators*.

Relators are individuals with the power of connecting entities. For example, a medical treatment connects a patient with a medical unit; an enrollment connects a student with an educational institution; a covalent bond connects two atoms. The notion of relator is supported by several works in the philosophical literature [7] and, they play an important role in answering questions of the sort: what does it mean to say that John is married to Mary? Why is it true to say that Bill works for Company *X* but not for Company *Y*? Again, relators are special types of moments which, therefore, are existentially dependent entities. The relation of *mediation* (symbolized *m*) between a relator *r* and the entities *r* connects is a sort of (non-exclusive) inherence and, hence, a special type of existential dependence relation. It is formally required that a relator mediates at least two distinct individuals [7].

An important notion for the characterization of relators (and, hence, for the characterization of material relations) is the notion of *foundation*. Foundation can be seen as a type of *historical dependence* [7,10], in the way that, for instance, an instance of *being kissed* is founded on an individual *kiss,* or an instance of *being punched by* is founded on an individual *punch*, an instance of *being connected to* between airports is founded on a particular flight connection. Suppose that John *is married to* Mary. In this case, we can assume that there is an individual relator m_1 of type *marriage* that mediates John and Mary. The foundation of this relator can be, for instance, a wedding event or the signing of a social contract between the involved parties. In other words, for instance, a certain event e_1 in which John and Mary participate can create an individual marriage m_1 which existentially depends on John and Mary and which mediates them. The event e_1 in this case is the foundation of relator m_1.

Now, let us elaborate on the nature of the relator m_1. There are many intrinsic moments that John acquires by virtue of being married to Mary. For example, imagine all the legal responsibilities that John has in the context of this relation. These newly acquired properties are intrinsic moments of John which, therefore, are existentially dependent on him. However, these moments also depend on the existence of Mary. We name this type of moment *externally dependent moments*, i.e., externally dependent moments are intrinsic moments that inhere in a single individual but are existentially dependent on (possibly multiple) other individuals. The individual which is the aggregation of all externally dependent moments that John acquires by virtue of being married to Mary is named a *qua individual* (in this case, John-qua-husband-of-Mary). A qua individual is, thus, defined as an individual composed of all externally dependent moments that inhere in the same individual and share the same foundation. In the same manner, by virtue of being married to John, Mary bears an individual Mary-qua-wife-of-John.

The notion of qua individuals is the ontological counterpart of what has been named *role instance* in the literature [11] and represent the properties that characterize a particular mode of participation of an individual in a relation. Now, the entity which is the sum of all qua individuals that share the same foundation is a relator. In this example, the relator m_1 which is the aggregation of all properties that John and Mary acquire by virtue of being married to each other is an instance of the relational property *marriage*. The relation between the two qua individuals and the relator m_1 is an example of formal relation of parthood [7].

The relator m_1 in this case is said to be the *truthmaker* of propositions such as "John is married to Mary", "Mary is married to John", "John is the husband of Mary", and "Mary is the wife of John". In other words, material relations such as *being married to*, *being legally bound to*, *being the husband of* can be said to hold for the individuals John and Mary because and only because there is an individual relator marriage m_1 mediating the two. Thus, as demonstrated in [7,10], material relations are purely linguistic/logical constructions which are founded on and can be completely derived from the existence of relators. In fact, in [7], we have defined a formal relation of derivation (symbolized as *der*) between a relator type (e.g., Marriage) and each material relation which is derived from it.

Finally, there is an intimate connection between qua individuals and *role types*: let T be a natural type (kind) instantiated by an individual *x*, and let *R* be a role type specializing *T*. We have that there is a qua individual type *Q* such that *x* instantiates *R* iff *x* bears an instance of *Q*. Alternatively, we have that for every role type *R* there is a relator type *RR* such that *x* instantiates *R* iff *x* is mediated by an instance of *RR*. Note that this conforms to the formal property of roles as *relationally dependent* types.

The summary of the discussion promoted in this section is illustrated in Figures 4 to 6. Figure 4, illustrates the inherence relation between John and his externally dependent moments which are existentially dependent on Mary (as well as analogous relations in the converse direction). In figure 5, John instantiates the role type Husband (which is a specialization of the kind (Male) Person) iff there is a qua individual John-qua-husband-of-Mary which inheres in John. Moreover, this figure illustrates that the qua individuals John-qua-husband-of-mary and Mary-qua-wife-of-John are mutually existentially dependent. In other words, John cannot be the Husband of Mary without Mary being the wife of John. Finally, Figure 6 shows that the material relation *married to* is derived from the relator type Marriage and, thus, tuples such as <John,Mary> and <John,Mary> are instances of this relation iff there is an instance of Marriage that mediates the elements of the tuple.

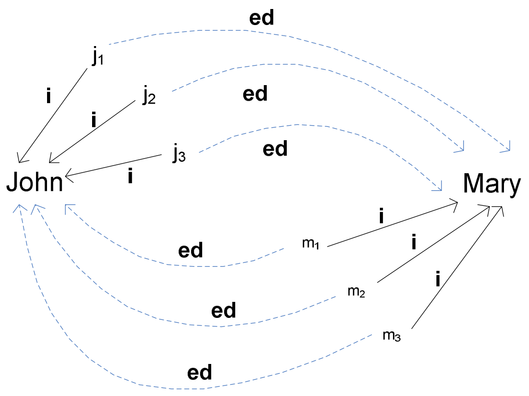


Fig. 4. Objects and their inhering externally dependent moments: in this example, the object bears a number of moments (j_1,j_2,j_3), which inhere (i) in John but which are also existentially dependent (*ed*) on Mary. Mutatis Mutandis, the model depicts a number of moments of Mary (m_1,m_2,m_3), which inhere in Mary but which are also existentially dependent on John.

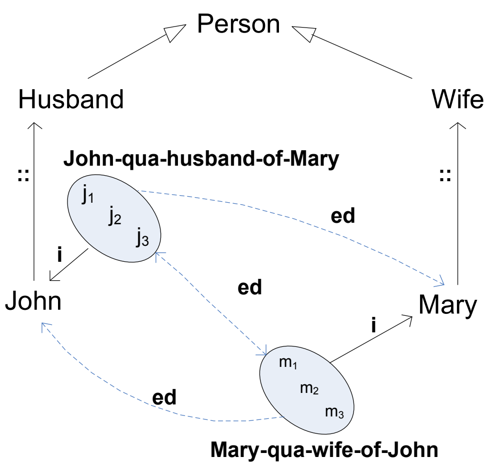


Fig. 5. Objects, their instantiating roles and their inhering qua individuals: in this example, John and Mary instantiate (::) the roles Husband and Wife, respectively, in virtue of the qua individuals that inhere (*i*) in them. These roles are specializations of the type Person ($\longrightarrow$). Moreover, *John-qua-husband-of-Mary* (which is an aggregations of the moments j_1, j_2 and j_3) is mutually existentially dependent (*ed*) on *Mary-qua-wife-of-John* (an aggregation of moments m_1, m_2 and m_3).

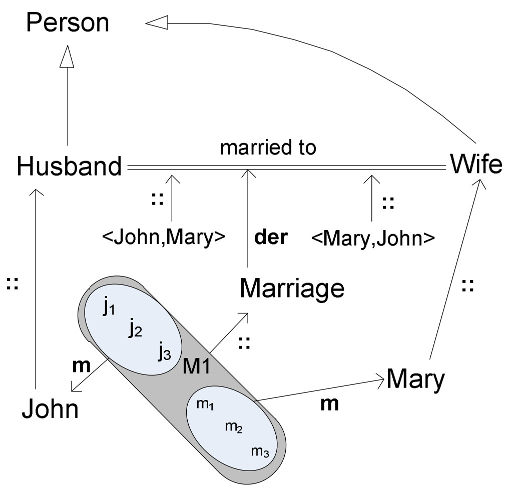


Fig. 6. Material Relations are founded on relators that mediate their relata: in this example, the marriage relator M_1 between John and Mary mediates (*m*) these two entities by virtue of being existentially dependent on both of them. This relator is an aggregation of the qua individuals *John-qua-husband-of-Mary* and *Mary-qua-wife-of-John* (represented by the two ellipses). Moreover, M_1 is the foundation for the tuples <John,Mary> and <Mary,John>, which instantiate (::) the material relation *married to*, which, in turn, is derived (*der*) from the relator universal Marriage which M_1 instantiates.

3 Using a Foundational Ontology to Design a Well-Founded Conceptual Modeling Language

In this section, we present a Conceptual Domain Modeling language termed OntoUML [7]. OntoUML is an example of a conceptual modeling language whose metamodel has been designed to comply with the ontological distinctions and axiomatization of the UFO foundational ontology. This language and its foundations are currently being considered as candidates to contribute to a response to the SIMF Request for Proposal [1].

The OntoUML metamodel contains: (i) elements that represent ontological distinctions prescribed by an underlying foundational ontology; (ii) constrains that govern the possible relations that can be established between these elements. These constraints, which are derived from the axiomatization of the ontological theory, restrict the ways in which the modeling primitives can be related. The goal is to have a metamodel such that all grammatically correct specifications according to this metamodel have logical models that represented *intended state of affairs* of the underlying conceptualization [5].

For instance, the language has modeling primitives to explicitly represent the notions of *kinds*, *subkind* and *roles* as well as the notions *quality* and *relator* previously discussed. Kinds and subkinds are represented by the corresponding stereotypes «kind» and «subkind». In an analogous manner, roles are represented by the stereotype «role». In the axiomatization of the UFO ontology we have that anti-rigid types cannot be a supertype of rigid one [7]. So, as an example of formal constraint in this language, we have that classes stereotyped as «kind» or «subkind» cannot appear in an OntoUML model as a subtype of class stereotyped as «role».

As discussed at length in [12], quality universals are typically not represented in a conceptual model explicitly but via *attribute functions* that map each of their instances to points (quality values) in a quality structure. Accordingly, the *datatype* associated with an attribute A of class C is the representation of the quality structure that is the co-domain of the attribute function represented by A. In other words, a quality structure is the ontological interpretation of the (Onto)UML datatype construct. Moreover, we have that a multidimensional quality structure (quality domain) is the ontological interpretation of the so-called *structured datatypes*. Quality domains are composed of multiple integral dimensions. This means that the value of one dimension cannot be represented without representing the values of others. The fields of a datatype representing a quality domain QD represent each of its integral quality dimensions. Alternatively, we can say that each field of a datatype should always be interpreted as representing one of the integral dimensions of the QD represented by the datatype. The constructor method of the datatype representing a quality domain must reinforce that its tuples always have values for all the integral dimensions. Finally, an algebra (as a set of formal constraints) can be defined for a datatype so that the relations constraining and informing the geometry of represented quality dimensions are also suitably characterized.

There are, nonetheless, two situations in which one might want to represent quality universals explicitly. The first of these is when we want to represent that a quality might be projected to different quality spaces (i.e., the underlying quality universal is associated with alternative quality structures). This idea is represented in figure 7. In this case, the color quality aggregates the different values (in different quality spaces) that can be associated with that object (the apple, in this case). Notice that, in these situations, it is as if the color quality is representing a certain 'aspectual slice' of the Apple, or the *Apple-qua-colored object.*

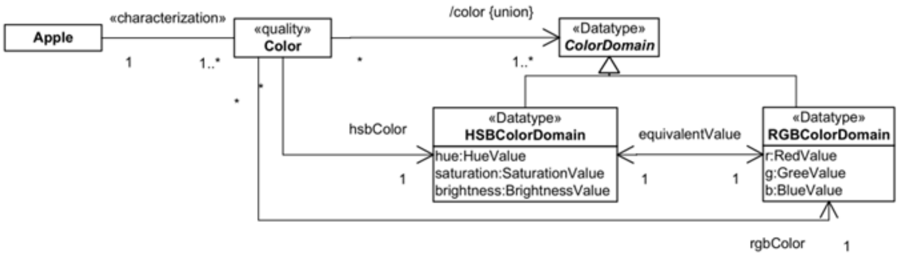


Fig. 7. Representing quality types which can be associated to multiple quality structures [12]

A second situation in which one might want to represent qualities explicitly is when modeling a temporal perspective on qualities. This is illustrated in Figure 8 below. In that model, we have different color qualities (with different associated quality values) inhering in a given apple in different situations.

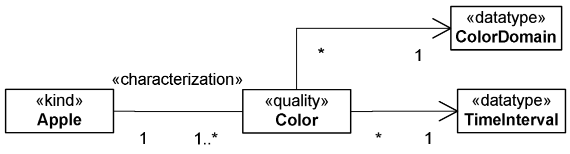


Fig. 8. Temporal change in properties as quality replacement

As illustrated in Figures 7 and 8, in this language, one can employ the stereotype «quality» to explicitly represented quality universals and a stereotyped relation of «characterization» to represent its ontological counterpart. As discussed in section 2, the *characterization* relation between an intrinsic moment universal and the universal it characterizes is mapped at the instance level onto an *inherence* relation between the corresponding individual moments and their bearers. That means that every instance *m* of a class *M* stereotyped as «quality» is existentially dependent of an individual *c*, which is an instance of the class *C* related to *M* via the «characterization» relation. Inherence has the following characteristics: (a) it is a sort of existential dependence relation; (b) it is a binary formal relation; (c) it is a functional relation. These three characteristics impose the following metamodel constraints on the «characterization» construct: by (a) and (c), the association end connected to the characterized universal must have the cardinality constraints of one and exactly one; by (a), the association end connected to the characterized universal must have the meta-attribute (isReadOnly = true); «characterization» associations are always binary associations.

Regarding mutability/immutability and necessity/contingency of qualities, we use the following representation strategy. The necessity of a given quality (or, consequently, of a given quality value) is represented by a minimum cardinality ≥ 1 in the association end connected to the «quality» class (or the association end connected to the associated datatype). Alternatively, a contingent quality is represented by a minimum cardinality = 0 in the referred association end. The immutability of a quality (or the corresponding quality value) is represented by using the tagged value *readOnly* applied to the referred association end (i.e., by making its meta-attribute *isReadOnly* = *true*). Finally, the absence of this tagged value in a given association end indicates that a mutable quality (quality value) is being represented.

Finally, in the language, the stereotype «relator» is used to represent the ontological category of relator universals. As discussed in section 2, a relator particular is the actual instantiation of the corresponding relational property. Material relations stand merely for the facts derived from the relator particular and its mediating entities. In other words, relations are logico-linguistic constructions which supervene on relators. Therefore, as argue at length in [7,10], the representation of the relators of material relations must have primacy over the representation of the material relations themselves. In this paper, we simply omit the representation of material relations.

In the sequel, we provide a final example of formal constraints incorporated in the OntoUML metamodel which is derived from its underlying ontological foundations. Relators are existentially dependent entities. Thus, as much as a characterization relation, mediation is also a directed, binary, existential dependence relation. As consequence, we have that a relation stereotyped as «mediation» in OntoUML must obey the following constraints: (i) the association end connected to the mediated universals must have the cardinality constraints of at least one; (ii) the association end connected to the mediated universals must have the meta-attribute (isReadOnly = true); (iii) «mediation» associations are always binary associations. Moreover, since a relator is dependent (mediates) on at least two numerically distinct entities, we have the following additional constraint (iv) Let *R* be a class representing a relator universal and let $\{C_1 \ldots C_n\}$ be a set of classes mediated by *R* (related to *R* via a *mediation* relation). Finally, let $lower_{Ci}$ be the value of the minimum cardinality constraint of the association end connected to C_i in the mediation relation. Then, $(\sum_{i=1}^{n} lower_{Ci}) \geq 2$.

3.1 Discussion

As shown in [7], the distinction among rigid and anti-rigid object types incorporated in the OntoUML language provides for a semantically precise and ontologically well-founded semantics for some of the much discussed but still *ad hoc* distinctions among conceptual modeling constructs. Since its first proposal in this line of work [13], this distinction has had an impact in conceptual model validation [14], in the discovery of important ontological design patterns [13], as well as in the formal and ontological semantics of derived types in conceptual modeling [15]. Moreover, it has influenced the evolution of other conceptual modeling languages, such as ORM 2.0 [16]. Finally, as argued in [7,17], this distinction has a direct impact even in the choice of different design alternatives in different implementation environments.

Analogously, the explicit representation of intrinsic moments and quality structures in the language allows for providing an ontological semantics and clear modeling guidelines for attributes, datatypes, weak entities and domain formal relations [10,12]. Moreover, the model presented in Figure 7 illustrates a design pattern for *modeling properties associated to alternative quality structures*. Since its first proposal in [12], this pattern has been applied in several domains (e.g., [18]).

Finally, the strategy for the representation of material relational properties discussed here has been applied in a series of publications to address a number of important and recurrent conceptual modeling problems. For instance, in [10], it was used to address the problem of the collapse of cardinality constraints in material relations; in [19], it has used as an integral part in the development of a solution the so-called problem of transitivity of parthood relations; in [20], in an industrial case study of ontology reverse engineering, the systematic identification of the material content of relations (i.e., relators) was reported as a fruitful technique for knowledge elicitation when interacting with domain experts; in [21], the ontological theory of relations underlying this approach has been used to disambiguate the semantics of two fundamental modeling constructs in Goal-Oriented Requirements Engineering; finally, in [22], the same theory has been employed to provide ontological semantics and clear modeling guidelines for disambiguating the constructs of association specialization, association subsetting and association redefinition in UML 2.0.

Because the distinctions and constraints comprising this language are explicitly and declaratively defined in the language metamodel, they can be directly implemented using metamodeling architectures such as the OMG's MOF (Meta Object Facility). Following this strategy, [7] reports on an implementation of an OntoUML graphical editor by employing a number of basic Eclipse-based frameworks such as the ECore (for metamodeling purposes) and MDT (for the purpose of having automatic verification of OCL constraints). An interesting aspect of this strategy is that, once the ontological constraints have been incorporated in the metamodel, they give rise to syntactical constraints. These constraints in the language metamodel, thus, limit the set of grammatically correct models that can be produced using the language to those whose instances only represent consistent state of affairs according to the underlying ontology.

4 From an Ontology-Driven Conceptual Domain Model to a Computationally-Driven Specification in OWL

4.1 Temporally Changing Information in Conceptual Domain Models

The model of Figure 9 below (termed as *running example* in the remainder of this section) illustrates some important aspects related to change that should be highlighted in the discussion that follows. This model represents a situation in which a person, who can be a man or a woman, is identified by his/her name. Moreover, he/she can have a social security number (ssn) that cannot change. He/she has an age that change annually, and can also be referred by one or more nicknames that may change along his/her life. Finally, a man can get married to only one woman per time (and vice-versa), thus, becoming husband and wife, respectively.

We distinguish here three sources of changes: attributes, relations and type instantiation. Regarding intrinsic properties, we can classify them under the dimensions of **necessary** (mandatory) versus **contingent** (optional), and **mutable** versus **immutable** as discussed in section 2. Furthermore, the generic dependence between the Kind Person and Quality Universal age is also present in the relation *marriedTo* between Husband and Wife (and vice-versa). In other words, both association ends of the relation *marriedTo*, albeit mandatory for the associated types (Husband and Wife), are mutable. Finally, we have a third source of change in this model, related to the anti-rigidity of the role universals Husband and Wife. As previously discussed, a particular man instantiates the Role Husband contingently and when mediated by a particular marriage relator. *Mutatis Mutandis*, the same can be said for the role Wife.

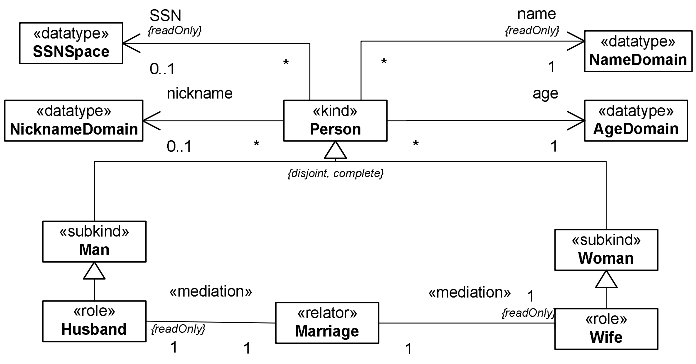


Fig. 9. An OntoUML Conceptual Domain Model with sources of temporal change

4.2 OWL

The OWL (Web Ontology Language) is a well known formal language for representing ontologies on the Semantic Web. In this work, we are particularly interested in its DL based variants, which we refer simply as OWL in the remainder of this text. DL consists of a family of subsets of classical first order logics that is designed focusing on decidable reasoning. Using DL-based languages, one is able to represent static scenarios with immutable truth-values such that the information about the domain can be completed but cannot be really changed. In particular, the instantiation of a class or property cannot be retracted, except through external intervention. For example, once a model represents that *John being 28 years old* instantiates the class *Husband*, this information cannot be changed.

DL has two important characteristics to be taken into account here, namely, open world assumption (OWA) and monotonicity. The former entails that what is stated in the model is true but not necessarily the entire truth about a domain, i.e., new information can always be discovered about the represented entities. However, a monotonic logical system is such that the addition of new information/premises must not interfere in the information that has been previously derived. Consequently, what is true in one situation must remain true regardless of any addition of information to the model.

In an OWL model, we can codify the distinction between mandatory versus optional properties (represented by cardinality constraints). However, we cannot represent the distinction between immutable versus mutable, neither the one between rigid versus anti-rigid types. Due to the aforementioned monotonicity of the language, all stated relations, attribute assignments and classification assignments become immutable. In order to circumvent these limitations, a number of authors have been investigating different strategies for representing temporally changing information in OWL [24, 25].

Most of these approaches employ a strategy which consists of interpreting all enduring entities in a domain model (e.g., objects, qualities, relators) as events (processes). This view is grounded in a philosophical stance named *Perdurantism* [26,27]. In a perdurantistic view, a domain individual is seen as a 4D (four-dimensional) "space-time worm" whose temporal parts are slices (snapshots) of the worm. As argued in [6], although such a view can be accurate in representing the current state of knowledge in natural sciences, the distinction between enduring and perduring entities is a fundamental cognitive distinction, present both in human cognition and language. For this reason, as argued in [5], we advocate that such a distinction should be explicitly considered both in conceptual modeling languages as well as in their underlying foundational ontologies. Moreover, besides the philosophical controversy associated with perdurantism, there are a number of issues triggered by such 4D-driven approaches which can become prohibitive for certain design scenarios. Some of these issues are discussed in the next section and are addressed by an alternative approach considered in this article, namely, a reification-driven approach.

Property-reification is definitely not a new idea. In fact, it is a well-know solution for representing temporal information in knowledge representation going back at least to the eighties. Despite this, and despite some clear advantages of this approach for certain design problems, this solution is dismissed in [24] for the lack of an ontological interpretation (or ontological counterpart) for the reified properties. In the next section, we demonstrate that: (i) the ontological categories underlying OntoUML provides for a direct ontological interpretation for these reified entities in the proposed approach; and (ii) these categories can be directly employed for creating transformation patterns between OntoUML models and OWL specifications, in which at least part of the original modal semantics is retained.

4.3 Reifying Temporal Knowledge in OWL Supported by Ontological Categories

Reification is an operation that makes the reifed (objectified) entity amenable to reference, qualification and quantification. In [28], Quine presents reification as a strategy for forging links between sentences or clauses represented in a first order logic (FOL) language. For example, the sentence 'Sebastian walked slowly and aimlessly in Bologna at *t*' can be reified as $\exists x$ (x is a walk and x is slow and x is aimless and x is in Bologna and x is at t and x is by Sebastian) where x is the objective reference that connect all clauses.

In this section, we are particularly interested in reification as a strategy for representing temporal knowledge using DL-based versions of OWL. It means that we are restricted to a subset of FOL whose predicates are at most binary. For example, the statement 'John is married to Mary at *t*' is to be reified as something like $\exists x$ (isRelatedTo(x, John) $\wedge$ isRelatedTo(x, Mary) $\wedge$ holds(x, t)). Indeed, in face of this representation some questions arise: what is this thing that is related to *John* and *Mary*?

Can this thing keep existing (holding) without being related to both *John* and *Mary*? Are *John* and *Mary* related to each other in the very same way?

In the sequel, we employ the ontological notions defined in section 2 to answer such questions and to provide ontological meaning for the reified temporal knowledge. More specifically, we intend to reify/objectify the individuals' properties and to attribute to them the time interval during which they hold having a certain value. For example, the time interval during which John has the age of 27 years old, or the one during which he is married to Mary.

As mentioned, reifying the properties of an individual allows one predicating and quantifying over them. It includes attributing to them a time interval during which it is held to be true. Thereby, in Figure 10, we present two illustrative schemas of applying an ontologically-grounded reification approach to the running example in a temporal view. The object and moment individuals are represent by graphical elements in different shapes, whose projection onto the timeline corresponds to the individual's temporal extension. Moreover, the spatial inclusion of elements represents the *inherence* relation (i.e. the spatially included elements *inhere* in the container) and also reflects the temporal inclusion imposed by the existential dependence. The mandatory properties are represented as rectangles, while the optional properties are represented as rounded corner rectangles. Moreover, the mutable properties are in a lighter grey shade than those immutable ones.

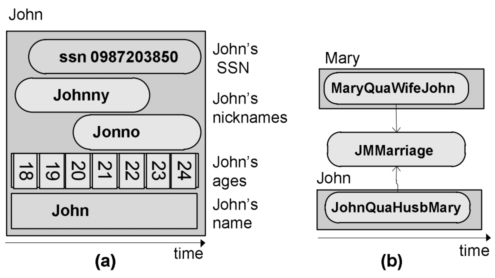


Fig. 10. A schematic representation of an object with (a) its reified qualities; (b) representing reified relators and qua individuals

In Figure 10a, the larger rectangle represents the object individual *John* that is an instance of the class *Person*; the elements contained in that rectangle represent the qualities corresponding to the reification of John´s attributes. Particularly, the quality *John's name* has the same width extension than the individual *John*, representing that it has the same temporal extension of John. In contrast, the necessary and mutable attribute *age* is represented by many qualities (*John's ages*) that together must have the same width extension than the individual *John*. The Figure 10b represents the founding relator of the material relation *marriedTo* between the object individuals *John* and *Mary*, as well as the reification of the correspondent role instantiations (qua individuals). The relator that mediates the couple is represented by the rounded corner rectangle identified as *JMMarriage*, and the qua-individuals that compose it are represented by the elements connected to it by an arrow.

In Figure 11, we propose a framework that reflects the ontological notions presented in section 2 and allows for representing temporal information in OWL. Every individual *has a temporal extent;* individuals are specialized into *moments* and *objects*; a *moment* is *existentially dependent of* at least one individual, and can be either a *relator* or an *intrinsic moment*. The former *mediates* two or more individuals, whilst the latter *inheres in* exactly one individual and can be either a *quality* or a *qua-individual*; a *quality* has one *datatype value;* a *qua individual* is *part of* one *relator* and is *existentiallyDependentOf* at least another qua-individual. The relations *inheresIn*, *mediates* and *partOf* are specializations of *existentiallyDependentOf.*

Following, the model of Figure 9 will be used as support for explaining a number of methodological guidelines discussed in the sequel which can systematically be used to specialize the framework represented in Figure 11.

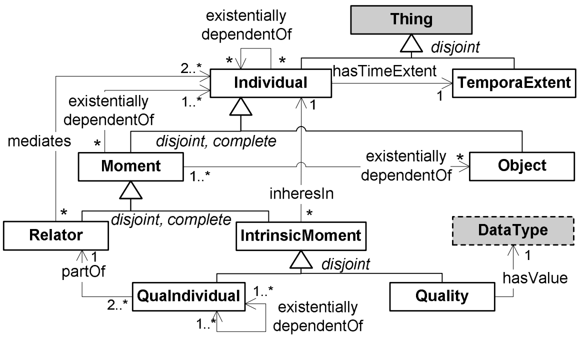


Fig. 11. An Ontology-based Framework for the systematic reification of properties in OWL. The classes depicted in gray are original OWL constructs which are then specialized by elements of the proposed framework (whose classes are depicted in white).

a. The rigid/**necessary classes** (e.g. *Person*) should specialize the class ***Object***.
b. The anti-rigid/**contingent classes** (roles) should be represented as subclasses of the class ***QuaIndividual***. This qua individual type classifies all the qua-individuals resulting from the reification of the participation of individuals of a same object class in a same material relation. For example, the class *Husband* is represented as the class *QuaHusband*, which group all the qua-individuals resulting from the reification of the participation of *Man's* individuals in the material relation *marriedTo*.
c. The **material relations** of the domain should be explicitly represented as subclasses of class ***Relator***. This relator types classifies all the relator individuals resulting from the reification of the same material relation. For example, the material relation *marriedTo* is represented as the *Marriage* class;
d. **Attributes** should be represented as subclasses of the class ***Quality***. A quality type classifies all the qualities resulting from the reification of a certain attribute of individuals of the same type. For example, the attribute *name* of the concept *Person* is represented as the class *Name*, which classifies all the quality individuals resulting from the reification of the instantiation of the attribute name of individuals of the class *Person*.

Moreover, we must restrict which and how properties can be or must be applied over the classes. We use the terms minC, maxC and exacC for referring to the minimum, maximum and exact values of cardinality holding for attributes or relation association ends, respectively.

a. every instance of a **qua-individual** class must ***inheresIn*** exactly one individual of the correspondent object class and only *inheresIn* it. For example, any individual *quaHusband* must *inheresIn* exactly one instance of *Man* and cannot *inheresIn* anything else;
b. every instance of a **qua-individual** class must be ***partOf*** exactly one individual of the correspondent relator class and only be *partOf* it. For example, any individual *quaHusband* must be *partOf* exactly one instance of *Marriage* and cannot be *partOf* anything else;
c. every instance of a **qua-individual** class must be ***existentiallyDependentOf*** all other qua-individuals participating in the same relation. For example, any individual *quaHusband* must be ***existentiallyDependentOf*** all other qua-individuals that are part of the relator *Marriage* and cannot be ***existentiallyDependentOf*** anyother qua-individual;
d. every instance of a **relator** class must ***mediates*** only individuals of the correspondent object classes (e.g. an individual of the class *Marriage* must ***mediates*** only instances of the classes *Man* or *Woman*);
e. every instance of a **relator** class must have as part (***inverse partOf***) only individuals of the qua-individual classes that inhere in the individuals of object classes that the relators mediate (e.g. any individual of the class *Marriage* must have as part only instances of the classes *QuaHusband* or *QuaWife*. These qua individuals inhere in individuals of the classes *Man* and *Woman* mediated by individuals of the class *Marriage*);
f. every instance of a **relator** class must have as part (***inverse partOf***) at least ***minC***, at most ***maxC*** or exactly ***exactC*** instances of the correspondent qua-individual classes (e.g. any individual of the class *Marriage* must be part of exactly one instance of the class *Man* and exactly one instance of the class *Woman*);
g. every instance of a **relator** class must ***mediate*** at least ***minC***, at most ***maxC*** or exactly ***exactC*** instances of the correspondent object classes (e.g. any individual of the class *Marriage* must mediate exactly one instance of the class *Man* and exactly one instance of the class *Woman*);
h. for **immutable material relation**, the domain individuals must be mediated by (***inverse mediates***) at most ***maxC*** or ***exactC*** instances of the **relator** class. Otherwise, if it is **mutable**, **no cardinality restrictions** are imposed to the number of **relators** mediating the domain individuals (***inverse mediates***);
i. every instance of a **quality** class must ***inheresIn*** exactly one individual of the correspondent object class and only *inheresIn* it. For example, any individual *Name* must *inheresIn* exactly one instance of *Person* and cannot *inheresIn* anything else;
j. every instance of a **quality** class must ***hasValue*** exactly one value of the correspondent datatype and *only it*. For example, any individual *Name* must *hasValue* exactly one *String* value and cannot be related via *hasValue* to anything else;
k. for **necessary attributes**, every instance of the correspondent object class must bear (***inverse inheresIn***) at least one instance of the **quality** class. Otherwise, for **contingent attributes**, the minimum cardinality is not restricted. For example,

every instance of the class *Person* must have at least one instance of the quality *Age* inhering in it, whilst such restriction does not hold for the quality *SSN*.

1. for **immutable attributes**, every instance of the correspondent object class must bear (***inverse inheresIn***) at most maxC or exactly exacC instances of the **quality** class. In contrast, for **mutable attributes**, the maximum cardinality is not restricted. It means that every time the attribute changes, a new *quality* individual is necessary for holding the new value. For example, every instance of the class *Person* must have at most one instance of the quality *SSN* inhering in it, whilst such restriction does not hold for the quality *Age*.

Figure 12 depicts an implementation of the running example following the proposed reification approach. Notice that possible instantiations of this model are the situations illustrated by Figures 10.a and 10.b.

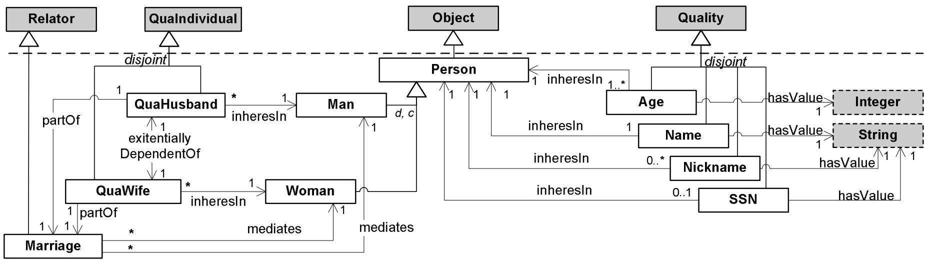


Fig. 12. Mapping the model of Figure 9 to OWL using the framework of Figure 11. In this model, the domain independent classes specializing Thing in figure 11 (i.e., the classes proposed in our framework representing different ontological categories of individuals) are depicted in full-lined grey boxes; domain-specific classes extending those are represented in full-lined white boxes. Finally, specializations of the OWL construct datatype are represented in dashed grey boxes.

4.4 Discussion

In a logical theory representing a conceptual model, time-indexed properties are often represented introducing a temporal parameter in the instantiation relation, i.e. at *t*, *x* is an instance of the property *P*. There are at least three different interpretations of this temporalization: (i) 'at *t*' is a modal operator that applies to propositions, like 'at *t* (*x* is red)'; (ii) *t* is just an additional argument that transforms unary properties in binary ones (i.e. in relations) like '*x* is red-at-*t*'; (iii) 'at *t*' is a modifier of the particular, i.e., '*x*-at-*t* is red', where '*x*-at-*t*' is, in a four dimensional view, the temporal slice of *x*.

Both option (i) and a solution somewhat similar to option (ii) are widely used in conceptual modeling (see Figures 13.a and 13.b, respectively). Option (iii) can be found in some novel proposals in data modeling (see, for instance, [29]). The view defended here allows for an alternative representation, which is similar but not equivalent to (iii). As previously discussed, this alternative view assumes that a change in an endurant is given by a substitution of moments, i.e., the temporal information is coded in the temporal extension of moments. This solution could be easily represented in Figure 8 without adding complexity to the definition of intrinsic properties. Additionally, this solution has two benefits when compared to (iii). First, as opposed to (iii), one does not necessarily commit to four-dimensionalism, since moments can be conceived as persisting entities in the same way as substantial individuals (objects). In other words, in the alternative proposed in this paper, one does not have to assume the existence of temporal slices of moments. Second, in a (Onto)UML class diagram for conceptual modeling, classes are supposed to represent persisting objects such as Apple in (iii), not snapshots of objects such as AppleSnapshot in the same model. Snapshots of objects that instantiate the types depicted in a class diagram are supposed be represented via instance diagrams.

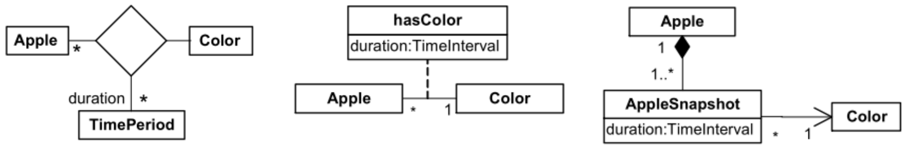


Fig. 13. Different strategies for representing temporally changing information in Conceptual Modeling: (a-left) time modality; (b-center) time-indexed relations (c) entity snapshots

Regarding the conceptual representations in Figure 13, notice that neither (a) nor (b) could be directly represented in OWL since: (i) OWL cannot represent ternary relations; (ii) OWL properties cannot have properties themselves. Regarding (c), in [25], we have proposed two alternative approaches for representing temporal information in OWL following a 4D (perdurantist) view. In both approaches, we divide the entities in two levels: individual concepts level, for the properties that do not change, and time slice level, for registering the changes on mutable properties. Although that proposal allows one to reasonably represent the intended models, those approaches have the following drawbacks:

- proliferation of time slices: any change occurred in a certain time slice leads to what we call a *proliferation of time slices*. It means that it is necessary to duplicate every time slice in the chain of connected instances that includes the instance on change;
- oddity in ontological interpretation of contingent concepts: in 4D approaches the anti-rigid classes are classes that apply only to time-slices, whilst the rigid classes apply both to 3D entities (ordinary objects, qualities and relators) and their time-slices. This makes the ontological interpretation for the anti-rigid classes (like *Husband* and *Wife*) rather odd;
- repetition of the immutable information on time slice level: the properties that are immutable but not necessary are represented at the time slice level. This leads to a tedious repetition of this information across the time slices of the same individual concept;
- not guaranteeing immutability in the time slice level: since the immutable properties represented at time slice level must be repeated across the time slices of the same individual concept, we cannot guarantee that this property value does not change across time slices.

If we compare the reification approach proposed here with these 4D-based proposals, the following can be stated regarding the aforementioned drawbacks:

- *proliferation of (time-slice) individuals*: changes no longer cause proliferation of individuals. Although we do have, in this case, the need for new (reified) individuals, the number of these individuals do not increase for each change. For this reason, under this respect, we consider the reification proposal more scalable than the 4D ones;
- *oddity in the ontological interpretation of contingent concepts*: we have homogeneous ontological interpretation for necessary and contingent concepts in the reification proposal;
- *repetition of the immutable contingent information:* except for the mutable properties, no other property is repeated in the reification proposal;
- *not guaranteeing the immutability of contingent properties*: since the immutable properties are represented just once in the reification proposal, its value cannot change.

It is important to highlight that, despite these benefits, there are also limitations and drawbacks in the reification approach. For instance, as pointed out in [24], when reifying relations, we lose the ability to (directly) associate with them meta-properties such as symmetry, reflexivity, transitivity and functionality. However, as discussed in depth in [7], the application of these meta-properties to material relation is far from a trivial issue. For instance: (i) material relations are never reflexive (since relators must mediate at least two distinct individuals); (ii) symmetry has to differentiate extensional symmetry from intentional symmetry (which can properly be represented here by the roles associated with relators); (iii) transitivity of material relations is an issue of great complexity which has been partially treated, for example, in [19], for the case of parthood relations.

In any case, this discussion highlights our argument in section 1 that there is not one single design solution that should fit all design problems. This is by itself enough a good reason for separating conceptual domain modeling from the multiple implementations which can be derived from it and which can be chosen for maximizing specific sets of non-functional requirements.

Finally, although we are aware of initiatives for addressing time representation and reasoning in OWL, we deemed this issue out of scope for this particular paper. However, having a proper axiomatization in that respect is necessary for imposing the temporal restrictions pointed out in our reification proposal, namely: (i) the existential dependence relation must imply temporal inclusion of the dependent individual in the time-extent of the individual(s) it depends on; (ii) a reified necessary and immutable property must have exactly the same time-extent of the individual it depends on; and (iii) a reified necessary and mutable property must have the temporal projection of all its individuals equal to the time-extent of the individual they depend on (i.e. the property age). These issues should be properly dealt with in a fuller approach.

5 Final Considerations

To promote semantic interoperability between information models (and applications which depend on them), we need to be able to guarantee truthfulness to the domain in reality being represented in these models (intra-model consistency). Moreover, we need to guarantee that we establish the correct relations (with the correct semantics) between elements pertaining to different models (inter-model consistency). In order to achieve these objectives, we must rely on representation mechanisms that are able to make explicit its ontological commitments and which are able to capture the subtleties of the subject domains being represented. Moreover, this should be done in a manner that are consistent with how humans as cognitive agents construct and shared their mental models of those subject domains. After all, tasks such as domain understanding, problem-solving and meaning negotiation are meant to be performed by human agents in these scenarios.

Following a tradition on what is now termed *Ontology-Driven Conceptual Modeling*, we argue in this article that these representation mechanisms should be grounded in Foundational Ontologies. In this paper, we present an ontological theory which is built on the idea of property-instances (tropes, moments, modes). This idea affords an ontology which has an illustrious pedigree in philosophy and which has corresponding support both in cognition and language. Moreover, this idea can provide an ontological interpretation and can be used to derive modeling guidelines to many conceptual modeling constructs (e.g., weak entities, reified attributes and associations, datatypes). Finally, as demonstrated in this paper, this idea provides a modeling framework for systematically representing temporally changing information in a class of description-logics/frame-based languages, represented here by the language OWL.

Acknowledgments. This research is supported by FAPES (PRONEX grant #52272362) and CNPq (productivity grant #311578/2011-0). Moreover, the authors would like to thank Claudio Masolo and Stefano Borgo for the fruitful discussions regarding "change as moment replacement" and, in particular, for important insights regarding the representation of temporal information in conceptual modeling as discussed in section 4.4.

References

1. Object Management Group, Semantic Information Model Federation (SIMF): Candidates and Gaps, `http://www.omgwiki.org/architecture-ecosystem/`
2. Mylopoulos, J.: Conceptual modeling and Telos. In: Conceptual Modeling, Databases, and CASE, ch. 2, pp. 49–68. Wiley (1992)
3. Wand, Y., Weber, R.: An ontological evaluation of systems analysis and design methods. In: Information System Concepts: An In-Depth Analysis. Elsevier Science Publishers B.V., North-Holland (1989)
4. Recker, J., Rosemann, M., Indulska, M., Green, P.: Do Ontological Deficiencies in Process Modeling Grammars Matter? MIS Quarterly 35(1), 57–80 (2011)
5. Guizzardi, G.: On Ontology, ontologies, Conceptualizations, Modeling Languages, and (Meta)Models. In: Frontiers in Artificial Intelligence and Applications, Databases and Information Systems IV. IOS Press, Amsterdam (2007) ISBN 978-1-58603-640-8
6. Masolo, C., Borgo, S., Gangemi, A., Guarino, N., Oltramari, A.: Ontology Library. WonderWeb Deliverable D18 (2003)
7. Guizzardi, G.: Ontological Foundations for Structural Conceptual Models. Universal Press, The Netherlands (2005) ISBN 1381-3617

8. Lowe, E.J.: The Four Category Ontology. Oxford University Press (2006)
9. Gärdenfors, P.: Conceptual Spaces: the Geometry of Thought. MIT Press, USA (2000)
10. Guizzardi, G., Wagner, G.: What's in a Relationship: An Ontological Analysis. In: Li, Q., Spaccapietra, S., Yu, E., Olivé, A. (eds.) ER 2008. LNCS, vol. 5231, pp. 83–97. Springer, Heidelberg (2008)
11. Wieringa, R.J., de Jonge, W., Spruit, P.A.: Using dynamic classes and role classes to model object migration. Theory and Practice of Object Systems 1(1), 61–83 (1995)
12. Guizzardi, G., Masolo, C., Borgo, S.: In Defense of a Trope-Based Ontology for Conceptual Modeling: An Example with the Foundations of Attributes, Weak Entities and Datatypes. In: Embley, D.W., Olivé, A., Ram, S. (eds.) ER 2006. LNCS, vol. 4215, pp. 112–125. Springer, Heidelberg (2006)
13. Guizzardi, G., Wagner, G., Guarino, N., van Sinderen, M.: An Ontologically Well-Founded Profile for UML Conceptual Models. In: Persson, A., Stirna, J. (eds.) CAiSE 2004. LNCS, vol. 3084, pp. 112–126. Springer, Heidelberg (2004)
14. Benevides, A.B., et al.: Validating modal aspects of OntoUML conceptual models using automatically generated visual world structures. Journal of Universal Computer Science 16(20) (2010); Special Issue on Evolving Theories of Conceptual Modeling
15. Guizzardi, G.: Ontological Meta-properties of Derived Object Types. In: Ralyté, J., Franch, X., Brinkkemper, S., Wrycza, S. (eds.) CAiSE 2012. LNCS, vol. 7328, pp. 318–333. Springer, Heidelberg (2012)
16. Halpin, T., Morgan, T.: Information Modeling and Relational Dababases. Morgan Kaufman (2008) ISBN 1558606726
17. Bauman, B.T.: Prying Apart Semantics and Implementation: Generating XML Schemata directly from ontologically sound conceptual models. In: Balisage Markup Conference (2009)
18. Fiorini, S., Abel, M., Scherer, C.: A Symbol Grounding Model for the Interpretation of 2D-line charts. In: 15th IEEE International Enterprise Computing Conference (EDOC 2010) Workshop Proceedings, Vitória, Brazil (2010)
19. Guizzardi, G.: The Problem of Transitivity of Part-Whole Relations in Conceptual Modeling Revisited. In: van Eck, P., Gordijn, J., Wieringa, R. (eds.) CAiSE 2009. LNCS, vol. 5565, pp. 94–109. Springer, Heidelberg (2009)
20. Guizzardi, G., Lopes, M., Baião, F., Falbo, R.: On the importance of truly ontological representation languages. International Journal of Information Systems Modeling and Design, IJISMD (2010) ISSN: 1947-8186
21. Guizzardi, R.S.S., Guizzardi, G.: Ontology-Based Transformation Framework from Tropos to AORML. In: Social Modeling for Requirements Engineering. Cooperative Information Systems Series. MIT Press, Boston (2010)
22. Costal, D., Gómez, C., Guizzardi, G.: Formal Semantics and Ontological Analysis for Understanding Subsetting, Specialization and Redefinition of Associations in UML. In: Jeusfeld, M., Delcambre, L., Ling, T.-W. (eds.) ER 2011. LNCS, vol. 6998, pp. 189–203. Springer, Heidelberg (2011)
23. Benevides, A.B., Guizzardi, G.: A Model-Based Tool for Conceptual Modeling and Domain Ontology Engineering in OntoUML. In: Filipe, J., Cordeiro, J. (eds.) ICEIS 2009. LNBIP, vol. 24, pp. 528–538. Springer, Heidelberg (2009)
24. Welty, C., Fikes, R.: A Reusable Ontology for Fluents in OWL. In: Proceeding of the 2006 conference on Formal Ontology in Information Systems: Proceedings of the Fourth International Conference (FOIS 2006), pp. 226–236. IOS Press (2006)

25. Zamborlini, V., Guizzardi, G.: On the representation of temporally changing information in OWL. In: 15th IEEE International Enterprise Computing Conference (EDOC 2010) Workshop Proceedings, Vitória, Brazil (2010)
26. Varzi, A.C.: Naming the stages. Dialectica 57(4), 387–412 (2003)
27. Sider, T.: Fourdimensionalism: An Ontology of Persistence and Time. Oxford University Press (2003)
28. Quine, W.V.: Events and reification. In: Actions and Events: Perspectives on the Philosophy of Davidson, pp. 162–171. Blackwell (1985)
29. West, M.: Information Modeling: An Analysis of uses and meanings of Associations. PDT Europe (2002)

Declarative Name Binding and Scope Rules

Gabriël Konat, Lennart Kats, Guido Wachsmuth, and Eelco Visser

Delft University of Technology, The Netherlands
g.d.p.konat@student.tudelft.nl,
{l.c.l.kats,g.h.wachsmuth,e.visser}@tudelft.nl

Abstract. In textual software languages, names are used to reference elements like variables, methods, classes, etc. Name resolution analyses these names in order to establish references between definition and use sites of elements. In this paper, we identify recurring patterns for name bindings in programming languages and introduce a declarative meta-language for the specification of name bindings in terms of namespaces, definition sites, use sites, and scopes. Based on such declarative name binding specifications, we provide a language-parametric algorithm for static name resolution during compile-time. We discuss the integration of the algorithm into the Spoofax Language Workbench and show how its results can be employed in semantic editor services like reference resolution, constraint checking, and content completion.

1 Introduction

Software language engineering is concerned with *linguistic abstraction*, the formalization of our understanding of domains of computation in higher-level software languages. Such languages allow direct expression in terms of the domain, instead of requiring encoding in a less specific language. They raise the level of abstraction and reduce accidental complexity. One of the key goals in the field of language engineering is to apply these techniques to the discipline itself: high-level languages to specify all aspects of software languages. Declarative languages are of particular interest since they enable language engineers to focus on the *What?* instead of the *How?*. Syntax definitions are a prominent example. With declarative formalisms such as EBNF, we can specify the syntactic concepts of a language without specifying how they can be recognized programmatically. This declarativity is crucial for language engineering. Losing it hampers evolution, maintainability, and compositionality of syntax definitions [15].

Despite the success of declarative syntax formalisms, we tend to programmatic specifications for other language aspects. Instead of specifying languages, we build programmatic language processors, following implementation patterns in rather general specification languages. These languages might still be considered domain-specific, when they provide special means for programmatic language processors. They also might be considered declarative, when they abstract over computation order. However, they enable us only to implement language processors faster, but not to specify language aspects. They lack domain concepts for these aspects and focus on the *How?*. That is a problem since (1) it entails overhead in encoding concepts in a programming language and (2) the encoding obscures the intention; understanding the definition requires decoding.

Our goal is to extend the set of really declarative, domain-specific languages for language specifications. In this paper, we are specifically concerned with *name binding and scope rules*. Name binding is concerned with the relation between definitions and references of identifiers in textual software languages, including scope rules that govern these relations. In language processors, it is crucial to make information about definitions available at the references. Therefore, traditional language processing approaches provide programmatic abstractions for name binding. These abstractions are centered around tree traversal and information propagation from definitions to references. Typically, they are not specifically addressing name binding, but can also be used for other language processing tasks such as compilation and interpretation.

Name binding plays a role in multiple language engineering processes, including editor services such as reference resolution, code completion, refactorings, type checking, and compilation. The different processes need different information about definitions. For example, name resolution tries to find one definition, while code completion needs to determine all possible references in a certain place. The different requirements lead either to multiple re-implementations of name binding rules for each of these purposes, or to non-trivial, manual weaving into a single implementation supporting all purposes. This results in code duplication with as result errors, inconsistencies, and increased maintenance effort.

The traditional paradigm influences not only language processing, but also language specification. For example, the OCL language standard [19] specifies name binding in terms of nested environments, which are maintained in a tree traversal. The C# language specification [1] defines name resolution as a sequence of imperative lookup operations. In this paper, we abstract from the programmatic mechanics of name resolution. Instead, we aim to declare the roles of language constructs in name binding and leave the resolution mechanics to a generator and run-time engine. We introduce the *Name Binding Language* (NBL), a language with linguistic abstractions for declarative definition of name binding and scope rules. NBL supports the declaration of definition and use sites of names, properties of these names associated with language constructs, namespaces for separating categories of names, scopes in which definitions are visible, and imports between scopes.

NBL is integrated in the Spoofax Language Workbench [14], but can be reused in other language processing environments. From definitions in the name binding language, a compiler generates a language-specific name resolution strategy in the Stratego rewriting language [25] by parametrizing an underlying generic, language independent strategy. Name resolution results in a persistent symbol table for use by semantic editor services such as reference resolution, consistency checking of definitions, type checking, refactoring, and code generation. The implementation supports multiple file analysis by default.

We proceed as follows. In Sect. 2 and 3 we introduce NBL by example, using a subset of the C# language. In Sect. 4 we discuss the derivation of editor services from a name binding specification. In Sect. 5 we give a high-level description of the generic name resolution algorithm underlying NBL. In Sect. 6 we discuss the integration of NBL into the Spoofax Language Workbench. Sect. 7 and 8 are for evaluation and related work.

2 Declarative Name Binding and Scope Rules

In this section we introduce the Spoofax Naming Binding Language illustrated with examples drawn from the specification of name binding for a subset of C# [1]. Fig. 1 defines the syntax of the subset in SDF [24]. The subset is by no

```
Using* NsMem*                                   → CompilationUnit {"Unit"}

"using" NsOrTypeName ";"                        → Using           {"Using"}
"using" ID "=" NsOrTypeName                     → Using           {"Alias"}
ID                                              → NsOrTypeName    {"NsOrType"}
NsOrTypeName "." ID                             → NsOrTypeName    {"NsOrType"}

"namespace" ID "{" Using* NsMem* "}"            → NsMem           {"Namespace"}
Partial "class" ID Base "{" ClassMem* "}"       → NsMem           {"Class"}

                                                → Partial         {"NonPartial"}
"partial"                                       → Partial         {"Partial"}
                                                → Base            {"NoBase"}
":" ID                                          → Base            {"Base"}

Type ID ";"                                     → ClassMem        {"Field"}
RetType ID "(" {Param ","}* ")" Block ";"       → ClassMem        {"Method"}

ID                                              → Type            {"ClassType"}
"int"                                           → Type            {"IntType"}
"bool"                                          → Type            {"BoolType"}
Type                                            → RetType
"void"                                          → RetType         {"Void"}
Type ID                                         → Param           {"Param"}

"{" Stmt* "}"                                   → Block           {"Block"}
Decl                                            → Stmt
EmbStmt                                         → Stmt
"return" Exp ";"                                → Stmt            {"Return"}
Type ID ";"                                     → Decl            {"Var"}
Type ID "=" Exp ";"                             → Decl            {"Var"}
Block                                           → EmbStmt
StmtExp ";"                                     → EmbStmt
"foreach" "(" Type ID "in" Exp ")" EmbStmt      → EmbStmt         {"Foreach"}

INT                                             → Exp             {"IntLit"}
"true"                                          → Exp             {"True"}
"false"                                         → Exp             {"False"}
ID                                              → Exp             {"VarRef"}
StmtExp                                         → Exp
Exp "." ID                                      → StmtExp         {"FieldAccess"}
Exp "." ID "(" {Exp ","}* ")"                   → StmtExp         {"Call"}
ID "(" {Exp ","}* ")"                           → StmtExp         {"Call"}
```

Fig. 1. Syntax definition in SDF for a subset of C#. The names in the annotations are abstract syntax tree constructors.

means complete; it has been selected to model representative features of name binding rules in programming and domain-specific languages. In the following subsections we discuss the following fundamental concepts of name binding: *definition and use sites*, *namespaces*, *scopes*, and *imports*. For each concept we give a general definition, illustrate it with an example in C#, and then we show how the concept can be modeled in NBL.

2.1 Definitions and References

The essence of name binding is establishing relations between a *definition* that *binds* a name and a *reference* that *uses* that name. Name binding is typically defined programmatically through a *name resolution algorithm* that connects references to definitions. A *definition site* is the location of a definition in a program. In many cases, definition sites are required to be *unique*, that is, there should be exactly one definition site for each name. However, there are cases where definition sites are allowed to be *non-unique*.

```
class C1 {}
class C2:C1 {}
partial class C3:C2 {}
partial class C3 {}
```

Fig. 2. Class declarations in C#

Example. Figure 2 contains class definitions in C#. Each class definition binds the name of a class. Thus, we have definition sites for `C1`, `C2`, and `C3`. Base class specifications are references to these definition sites. In the example, we have references to `C1` as the base class of `C2` and `C2` as the base class of `C3`. (Thus, `C2` is a sub-class of, or inherits from `C1`.) There is no reference to `C3`. The definition sites for `C1` and `C2` are unique. By contrast, there are two definition sites for `C3`, defining parts of the same class `C3`. Thus, these definition sites are non-unique. This is correct in C#, since regular class definitions are required to be unique, while partial class definitions are allowed to be non-unique.

Abstract Syntax Terms. In Spoofax abstract syntax trees (ASTs) are represented using first-order terms. Terms consist of strings (`"x"`), lists of terms (`["x","y"]`), and constructor applications (`ClassType("C1")`) for labelled tree nodes with a fixed number of children. Annotations in grammar productions (Fig. 1) define the constructors to be used in AST construction. For example, `Class(Partial(), "C3", Base("C2"), [])` is the representation of the first partial class in Figure 2. A term *pattern* is a term that may contain variables (`x`) and wildcards (`_`).

Model. A specification in the name binding language consists of a collection of rules of the form `pattern : clause*`, where `pattern` is a term pattern and `clause*` is a list of name binding declarations about the language construct that matches with **`pattern`**. Figure 3 shows a declaration of the definitions and references for class names in C#. The first two rules declare class definition sites for class names. Their patterns distinguish regular (non-partial) and partial class declarations. While non-partial class declarations are unique definition sites, partial class declarations are non-unique definition sites. The third rule declares that the term pattern `Base(c)` is a reference to a class with name `c`.

```
rules
  Class(NonPartial(), c, _, _): defines unique class c
  Class(Partial(), c, _, _)   : defines non-unique class c
  Base(c)                     : refers to class c
  ClassType(c)                : refers to class c
```

Fig. 3. Declaration of definitions and references for class names in C#

Thus, the `": C1"` in Figure 2 is a reference to class `C1`. Similarly, the second rule declares a class type as a reference to a class.

2.2 Namespaces

Definitions and references declare relations between named program elements and their uses. Languages typically distinguish several *namespaces*, i.e. different kinds of names, such that an occurrence of a name in one namespace is not related to an occurrence of that same name in another.

Example. Figure 4 shows several definitions for the same name `x`, but of different kinds, namely a class, a field, a method, and a variable. Each of these kinds has its own namespace in C#, and each of these namespaces has its own name `x`. This enables us to distinguish the definition sites of class `x`, field `x`, method `x`, and variable `x`, which are all unique.

```
class x {
  int x;
  void x() {
    int x; x = x + 1;
  }
}
```

Fig. 4. Homonym class, field, method, and variable declarations in C#

Model. We declared definitions and references for the namespace `class` already in the previous example. Figure 5 extends that declaration covering also the namespaces `field`, `method`, and `variable`. Note that it is required to declare namespaces to ensure the consistency of name binding rules. Definition sites are bound to a single namespace (**`defines`** *`class`* `c`), but use sites are not. For example, a variable in an expression might either refer to a variable, or to a field, which is modeled in the last rule. In our example, this means that variable declarations hide field declarations, because variables are resolved to variables, if possible. Thus, both `x` in the assignment in Figure 4 refer to the variable `x`.

2.3 Scopes

Scopes restrict the visibility of definition sites. A *named scope* is the definition site for a name which scopes other definition sites. By contrast, an *anonymous*

```
namespaces class field method variable
rules
  Field(_, f)         : defines unique field f
  Method(_, m, _, _): defines unique method m
  Call(m, _)          : refers to method m

  Var(_, v): defines unique variable v
  VarRef(x): refers to variable x otherwise to field x
```

Fig. 5. Declaration of name bindings for different namespaces in C#

```
 1  class C {
 2      void m() { int x; }
 3  }
 4
 5  class D {
 6    void m() {
 7      int x;
 8      int y;
 9      { int x; x = y + 1; }
10      x = y + 1;
11    }
12  }
```

Fig. 6. Scoped homonym method and variable declarations in C#

```
rules
  Class(NonPartial(), c, _, _):
    defines unique class c
    scopes field, method

  Class(Partial(), c, _, _):
    defines non-unique class c
    scopes field, method

  Method(_, m, _, _):
    defines unique method m
    scopes variable

  Block(_): scopes variable
```

Fig. 7. Declaration of scopes for different namespaces in C#

scope does not define a name. Scopes can be nested and name resolution typically looks for definition sites from inner to outer scopes.

Example. Figure 6 includes two definition sites for a method `m`. These definition sites are not distinguishable by their namespace `method` and their name `m`, but, they are distinguishable by the scope they are in. The first definition site resides in class `C`, the second one in class `D`. In C#, class declarations scope method declarations. They introduce named scopes, because class declarations are definition sites for class names. The listing also contains three definition sites for a variable `x`. Again, these are distinguishable by their scope. In C#, method declarations and blocks scope variable declarations. Method declarations are named scopes, blocks are anonymous scopes. The first definition site resides in method `m` in class `C`, the second one in method `m` in class `D`, and the last one in a nameless block inside method `m` in class `D`. In the assignment inside the block (line 9), `x` refers to the variable declaration in the same block, while the `x` in the outer assignment (line 10) refers to the variable declaration outside the block. In both assignments, `y` refers to the variable declaration in the outer scope, because the block does not contain a definition site for `y`.

Model. The **`scopes`** `ns` clause in NBL declares a construct to be a scope for namespace `ns`. Figure 7 declares scopes for fields, methods, and variables. Named scopes are declared at definition sites. Anonymous scopes are declared similarly, but lack a `defines` clause.

Namespaces as Language Concepts. C# has a notion of 'namespaces'. It is important to distinguish these *namespaces as a language concept* from *namespaces as a naming concept*, which group names of different kinds of declarations. Specifically, in C#, namespace declarations are top-level scopes for class declarations. Namespace declarations can be nested. Figure 8 declares a top-level namespace `N`, scoping a class declaration `N` and an inner namespace declaration `N`.

```
namespace N {
  class N {}
  namespace N { class N {} }
}
```

Fig. 8. Nested namespace declarations in C#

```
namespaces namespace
rules
  Namespace(n, _):
    defines namespace n
    scopes namespace, class
```

Fig. 9. Declaration of name bindings for nested namespace declarations in C#

The inner namespace declaration scopes another class declaration `N`. The definition sites of the namespace name `N` and the class name `N` are distinguishable, because they belong to different namespaces (as a naming concept). The two definition sites of namespace name `N` are distinguishable by scope. The outer namespace declaration scopes the inner one. Also, the definition sites of the class name `N` are distinguishable by scope. The first one is scoped by the outer namespace declaration, while the second one is scoped by both namespace declarations.

Model. The names of C# namespace declarations are distinguishable from names of classes, fields, etc. As declared in Figure 9, their names belong to the `namespace` namespace. The name binding rules for definition sites of names of this namespace models the scoping nature of C# namespace declarations.

Imports. An import introduces into the current scope definitions from another scope, either under the same name or under a new name. An import that imports all definitions can be transitive.

Example. Figure 10 shows different kinds of imports in C#. First, a `using` directive imports type declarations from namespace `N`. Second, another `using` directive imports class `C` from namespace `M` into namespace `O` under a new name `D`. Finally, classes `E` and `F` import fields and methods from their base classes. These imports are transitive, that is, `F` imports fields and methods from `E` and `D`.

Model. Figure 11 shows name binding rules for import mechanisms in C#. The first rule handles `using` declarations, which import all classes from the namespace to which the qualified name `qname` resolves to. The second rule models aliases, which either import a namespace or a class under a new name, depending on the resolution of `qname`. The last rule models inheritance, where fields and methods are imported transitively from the base classes.

2.4 Types

So far, we discussed names, namespaces, and scopes to distinguish definition sites for the same name. Types also play a role in name resolution and can be used to distinguish definition sites for a name or to find corresponding definition sites for a use site.

```
using N;

namespace M {
  class C { int f; }
}

namespace O {
  using D = M.C;
  class E:D {
    void m() {}
  }
  class F:E { }
}
```

Fig. 10. Various forms of imports in C#

```
rules
  Using(qname):
    imports class from namespace ns
    where qname refers to namespace ns

  Alias(alias, qname):
    imports namespace ns as alias
    where qname refers to namespace ns
    otherwise imports class c as alias
    where qname refers to class c

  Base(c):
    imports field (transitive),
            method (transitive)
      from class c
```

Fig. 11. Declaration of import mechanisms in C#

Example. Figure 12 shows a number of overloaded method declarations. These share the same name `m`, namespace `method`, and scope class `C`. But we can distinguish them by the types of their parameters. Furthermore, all method calls inside method `x` can be uniquely resolved to one of these methods by taking the argument types of the calls into account.

```
class C {
  void m() {}
  void m(int x) {}
  void m(bool x) {}
  void m(int x, int y) {}
  void m(bool x, bool y) {}

  void x() {
    m();
    m(42);
    m(true);
    m(21, 21);
    m(true, false);
  }
}
```

Fig. 12. Overloaded method declarations in C#

Model. Figure 13 includes type information into name binding rules for fields, methods, and variables. Definition sites might have types. In the simplest case, the type is part of the declaration. In the example, this holds for parameters. For method calls, the type of the definition site for a method name depends on the types of the parameters. A type system is needed to connect the type of a single parameter, as declared in the rule for parameters, and the type of a list of parameters, as required in the rule for methods. We will discuss the influence of a type system and the interaction between name and type analysis later. For now, we assume that the type of a list of parameters is a list of types of these parameters.

Type information is also needed to resolve method calls to possibly overloaded methods. The `refers` clause for method calls therefore requires the corresponding definition site to match the type of the arguments. Again, we omit the details how this type can be determined. We also do not consider subtyping here. Method calls and corresponding method declarations need to have the same argument and parameter types.

3 Name Binding Patterns

We now identify typical name binding patterns. These patterns are formed by scopes, definition sites and their visibility, and use sites referencing these definition sites. We explain each pattern first and give an example in C# next. Afterwards, we show how the example can be modelled with declarative name binding rules.

```
rules
  Method(t, m, p*, _):
    defines unique method m of type (t*, t)
    where p* has type t*

  Call(m, a*):
    refers to method m of type (t*, _)
    where a* has type t*

  Param(t, p): defines unique variable p of type t
```

Fig. 13. Types in name binding rules for overloaded methods in C#

Unscoped Definition Sites. In the simplest case, definition sites are not scoped and globally visible.

Example. In C#, namespace and class declarations (as well as any other type declaration) can be unscoped. They are globally visible across file boundaries. For example, the classes `C1`, `C2`, and `C3` in Figure 2 are globally visible. In Figure 4, only the outer namespace `N` is globally visible.

In contrast to C#, C++ has file scopes and all top-level declarations are only visible in a file. To share global declarations, each file has to repeat the declaration and mark it as `extern`. This is typically achieved by importing a shared header file.

```
rules
  CompilationUnit(_, _):
    scopes namespace, class

  (f, CompilationUnit(_, _)):
    defines file f
    scopes namespace, class
```

Fig. 14. Different ways to model file scope for top-level syntax tree nodes

Model. We consider any definition site that is not scoped by another definition site or by an anonymous scope to be in global scope. These definition sites are visible over file boundaries. File scope can be modelled with a scoping rule in two different ways. Both are illustrated in Figure 14. The first rule declares the top-level node of abstract syntax trees as a scope for all namespaces which can have top-level declarations. This scope will be anonymous, because the top-level node cannot be a definition site (otherwise this definition site would be globally visible). The second rule declares a tuple consisting of file name and the abstract syntax tree as a scope. This tuple will be considered a definition site for the file name. Thus, the scope will be named after the file.

Definition Sites Inside Their Scopes. Typically, definition sites reside inside the scopes where they are visible. Such definition sites can either be visible only after their declaration, or everywhere in their surrounding scope.

Example. In C#, namespace members such as nested namespace declarations and class declarations are visible in their surrounding scope. The same holds for class members. In contrast, variable declarations inside a method scope become visible only after their declaration.

Model. Scoped definition sites are by default visible in the complete scope. Optionally, this can be stated explicitly in `defines` clauses. Figure 15 illustrates this for namespace declarations. The second rule in this listing shows how to model definition sites which become visible only after their declaration.

Definition Sites Outside Their Scopes Some declarations include not only the definition site for a name, but also the scope for this definition site. In such declarations, the definition site resides outside its scope.

```
rules
  Namespace(n, _):
    defines non-unique namespace n in surrounding scope

  Var(t, c):
    defines unique variable of type t in subsequent scope
```

Fig. 15. Declaration of the visibility of definition sites inside scopes

Example. Let expressions are a classical example for definition sites outside their scopes. In C#, `foreach` statements declare iterator variables, which are visible in embedded statement. Figure 16 shows a method with a parameter `x`, followed by a `foreach` statement with an iterator variable of the same name. This is considered incorrect in C#, because definition sites for variable names in inner scopes collide with definition sites of the same name in outer scopes. However, the use sites can still be resolved based on the scopes of the definition sites. The use site for `x` inside the loop refers to the iterator variable, while the `x` in the collection expression refers to the parameter.

```
class C {
  void m(int[] x) {
    foreach (int x in x)
      System.Console.WriteLine(x);
  }
}
```

Fig. 16. `foreach` loop with scoped iterator variable `x` in C#

Model. Figure 17 shows the name binding rule for `foreach` loops, stating the scope of the variable explicitly. Note that definition sites which become visible after their declaration are a special case of this pattern. Figure 18 illustrates how this can be modelled in the same way as the `foreach` loop. The first rule assumes a nested representation of statement sequences, while the second rule assumes a list of statements.

Contextual Use Sites. Definition sites can be referenced by use sites outside of their scopes. These use sites appear in a context which determines the scope into which they refer. This context can either be a direct reference to this scope, or has a type which determines the scope.

```
using N.N.N;

namespace N' {
  class C {
    C f;
    void m(C p) { }
  }
  class D {
    void m(C p) {
      p.m(p.f);
    }
  }
}
```

Fig. 19. Contextual use sites in C#

Example. In C#, namespace members can be imported into other namespaces. Figure 8 shows a class `N` in a nested namespace. In Figure 19, this class is imported. The `using` directive refers to the class with a qualified name. The first part of this name refers to the outer namespace `N`. It is the context of the second part, which refers to the inner namespace `N`. The second part is then the context for the last part of the qualified name, which refers to the class `N` inside the inner namespace.

```
rules
  Foreach(t, v, exp, body):
    defines unique variable v of type t in body
```

Fig. 17. Declaration of definition sites outside of their scopes

```
rules
  Seq(Var(t, v), stmts):
    defines unique variable v of type t in stmts

  [Var(t, v) | stmts]:
    defines unique variable v of type t in stmts
```

Fig. 18. Alternative declaration of definition sites becoming visible after their declaration

```
rules
  NsOrType(n1, n2):
    refers to namespace n2 in ns
    otherwise to class n2 in ns
    where n1 refers to namespace ns

  FieldAccess(e, f):
    refers to field f in c
    where e has type ClassType(c)

  MethodCall(e, m, p*):
    refers to method m of type (t*, _) in c
    where e has type ClassType(c)
    where p* has type t*
```

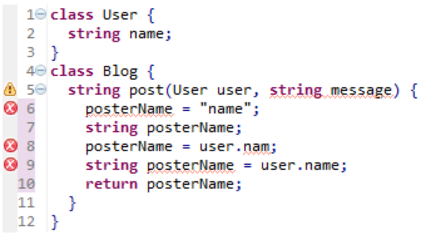
Fig. 20. Declaration of contextual use sites

Figure 19 also illustrates use sites in a type-based context. In method `m` in class `D`, a field `f` is accessed. The corresponding definition site is outside the scope of the method in class `C`. But this scope is given by the type of `p`, which is the context for the field access. Similarly, the method call is resolved to method `m` in class `C` because of the type of `p`.

Model. Figure 20 illustrates how to model contextual use sites. The scope of the declaration site corresponding to a use site can be modelled in `refers` clauses. This scope needs to be determined from the context of the use site. The first rule resolves the context of a qualified name part to a namespace `ns` and declares the use site to refer either to a namespace or to a class in `ns`. The remaining rules declare use sites for field access and method calls. They determine the type of the context, which needs to be a class type. A field access refers to a field in that class. Similarly, a method call refers to a method with the right parameter types in that class.

4 Editor Services

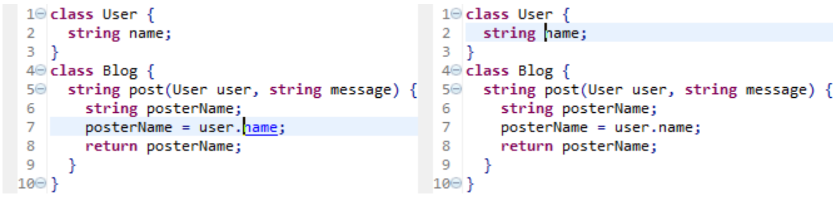
Modern IDEs provide a wide range of editor services where name resolution plays a large role. Traditionally, each of these services would be handcrafted for each language supported by the IDE, requiring substantial effort. However, by accurately modeling the relations between names in NBL, it is possible to generate a name resolution algorithm and editor services that are based on that algorithm.



```
class User {
  string name;
}
class Blog {
  string post(User user, string message) {
    posterName = "name";
    string posterName;
    posterName = user.nam;
    string posterName = user.name;
    return posterName;
  }
}
```

Fig. 21. Error checking

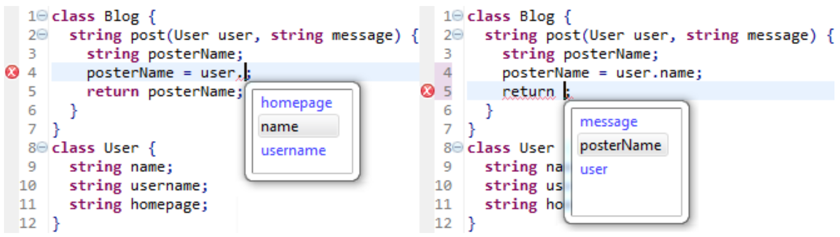
Reference Resolving. Name resolution is exposed directly in the IDE in the form of reference resolving: press and hold Control and hover the mouse cursor over an identifier to reveal a blue hyperlink that leads to its definition side. This behavior is illustrated in Fig. 22.



```
class User {
  string name;
}
class Blog {
  string post(User user, string message) {
    string posterName;
    posterName = user.name;
    return posterName;
  }
}
```

```
class User {
  string name;
}
class Blog {
  string post(User user, string message) {
    string posterName;
    posterName = user.name;
    return posterName;
  }
}
```

Fig. 22. Reference resolution of `name` field reference to `name` field definition



```
class Blog {
  string post(User user, string message) {
    string posterName;
    posterName = user.;
    return posterName;
  }
}
class User {
  string name;
  string username;
  string homepage;
}
```

```
class Blog {
  string post(User user, string message) {
    string posterName;
    posterName = user.name;
    return ;
  }
}
class User
  string na
  string us
  string ho
}
```

Fig. 23. Code completion for fields and local variables

Constraint Checking. Modern IDEs statically check programs against a wide range of constraints. Constraint checking is done on the fly while typing and directly displayed in the editor via error markers on the text and in the outline view. Error checking constraints are generated from the NBL for common name binding errors such as unresolved references, duplicate definitions, use before definition and unused definitions. Fig. 21 shows an editor with error markers. The `message` parameter in the `post` method has a warning marker indicating that it is not used in the method body. On the line that follows it, the `posterName` variable is assigned but has not yet been declared, violating the visibility rules of Figure 15. Other errors in the method include a subsequent duplicate definition of `posterName`, which violates the uniqueness constraint of the `variable` namespace of Figure 5, and referencing a non-existent property `nam`.

Code Completion. With code completion, partial (or empty) identifiers can be completed to full identifiers that are valid at the context where code completion is executed. Figure 23 shows an example of code completion. In the left program code completion is triggered on a field access expression on the `user` object. The `user` object is of type `User`, so all fields of `User` are shown as candidates. On the right, completion is triggered on a variable reference, so all variables in the current scope are shown.

5 Implementation

To implement name resolution based on NBL, we employ a name resolution algorithm that relies on a symbol table data structure to persist name bindings and lazy evaluation to resolve all references. In this section we give an overview of the data structure, the name resolution algorithm, and their implementation.

Persistence of Name Bindings. To persist name bindings, each definition and reference is assigned a qualified name in the form of a URI. The URI identifies the occurrence across a project. Use sites share the URIs of their corresponding definition sites.

A URI consists of the namespace, the path, and the name of a definition site. As an example, the URI `method://N/C/m` is assigned to a method `m` in a class `C` in a namespace `N`. Here, the segments represent the names of the scopes. Anonymous scopes are represented by a special path segment `anon(u)`, where `u` is a unique string to distinguish different anonymous scopes. For use in analyses and transformations, URIs can be represented in the form of ATerms, e.g. `[method(),"N","C","m"]` is URI for the method `m`.

All name bindings are persisted in an in-memory data structure called the semantic index. It consists of a symbol table that lists all URIs that exist in a project, and can be efficiently implemented as a hash table. It maps each URI to the file and offset of their occurrences in the project. It can also store additional information, such as the type of a definition.

Resolving Names. Our algorithm is divided into three phases. First, in the annotation phase, all definition and use sites are assigned a preliminary URI, and definition sites are stored in the index. Second, definition sites are analyzed, and their types are stored in the index. And third, any unresolved references are resolved and stored in the index.

Annotation Phase. In the first phase, the AST of the input file is traversed in top-down order. The logical nesting hierarchy of programs follows from the AST, and is used to assign URIs to definition sites. For example, as the traversal enters the outer namespace scope `n`, any definitions inside it are assigned a URI that starts with '`n.`'. As a result of the annotation phase, all definition and use sites are annotated with a URI. In the case of definition sites, this is the definitive URI that identifies the definition across the project. For references, a temporary URI is assigned that indicates its context, but the actual definition it points to has to be resolved in a following phase. For reference by the following phases, all definitions are also stored in the index.

Definition Site Analysis Phase. The second phase analyzes each definition site in another top-down traversal. It determines any local information about the definition, such as its type, and stores it in the index so it can be referenced elsewhere. Types and other information that cannot be determined locally are determined and stored in the index in the last phase.

Use Site Analysis Phase. When the last phase commences, all local information about definitions has been stored in the index, and non-local information about definitions and uses in other files is available. What remains is to resolve references and to determine types that depend on non-local information (in particular, inferred types). While providing a full description of the use site analysis phase and the implementation of all name binding constructs is outside the scope of this paper, the below steps sketch how each reference is resolved. See the NBL website [1] for links to the algorithm's source files.

1. Determine the temporary URI `ns://path/n` which was annotated in the first analysis phase.
2. If an import exists in scope, expand the current URI for that import.
3. If the reference corresponds to a name-binding rule that depends on non-local information such as types, retrieve that information.

[1] `http://strategoxt.org/Spoofax/NBL`

4. Look for a definition in the index with namespace `ns`, path `path`, and name `n`. If it does not exist, try again with a prefix of `path` that is one segment shorter. If the no definition is found this way, store an error for the reference.
5. If the definition is an alias, resolve it.

An important part to highlight in the algorithm is the interaction between name and type analysis that happens for example with the `FieldAccess` expression of Figure 20. For name binding rules that depend on types or other non-local information, it is possible that determining the type recursively triggers name resolution. For this reason, we apply lazy evaluation, ensuring that any reference can be resolved lazily as requested in this phase. By traversing through the entire tree, we ensure that all use sites are eventually resolved and persisted to the index.

6 Integration into Spoofax

The NBL, together with the index, is integrated into the Spoofax Language Workbench. Stratego rules are generated by the NBL that use the index API to interface with Spoofax. In this section we will show the index API and how the API is used to integrate the editor services seen in Section 4.

Index API. Once all analysis phases have been completed, the index is filled with a summary of every file. To use the summaries we provide the index API with a number of lookups and queries. Lookups transform annotated identifiers into definitions. Queries transform definitions (retrieved using a lookup) into other data. The API is used for integrating editor services, but is also exposed to Spoofax language developers for specifying additional editor services or other transformations.

`index-lookup-one` performs a lookup that looks for a definition of given identifier in its owning scope. The `index-lookup` lookup performs a lookup that tries to look for a definition using `index-lookup-one`. If it cannot be found, the lookup is restarted on the outer scope until the root scope is reached. If no definition is found at the root scope, the lookup fails. There is also an `index-lookup-all` variant that returns all found definitions instead of stopping at the first found definition. Finally, `index-lookup-all-levels` is a special version of `index-lookup-all` that supports partial identifiers.

```
editor-complete:
  ast → identifiers
  where
    node@COMPLETION(name) := <collect-one(?COMPLETION(_))> ast ;
    proposals             := <index-lookup-all-levels(|name)> node ;
    identifiers           := <map(index-uri-name)> proposals
```

Fig. 25. Code completion

To get data from the index, `index-get-data` is used. Given a definition and a data kind, it will return all data values of that kind that is attached to the definition. Uses are retrieved in the same way using `index-get-uses-all`.

Reference Resolution. Resolving a reference to its definition is very straightforward when using `index−lookup`, since it does all the work for us. The only thing that has to be done when Spoofax requests a reference lookup is a simple transformation: `node →<index−lookup> node`. The resulting definition has location information embedded into it which is used to navigate to the reference. If the lookup fails, this is propagated back to Spoofax and no blue hyperlink will appear on the node under the cursor.

Constraint Checking. Constraint checking rules are called by Spoofax after analysis on every AST node. If a constraint rule succeeds it will return the message and the node where the error marker should be put on.

```
constraint−error:
  node → (key, "Duplicate␣definition")
  where
    <nam−unique> node ;
    key  := <nam−key> node ;
    defs := <index−lookup−one> key ;
    <gt> (<length> defs, 1)
```

Fig. 24. Duplicate definitions constraint check

The duplicate definition constraint check that was shown earlier is defined in Figure 24. First `nam−unique` (generated for unique definitions by the NBL) is used to see if the node represents a unique definition; non-unique definition such as partial classes should not get duplicate definition error markers. The identifier is retrieved using `nam−key` and a lookup in the current scope is done with `index−lookup−one`. If more than one definition is found, the constraint check succeeds and an error marker is shown on the node.

Code Completion. When code completion is requested in Spoofax, a completion node is substituted at the place where the cursor is. For example, if we request code completion on `VarRef("a")`, it will be substituted by `VarRef(COMPLETION("a"))` to indicate that the user wants to complete this identifier. See Figure 25 for the code completion implementation. We first retrieve the completion node and name using `collect−one`. Completion proposals are gathered by `index−lookup−all−levels` since it can handle partial identifiers. Finally the retrieved proposals are converted to names by mapping `index−uri−name` over them.

7 Evaluation and Discussion

Our aim with this work has been to design high-level abstractions for name resolution applicable to a wide range of programming languages. In this section we discuss the limitations of our approach and evaluate its applicability to different languages and other language features than those covered in the preceding sections.

Limitations. There are two areas of possible limitations of NBL. One is in the provided abstraction, the other is in the implementation algorithm that supports it. As for the provided abstraction, as a definition language, NBL is inherently limited in the number of features it can support. While the feature space it supports is extensive, ultimately there may always be language features or variations that are not supported. For these cases, the definition of NBL, written in Stratego, can be extended, or it is possible to escape NBL and extend an NBL specification using handwritten Stratego rules. As for the implementation algorithm, NBL's current implementation strategy relies on laziness, and does not provide much control over the traversal for the computation of names or types. In particular, sophisticated type inference schemes are not supported with the current algorithm. To implement such schemes, the algorithm would have to be extended, preferably in a way that maintains compatibility with the current NBL definition language.

Coverage. During the design and construction of NBL, we have performed a number of studies on languages and language features to determine the extent of the feature space that NBL would support. In this paper we highlighted many of the features by using C# as a running example, but other languages that we studied include a subset of general-purpose programming languages C, Java, and domain-specific languages WebDSL [10], the Hibernate Query Language (HQL), and Mobl [12]. We also applied our approach to the Java Bytecode stack machine language using the Jasmin [17] syntax.

For our studies we used earlier prototypes of NBL, which led to the design as it is now. Notable features that we studied and support in NBL are partial classes, inheritance, visibility, lexical scoping, imports, type-based name resolution, and overloading; all of which have been discussed in Sect. 4. In addition, we studied aspect-oriented programming with intertype declarations and pointcuts, file-based scopes in C, and other features. Our design has also been influenced by past language definitions, such as SDF and Stratego. Altogether, it is fair to say that NBL supports a wide range of language features and extensive variability, but can only support the full range of possible programming languages by allowing language engineers to escape the abstraction. In future work, we would like to enhance the possibilities of extending NBL and design a better interface for escapes.

8 Related Work

We give an overview of other approaches for specifying and implementing name resolution. The main distinguishing feature of our approach is the use of linguistic abstractions for name bindings, thus hiding the low level details of writing name analysis implementations.

Symbol Tables. In classic compiler construction, symbol tables are used to associate identifiers with information about their definition sites. This typically includes type information. Symbol tables are commonly implemented using hash tables where the identifiers are indexed for fast lookup. Scoping of identifiers can be implemented in a number of ways; for example by using qualified identifiers as index, nesting symbol tables or destructively updating the table during program analysis. The type of symbol table influences the lookup strategy. When using qualified identifiers the entire identifier can be looked up efficiently, but considering outer scopes requires multiple lookups. Nesting symbol tables always requires multiple lookups but is more memory efficient. When destructively updating the symbol table, lookups for visible variables are very efficient, but the symbol table is not available after program analysis. The index we use is a symbol table that uses qualified identifiers. We map qualified identifiers (URIs) to information such as definitions, types and uses.

Attribute Grammars. Attribute Grammars [16] (AGs) are a formal way of declaratively specifying and evaluating attributes for productions in formal grammars. Attribute values are associated with nodes and calculated in one or more tree traversals, where the order of computations is determined by dependencies between attributes.

Eli provides an attribute grammar specification language for modular and reusable attribute computations [13]. Abstract, language-independent computations can be reused in many languages by letting symbols from a concrete language inherit these computations. For example, computations `Range`, `IdDef`, and `IdUse` would calculate a scope, definitions, and references. A method definition can then inherit from *Range* and *IdDef*, because it defines a function and opens a scope. A method call inherits from *IdUse* because it references a function. These abstract computations are reflected by naming concepts of NBL and the underlying generic resolution algorithm. However, NBL is less expressive, more domain-specific. Where Eli can be used to specify general (and reusable) computations on trees, NBL is restricted to name binding concepts, helping to understand and specify name bindings more easily.

Silver [26] is an extensible attribute grammar specification language which can be extended with general-purpose and domain-specific features. Typical examples are auto-copying, pattern matching, collection attributes, and support for data-flow analysis. However, name analysis is mostly done the traditional way; an environment with bindings is passed down the tree using inherited properties.

Reference Attribute Grammars (RAGs) extend AGs by introducing attributes that can reference nodes. This substantially simplifies name resolution implementations. JastAdd [7] is a meta-compilation system for generating language processors relying on RAGs and object orientation. It also supports parametrized attributes to act as functions where the value depends on the given parameters. A typical name resolution as seen in [5,7,2] is implemented in lookup attributes parameterised by an identifier of use sites, such as variable references. All nodes that can have a variable reference as a child node, such as a method body, then have to provide an equation for performing the lookup. These equations implement scoping and ordering using Java code. JastAdd implementations have much more low level details than NBL declarations. This provides flexibility, but entails overhead on encoding and requires decoding for understanding. For example, scopes for certain program elements are encoded within a set of equations, usually implemented by early or late returns.

Visibility Predicates. CADET [20] is a notation for predicates and functions over abstract syntax tree nodes. Similar to attribute grammar formalisms, it allows to specify general computations in trees but lacks reusable concepts for name binding. Poetsch-Heffter proposes dedicated name binding predicates [21], which can be translated into efficient name resolution functions [22]. In contrast to NBL, scopes are expressed in terms of start and end points and multi-file analyses are not supported.

Dynamic Rewrite Rules. In term rewriting, an environment passing style does not compose well with generic traversals. As an alternative, Stratego allows rewrite rules to create dynamic rewrite rules at run-time [3]. The generated rules can access variables available from their definition context. Rules generated within a rule scope are automatically retracted at the end of that scope. Hemel et al. [11] describe idioms for applying dynamic rules and generic traversals for composing definitions of name analysis, type analysis, and transformations without explicitly staging them into different phases. Our current work builds on the same principles, but applies an external index and provides a specialized language for name binding declarations.

Name analysis with scoped dynamic rules is based on consistent renaming, where all names in a program are renamed such that they are unequal to all other names that do not correspond to the same definition site. Instead of changing the names directly in the tree, annotations can be added which ensure uniqueness. This way, the abstract syntax tree remains the same modulo annotations. Furthermore, unscoped dynamic rewrite rules can be used for persistent mappings [14].

Textual Language Workbenches. Xtext [6] is a framework for developing textual software languages. The Xtext Grammar Language is used to specify abstract and concrete syntax, but also name bindings by using cross-references in the grammar. Use sites are then automatically resolved by a simplistic resolution algorithm. Scoping or visibility cannot be defined in the Grammar Language, but have to be implemented in Java with help of a scoping API with some default resolvers. For example field access, method calls, and block scopes would all need custom Java implementations. Only package imports have special support and can be specified directly in the Grammar Language. Common constraint checks such as duplicate definitions, use before definition, and unused definitions also have to be specified manually. This increases the amount of boilerplate code that has to be rewritten for every language.

In contrast to Xtext's Grammar Language, NBL definitions are separated from syntax definitions in Spoofax. This separation allows us to specify more advanced name binding concepts without cluttering the grammar with these concepts. It also preserves language modularity. When syntax definitions are reused in different contexts, different name bindings can be defined for these contexts, without changing the grammar. From an infrastructure perspective, Spoofax and Xtext work similarly, using a global index to store summaries of files and URIs to identify program elements.

EMFText [8] is another framework for developing textual software languages. Like Xtext, it is based on the Eclipse Modeling Framework [23] and relies on metamodels to capture the abstract syntax of a language. While in Xtext this metamodel is generated from a concrete syntax definition, EMFText takes the opposite approach and generates a default syntax definition based on the UML Human-Usable Textual Notation [18] from the metamodel. Language designers can then customize the syntax definition by adding their own grammar rules.

In the default setup, reference resolution needs to be implemented in Java. Only simple cases are supported by default implementations [9]. JastEMF [4] allows to specify the semantics of EMF metamodels using JastAdd RAGs by integrating generated code from JastAdd and EMF.

References

1. Standard ECMA-334 C# language specification, 4th edn (2006)
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. Bravenboer, M., van Dam, A., Olmos, K., Visser, E.: Program transformation with scoped dynamic rewrite rules. Fundamenta Informaticae 69(1-2), 123–178 (2006)
4. Bürger, C., Karol, S., Wende, C., Aßmann, U.: Reference Attribute Grammars for Metamodel Semantics. In: Malloy, B., Staab, S., van den Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 22–41. Springer, Heidelberg (2011)
5. Ekman, T., Hedin, G.: The JastAdd extensible Java compiler. In: Gabriel, R.P., Bacon, D.F., Lopes, C.V. (eds.) Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, pp. 1–18. ACM (2007)
6. Eysholdt, M., Behrens, H.: Xtext: implement your language faster than the quick and dirty way. In: Int. Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, pp. 307–309. ACM (2010)
7. Hedin, G.: An Introductory Tutorial on JastAdd Attribute Grammars. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2009. LNCS, vol. 6491, pp. 166–200. Springer, Heidelberg (2011)
8. 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)
9. Heidenreich, F., Johannes, J., Reimann, J., Seifert, M., Wende, C., Werner, C., Wilke, C., A§mann, U.: Model-driven modernisation of java programs with jamopp. In: Joint Proceedings of MDSM 2011 and SQM 2011. CEUR Workshop Proceedings, pp. 8–11 (March 2011)
10. Hemel, Z., Groenewegen, D.M., Kats, L.C.L., Visser, E.: Static consistency checking of web applications with WebDSL. Journal of Symbolic Computation 46(2), 150–182 (2011)
11. Hemel, Z., Kats, L.C.L., Groenewegen, D.M., Visser, E.: Code generation by model transformation: a case study in transformation modularity. Software and Systems Modeling 9(3), 375–402 (2010)
12. Hemel, Z., Visser, E.: Declaratively programming the mobile web with mobl. In: Fisher, K., Lopes, C.V. (eds.) 2011 Int. Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2011, pp. 695–712. ACM (2011)

13. Kastens, U., Waite, W.M.: Modularity and reusability in attribute grammars. Acta Inf. 31(7), 601–627 (1994)
14. Kats, L.C.L., Visser, E.: The Spoofax language workbench: rules for declarative specification of languages and IDEs. In: Cook, W.R., Clarke, S., Rinard, M.C. (eds.) Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, pp. 444–463. ACM (2010)
15. Kats, L.C.L., Visser, E., Wachsmuth, G.: Pure and declarative syntax definition: paradise lost and regained. In: Cook, W.R., Clarke, S., Rinard, M.C. (eds.) Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, pp. 918–932. ACM (2010)
16. Knuth, D.E.: Semantics of context-free languages. Theory Comput. Syst. 2(2), 127–145 (1968)
17. Meyer, J., Downing, T.: Java Virtual Machine. O Reilly (1997)
18. Object Management Group: Human Usable Textual Notation Specification (2004)
19. Object Management Group: Object Constraint Language, 2.3.1 edn. (2012)
20. Odersky, M.: Defining context-dependent syntax without using contexts. Transactions on Programming Languages and Systems 15(3), 535–562 (1993)
21. Poetzsch-Heffter, A.: Logic-based specification of visibility rules. In: PLILP, pp. 63–74 (1991)
22. Poetzsch-Heffter, A.: Implementing High-Level Identification Specifications. In: Pfahler, P., Kastens, U. (eds.) CC 1992. LNCS, vol. 641, pp. 59–65. Springer, Heidelberg (1992)
23. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: Eclipse Modeling Framework, 2nd edn. Addison-Wesley (2009)
24. Visser, E.: Syntax Definition for Language Prototyping. Ph.D. thesis, University of Amsterdam (September 1997)
25. Visser, E.: Program Transformation with Stratego/XT. In: Lengauer, C., Batory, D., Blum, A., Odersky, M. (eds.) Domain-Specific Program Generation. LNCS, vol. 3016, pp. 216–238. Springer, Heidelberg (2004)
26. 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)

On the Reusable Specification of Non-functional Properties in DSLs

Francisco Durán[1], Steffen Zschaler[2], and Javier Troya[1]

[1] Departamento de Lenguajes y Ciencias de la Computación
Universidad de Málaga
{duran,javiertc}@lcc.uma.es
[2] Department of Informatics
King's College London
szschaler@acm.org

Abstract. Domain-specific languages (DSLs) are an important tool for effective system development. They provide concepts that are close to the problem domain and allow analysis as well as generation of full solution implementations. However, this comes at the cost of having to develop a new language for every new domain. To make their development efficient, we must be able to construct DSLs as much as possible from reusable building blocks. In this paper, we discuss how such building blocks can be constructed for the specification and analysis of a range of non-functional properties, such as, for example, throughput, response time, or reliability properties. We assume DSL semantics to be provided through a set of transformation rules, which enables a range of analyses based on model checking. We demonstrate new concepts for defining language modules for the specification of non-functional properties, show how these can be integrated with base DSL specifications, and provide a number of syntactic conditions that we prove maintain the semantics of the base DSL even in the presence of non-functional–property specifications.

1 Introduction

Domain-specific languages (DSLs) are an important tool for reaping the proposed benefits of model-driven engineering [1]. DSLs are languages based on concepts closer to the problem domain than the technical solution. They are, therefore, a good way to allow domain-experts, who may lack programming skills, to construct or participate in constructing substantial parts of new systems. In addition, because much more knowledge of the domain is available when interpreting statements in a DSL, it is possible to provide much more extensive code generation; this can enable complete generation of running systems from a relatively simple DSL-based model [2]. However, for DSLs to be effective, they may need to be implemented for very narrow domains [1], which implies that a large number of DSLs needs to be implemented. This requires highly efficient techniques for developing new DSLs, ideally based on an ability to reuse and compose partial languages for new domains.

In the design of software systems, many researchers distinguish between functional and non-functional properties (NFPs)—also sometimes referred to as extra-functional properties or quality of service. While functional properties are constraints on *what* the software system does, NFPs are constraints on *how* it does it—for example, how much resources are used or how long it takes to process an individual request. NFPs are important for the overall quality of a system, so they clearly need to be taken into account throughout development. We need to be able to predict and analyse NFPs from an early stage of development, so as to avoid costly re-design or re-implementation at a later stage. When developing systems based on DSLs, these DSLs, consequently, need to include an ability to express and analyse relevant NFPs. However, the analysis of NFPs is difficult and usually requires substantial specialist expertise. Integrating an ability to specify NFPs into DSLs can substantially increase the effort required to build a DSL. In this paper, we propose a technique for allowing NFP specification to be encapsulated into reusable DSL components. This way, the burden of specifying the NFPs of DSLs is drastically reduced, and specialist expertise is mainly required when the language component is constructed. Developing new DSLs capable of specifying particular NFPs in the context of a particular domain then becomes a matter of weaving in the NFP's language component.

The *e-Motions* language and system allows the definition of visual DSLs and their semantics through in-place model-transformation rules, providing support for their analysis through simulation or model checking in Maude [3]. In [4], Troya, Rivera, and Vallecillo build on the ideas of the *e-Motions* framework [5,6] to keep track of specific NFPs by adding auxiliary objects to DSLs. However, their approach still requires the NFP specification and analysis component to be redefined from scratch for every new DSL. In this paper we build on their work, but aim to modularise the NFP part into its own language component. To do so, we take inspiration from the work in [7] where Zschaler introduced the notion of *context models* to provide an interface between TLA$^+$ specifications of non-functional and functional properties. We will use parametrisation over meta-models to achieve a similar effect for our language components. Specifically, we present a formal framework for such language components, syntactic conditions for their consistency and proofs of these conditions. We also present a basic prototype implementing these ideas in the context of *e-Motions*. However, a full integration is not in the scope of this current paper.

While our prototype and original motivation are for the case of *e-Motions*, both our approach and formal framework are more general. They can be applied for any DSL specification whose semantics are based on model transformations. Moreover, while our work is clearly motivated from the need of modularising NFP specifications, the formal framework covers arbitrary conservative extensions of such DSLs, guaranteeing them to be *spectative* in the sense of [8].

The remainder of this paper is structured as follows: In Section 2, we discuss a detailed motivating example to explain the vision of what we would like to achieve. Section 3 then presents a formalisation of these ideas together with consistency conditions and sketches of their proofs (see [9] for additional details on this). Section 4 briefly discusses our initial prototype. Finally, Section 5 discusses related work followed by conclusions and an outlook to future work in Section 6.

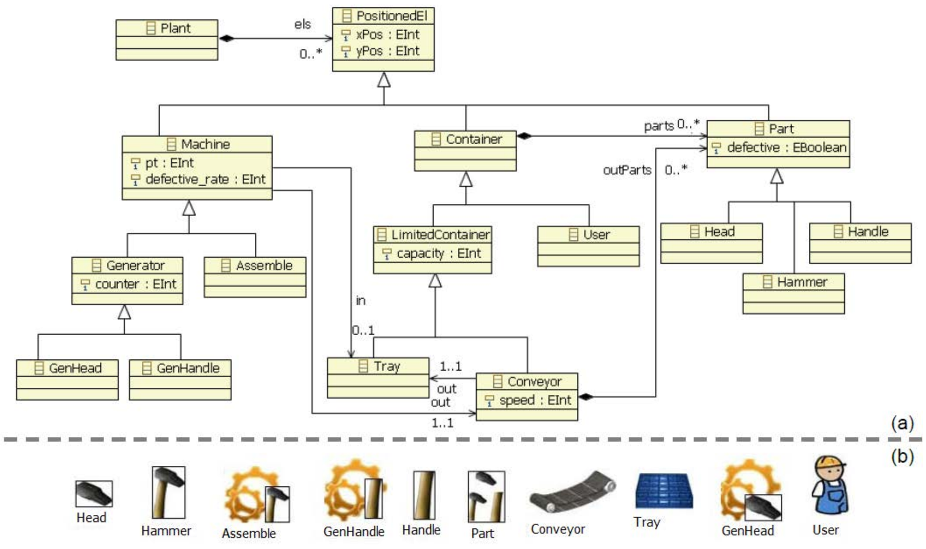


Fig. 1. Production line (a) metamodel and (b) concrete syntax (from [4])

2 Motivating Example

In this section, we present an example of what we want to achieve. This is based on work presented by Troya, Rivera, and Vallecillo in [4]. Their work defines DSLs from two parts: a meta-model of the language concepts and a set of transformation rules to specify the behavioural semantics of the DSL.

Figure 1(a) shows the metamodel of a DSL for specifying production-line systems, for producing hammers out of hammer heads and handles, which are generated in respective machines, and transported along the production line via conveyors and trays. As usual in MDE-based DSLs, this metamodel defines all the concepts of the language and their interconnections; in short, it provides the language's *abstract* syntax. In addition, a *concrete* syntax is provided. In the case of our example, this is sufficiently well defined by providing icons for each concept (see Figure 1(b)); connections between concepts are indicated through arrows connecting the corresponding icons.

Instances of this DSL are intended as token models [10]. That is, they describe a specific situation and not the set of all possible situations (as is the case, e.g., for class diagrams). The behavioural semantics of the DSL can, therefore, be given by specifying how models can evolve; that is, what changes can occur in a particular situation. This is specified through a set of model transformation rules. Figure 2 shows an example of such a rule. The rule consists of a left-hand side matching a situation before the execution of the rule and a right-hand side showing the result of applying the rule.[1] Specifically, this rule shows how a new hammer is assembled: a hammer generator a

[1] There are some other parts to the rule, but they are not relevant for our current discussion. For a more detailed discussion, please refer to material on *e-Motions* [5,6].

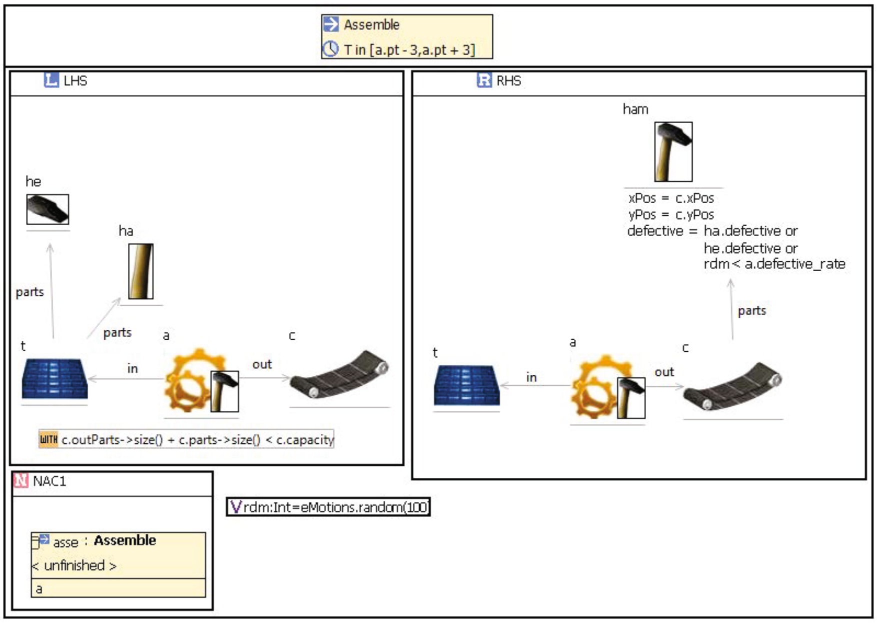

Assemble
T in [a.pt - 3,a.pt + 3]
LHS
RHS
he
ha
parts
parts
t
a
c
in
out
WITH c.outParts->size() + c.parts->size() < c.capacity
ham
xPos = c.xPos
yPos = c.yPos
defective = ha.defective or he.defective or rdm< a.defective_rate
NAC1
asse : Assemble
< unfinished >
a
V rdm:Int=eMotions.random(100)

Fig. 2. `Assemble` rule indicating how a new hammer is assembled (from [4])

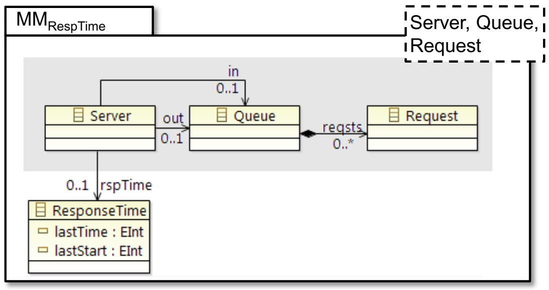
has an incoming tray of parts and is connected to an outgoing conveyor belt. Whenever there is a handle and a head available, and there is space in the conveyor for at least one part (specified by an OCL constraint in the left-hand side of the rule), the hammer generator can assemble them into a hammer. The new hammer is added to the `parts` set of the outgoing conveyor belt. The complete semantics of our production-line DSL is constructed from a number of such rules covering all kinds of atomic steps that can occur.[2]

For production line systems, we are interested in a number of non-functional properties. For example, we would like to assess the throughput of the product line or how long it takes for a hammer to be produced.[3] We can achieve this by extending our DSL specification with observers [4]. Different from [4], here we suggest defining specification languages for observers entirely separately from any specific DSL. We will use the same mechanisms we used for defining the product line DSL to define a DSL that enables us to specify throughput or production time of systems.

Figure 3(a) shows the meta-model for a DSL for specifying production time. Two things should be noted about this meta-model:

[2] The complete specification of the Production Line example can be found at `http://atenea.lcc.uma.es/E-motions/PLSExample`.

[3] We use this property as an example here. Other properties can be defined easily in a similar vein as shown in [4] and on `http://atenea.lcc.uma.es/index.php/Main_Page/Resources/E-motions/PLSObExample`.



(a) Meta-model.

(b) Concrete syntax.

Fig. 3. Meta-model and concrete syntax for response time observer

1. It defines no concept production time. Instead, it defines something called response time, which is a more generic concept. Production time is really only meaningful in the context of production systems. However, the general concept of response time covers this sufficiently well.
2. It is a parametric model (i.e., a model template). The concepts of `Server`, `Queue`, and `Request` and their interconnections are parameters of the meta-model, and they are shaded in grey for illustration purposes. We use them to describe in which situations response time can be specified, but these concepts will need to be mapped to concrete concepts in a specific DSL.

Figure 3(b) shows the concrete syntax for the response time observer object. Whenever that observer appears in a behavioural rule, it will be represented by that graphical symbol.

Figure 4 shows an example transformation rule defining the semantics of the response time observer. This states that if there is a server with an `in` queue and an `out` queue and there initially are some requests (at least one) in the `in` queue, and the `out` queue contains some requests after rule execution, the last response time should be recorded to have been equal to the time it took the rule to execute. Similar rules need to be written to capture other situations in which response time needs to be measured, for example, where a request stays at a server for some time, or where a server does not have an explicit `in` or `out` queue.

Note that the rule in Figure 4 looks different from the rule shown in Figure 2. This is because the rule is actually a rule transformation, while Figure 2 is a transformation rule. The upper part of Figure 4 (shaded in grey for illustration purposes) is a pattern or query describing transformation rules that need to be extended to include response-time accounting. The lower part describes the extensions that are required. So, in addition to reading Figure 4 as a 'normal' transformation rule (as we have done in the previous paragraph), we can also read it as a rule transformation stating: "Find all rules that match the shaded pattern and add `ResponseTime` objects to their left and right-hand sides as described." In effect, observer models become higher-order transformations [11].

As the rules in observer models are rule transformations, we can allow some additional concepts to be expressed. For example, Figure 4 uses multiplicities to express that

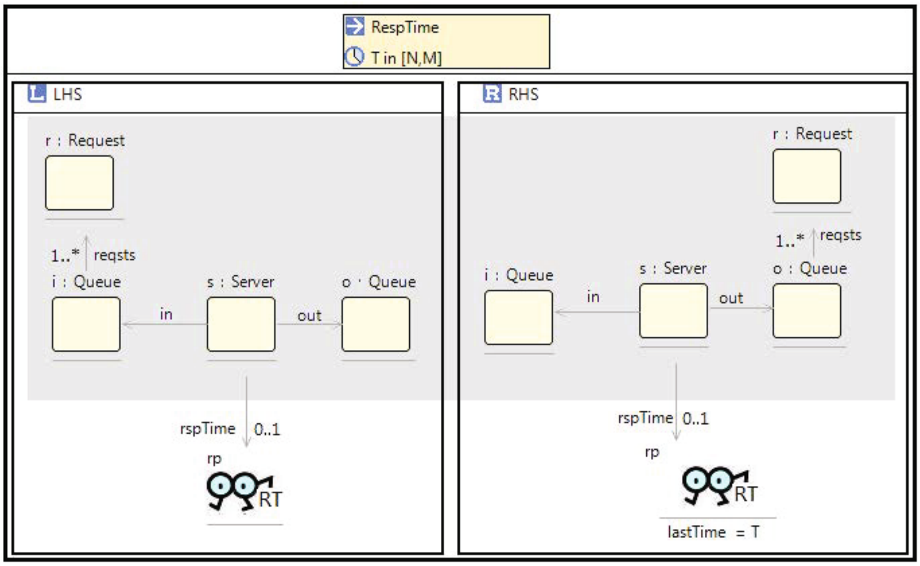


Fig. 4. Sample response time rule

there may be an arbitrary number of requests (but at least one) associated with a queue. This is not allowed in 'normal' transformation rules (there we need to explicitly show each instance). However, using multiplicities allows expressing patterns to be matched against transformation rules—a match is given by any rule that has the indicated number of instances in its left- or right-hand side.

To use our response-time language to allow specification of production time of hammers in our production-line DSL, we need to weave the two languages together. For this, we need to provide a binding from the parameters of the response-time meta-model (Figure 3(a)) to concepts in the production-line meta-model (Figure 1(a)). Specifically, we bind:

- **`Server` to `Assemble`** as we are interested in measuring response time of this particular machine;
- **`Queue` to `LimitedContainer`** as the `Assemble` machine is to be connected to an arbitrary `LimitedContainer` for queuing incoming and outgoing parts;
- **`Request` to `Part`** as `Assemble` only does something when there are `Parts` to be processed; and
- **Associations:**
  - The `in` and `out` associations from `Server` to `Queue` are bound to the corresponding `in` and `out` associations from `Machine` to `Tray` and `Conveyor`, respectively; and
  - The association from `Queue` to `Request` is bound to the association from `Container` to `Part`.

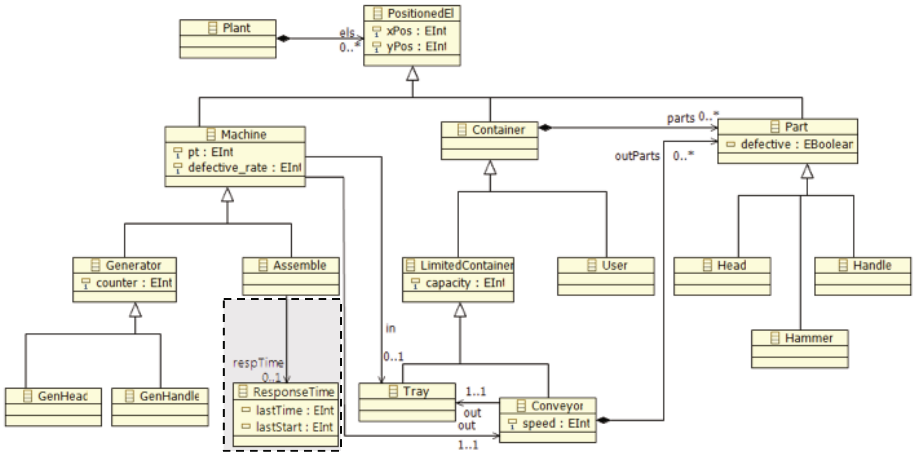


Fig. 5. Woven meta-model for measuring production time of the hammer assembler (highlighting added for illustration purposes)

Weaving the meta-models according to this binding produces the meta-model in Figure 5. The weaving process has added the `ResponseTime` concept to the meta-model. Notice that the weaving process also ensures that only sensible woven meta-models can be produced: for a given binding of parameters, there needs to be a match between the constraints expressed in the observer meta-model and the DSL meta-model. We will discuss this issue in more formal detail in Section 3.

The binding also enables us to execute the rule transformations specified in the observer language. For example, the rule in Figure 2 matches the pattern in Figure 4, given this binding: In the left-hand side, there is a `Server` (`Assemble`) with an in-`Queue` (`Tray`) that holds two `Requests` (`Handle` and `Head`) and an out-`Queue` (`Conveyor`). In the right-hand side, there is a `Server` (`Assemble`) with an in-`Queue` (`Tray`) and an out-`Queue` (`Conveyor`) that holds one `Request` (`Hammer`). Consequently, we can apply the rule transformation from Figure 4, which produces the rule shown in Figure 6. This rule is equivalent to what would have been written manually.

Clearly, such a separation of concerns between a specification of the base DSL and specifications of languages for non-functional properties is desirable. In the next section, we discuss the formal framework required for this and how we can distinguish safe bindings from unsafe ones.

3 Formal Framework

Graph transformation [12] is a formal, graphical and natural way of expressing graph manipulation based on rules. In graph-based modelling (and meta-modelling), graphs are used to define the static structures, such as class and object ones, which represent

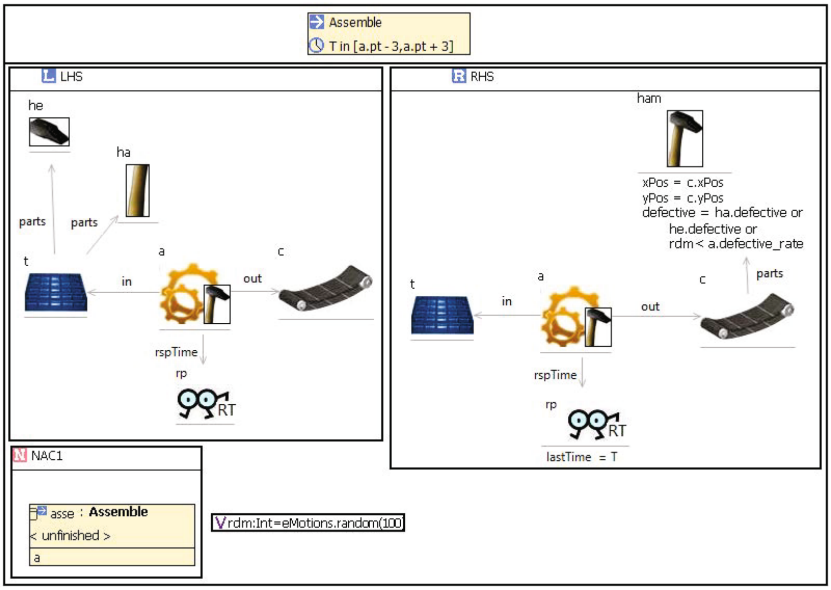


Fig. 6. Result of weaving Figure 2 and Figure 4

visual alphabets and sentences over them. We formalise our approach using the typed graph transformation approach, specifically the Double Pushout (DPO) algebraic approach, with positive and negative application conditions [13]. Our graphs are, in particular, typed attributed graphs [14]. We however carry on our formalisation for weak adhesive high-level replacement (HLR) categories (see [15]).

The concepts of adhesive and (weak) adhesive HLR categories abstract the foundations of a general class of models, and comes together with a collection of general semantic techniques. Thus, e.g., given proofs for adhesive HLR categories of general results such as the Local Church-Rosser, or the Parallelism and Concurrency Theorem, they are automatically valid for any category which is proved an adhesive HLR category. This framework has been a break-through for the DPO approach of algebraic graph transformation, for which most main results can be proven in these categorical frameworks, and instantiated to any HLR system. One of these cases is the one of interest to us: the category of typed attributed graphs was proven to be an adhesive HLR category in [14].

In this section, we present a formal framework of what it means to define specification languages for non-functional properties separately to 'normal' DSLs, and in a way that can be reused across such DSLs. To this end, we will first abstract away from the concrete representation of languages and models in *e-Motions* [5, 6] that we have used in Section 2. Instead, we will formally represent the key elements of which such languages and models consist and the functions which are used to manipulate them.

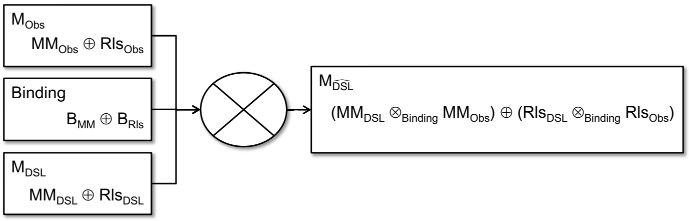


Fig. 7. Architecture of the formal framework

Figure 7 provides a graphical overview of the formal framework we are proposing. It can be seen that this consists of five parts:

1. M_{DSL}: The specification of a DSL (without any notion of non-functional properties);
2. M_{Obs}: The specification of a language for modelling non-functional properties of interest;
3. *Binding*: An artefact expressing how the parameters of M_{Obs} should be instantiated with concepts from M_{DSL} in order to weave the two languages;
4. $\otimes$: A function that performs the actual weaving; and
5. $M_{\widehat{DSL}}$: A DSL that combines the specification of some functionality (as per M_{DSL}) and some non-functional properties (as per M_{Obs}).

3.1 The Models Involved and Their Relationships

Following the algebraic graph transformation approach, a DSL can be seen as a typed graph grammar. A *typed graph transformation system* $GTS = (TG, P)$ consists of a type graph TG and a set of typed graph productions P. A *typed graph grammar* $GG = (GTS, S)$ consists of a typed graph transformation system GTS and a typed start graph S. A language is then defined by the set of graphs reachable from S using the transformation rules P.

Definition 1 (DSL). The specification M_X of a DSL X is given by a metamodel MM_X, representing the structural concepts of the language, and a set of transformation rules Rls_X, defining its behavioural semantics. ■

A metamodel is just a type graph, and a transformation rule associated to it is a graph production typed over the type graph provided by such metamodel.

The languages M_{DSL} and $M_{\widehat{DSL}}$ are DSL specifications. M_{Obs} is, essentially, also a normal DSL specification. Notice that we assume a single observer model M_{Obs} for each non-functional property. If we needed several of these properties, we could consider M_{Obs} to be the combination of the specifications of these non-functional properties, or we could iterate the process by instantiating $M_{\widehat{DSL}}$ once obtained with a second observers model $M_{Obs'}$ producing a resulting specification $M_{\widehat{\widehat{DSL}}}$, which could again be instantiated by another observers model $M_{Obs''}$, etc.

Although M_{Obs} is, essentially, a normal DSL specification, it is however parametrised and has a dual interpretation:

1. As a specification language for non-functional properties; and
2. As a higher-order transformation specification [11] expressing how DSLs need to be modified to enable the specification of a particular non-functional property.

Specifically, in the observer model M_{Obs}, we distinguish a parameter sub-model M_{Par} which specifies just enough information about real systems to define the semantics of the non-functional property, but not more. This parameter sub-model M_{Par} is a sub-model of M_{Obs} in the sense that MM_{Par} is a subgraph of MM_{Obs}, and each transformation rule in Rls_{Par} is a sub-rule of a rule in Rls_{Obs}.

This notion of sub-model and that of binding are captured by the general notion of DSL morphism, which can be defined as follows.

Definition 2 (DSL Morphism). Given DSL specifications M_A and M_B, a *DSL morphism* $M_A \to M_B$ is a pair (δ, ω) where δ is a meta-model morphism $MM_A \to MM_B$, that is, a mapping (or function) in which each class, attribute and association in the meta-model MM_A is mapped, respectively, to a class, attribute and association in MM_B such that

1. class maps in δ must be compatible with the inheritance relation, that is, if class C inherits from class D in MM_A, then $\delta(C)$ must inherit from class $\delta(D)$ in MM_B;
2. class maps and association maps in δ must be consistent, that is, the images of the extremes of an association K in MM_A must lead to classes associated by the association $\delta(K)$;
3. attribute maps and class maps in δ must be consistent, that is, given an attribute a of a class C, its image $\delta(a)$ must be an attribute of the class $\delta(C)$;

and where ω is a set of transformation rules such that for each rule $r_1 \in Rls_A$ there is a transformation rule $\sigma : r_1 \to r_2$ for some $r_2 \in Rls_B$.

Rules r_1 and r_2 in a rule map $\sigma : r_1 \to r_2$ must have the same time constraints (ongoing or atomic, same duration, softness, periodicity, etc.). ■

Definition 2 could be relaxed in several ways, e.g., condition 2 could be relaxed to allow superclasses of the images of the extremes of an association K to be related by its image $\delta(K)$; similarly for condition 3, since it could be that the attribute $\delta(a)$ is an attribute inherited from some superclass of $\delta(C)$. However, we leave these relaxations, and a study of their effects on the formal framework presented, to future work.

Note also that the role of the parameter model M_{Par} is useful for establishing the way in which the DSL's metamodel and rules are to be modified. The binding DSL morphism is key for it, since it says how the transformations are to be applied. An inclusion morphism $(\iota, \omega) : M_{Par} \hookrightarrow M_{Obs}$ can be seen as a transformation rule, with the binding indicating how such rule must be applied to modify the DSL system being instrumentalised. Specifically, the metamodel morphism $\iota : MM_{Par} \hookrightarrow MM_{Obs}$ indicates how the metamodel MM_{DSL} must be extended, and the family of higher-order transformations (HOT) rules $\sigma_i : r_{1,i} \hookrightarrow r_{2,i}$ in ω indicate how the rules in Rls_{DSL} must be modified. Notice that these rule transformations match those presented in Section 2:

A transformation rule like the one in Figure 4 is interpreted as a rule transformation where the parameter part (the shadowed sub-rule) is its left-hand side, and the entire rule its right-hand side.

Since M_{Par} is not intended for DSL specification, but only as constraints in the different transformations involved, i.e., for matching, we can enrich its expressiveness, for example, by allowing associations with multiplicity $1..n$ as in Figure 4. The left-hand side of this HOT rule can in this way match rules whose queues have 1, 2, or any number of request objects associated. Notice that this rule is woven with the `Assemble` rule in Figure 2, which has a head and a handle associated to its `in` tray. It could be seen as the inclusion and the binding happening on specific submodels, those defined by the concrete match. We do not consider this additional flexibility in the formalisation below. Notice however that the matches induce corresponding rules for which the formalisation does work.

3.2 Model Weaving

M_{Obs} and M_{DSL} are woven to produce a combined DSL, $M_{\widehat{DSL}}$. This weaving is encoded as a function $\otimes : M_{Obs} \times M_{DSL} \times \mathit{Binding} \rightarrow M_{\widehat{DSL}}$, which is graphically depicted in Figure 7. As indicated above, *Binding* is a DSL morphism, which expresses how the parameters of M_{Obs} should be instantiated with concepts from M_{DSL} in order to weave the two languages. Intuitively, $\otimes$ works in two stages:

1. *Binding stage.* In this stage, *Binding* is used to produce an instantiated version of M_{Obs} (and its parameter sub-model M_{Par}), $M_{Obs'} = (MM_{Obs'}, Rls_{Obs'})$, which is the result of replacing each parameter element $p \in MM_{Par}$ by a corresponding element from MM_{DSL} in M_{Obs} in accordance with *Binding*. The resulting $MM_{Obs'}$ is used to construct the output meta-model $MM_{\widehat{DSL}} = MM_{DSL} \uplus_{\mathit{Binding}} MM_{Obs'}$. The operator $\uplus_{\mathit{Binding}}$ stands for disjoint union, where elements related by *Binding* are identified and the rest are distinguished. Each rule $\sigma'_i : r'_{1,i} \hookrightarrow r'_{2,i}$ in $Rls_{Obs'}$ is the result of a similar replacement of a rule $\sigma_i : r_{1,i} \hookrightarrow r_{2,i}$ in Rls_{Obs}.
2. *Transformation stage.* In this stage, $Rls_{Obs'}$ is used to transform Rls_{DSL}. For each inclusion morphism $\sigma'_i : r'_{1,i} \hookrightarrow r'_{2,i}$, the corresponding rule $r \in Rls_{DSL}$ is identified and transformed according to σ'_i. This step produces $Rls_{\widehat{DSL}}$.

Note that the rules and appropriate matches to apply the HOT rules should be guided by *Binding*. Although there might be cases in which we can systematically apply the HOT rules on the rules in Rls_{DSL}, in general this is not the case. Note that each HOT rule defined by a rule in Rls_{Obs} may be applicable to different rules in Rls_{DSL}, and for each of them there might be more than one match. Although there might be many cases in which a partial binding might be enough, we however assume that the binding is complete.

The semantics of the weaving operation, informally described above, is provided by the pushout of DSL morphisms $M_{Par} \hookrightarrow M_{Obs}$ and $M_{Par} \hookrightarrow M_{DSL}$ in the category **DSL** of DSL specifications and DSL morphisms. Although the details of the pushout construction can be found in [9], we sketch it here. Given DSL morphisms $(\delta_1, \omega_1) : A \rightarrow B$ and $(\delta_2, \omega_2) : A \rightarrow C$, with DSLs $X = (MM_X, Rls_X)$ for $X = A$,

B, C, the pushout object of (δ_1, ω_1) and (δ_2, ω_2) is, up to isomorphism, the DSL specification $D = (MM_D, Rls_D)$, together with DSL morphisms $(\delta'_1, \omega'_1) : C \to D$ and $(\delta'_2, \omega'_2) : B \to D$, where $\delta'_1 : MM_C \to MM_D$ and $\delta'_2 : MM_B \to MM_D$ are the pushout of $\delta_1 : MM_A \to MM_B$ and $\delta_2 : MM_A \to MM_C$, and Rls_D is the disjoint union of those rules in Rls_B and Rls_C that are not targets of rules in Rls_A, and the set of rules $\tilde{r}_A$ resulting from the amalgamation of rule morphisms $r_1^A \to r_2^B$ in ω_1 and $r_1^A \to r_2^C$ in ω_2 for all rules r_A in Rls_A. The rule injections from rules in Rls_B and Rls_C that are not targets of rules in Rls_A and the amalgamation-induced rule morphisms $r_2^B \to \tilde{r}_A$ and $r_2^C \to \tilde{r}_A$ characterise the family of rule transformations ω'_2 (resp., ω'_1).

$$\begin{tikzcd}
A \arrow[r, "(\delta_2{,}\omega_2)"] \arrow[d, "(\delta_1{,}\omega_1)"'] \arrow[dr, phantom, "po"] & C \arrow[d, dotted, "(\delta'_1{,}\omega'_1)"] \\
B \arrow[r, dotted, "(\delta'_2{,}\omega'_2)"'] & D
\end{tikzcd}$$

3.3 Semantic Consistency

The construction of *Binding* as a binding morphism $(\delta, \omega) : M_{Par} \to M_{DSL}$ ensures basic syntactic consistency between the observer model and the DSL model to be woven. However, it does not ensure semantic consistency. We could, for instance, specify a deadlock behaviour in the observers, changing the behaviour of the system DSL after the weaving. While it will likely not be possible to provide sufficient conditions for binding validity (see [7, pp. 9–11] for a discussion of the reasons), we should be able to provide at least some necessary conditions. As a minimum, we require that the extension be *conservative*, not changing the very nature and behaviour of the original DSL, namely:

$$M_{\widehat{DSL}}|_{MM_{DSL}} \cong M_{DSL} \tag{1}$$

We use $M_{\widehat{DSL}}|_{MM_{DSL}}$ to denote the language specification that results from removing any non-MM_{DSL} elements from the meta-model and rule set of $M_{\widehat{DSL}}$. Essentially, we mean to say that adding observers does not change the basic structure and behaviour defined by M_{DSL}. This is the typical condition one would expect in this kind of situations, and has been established in many different contexts before—perhaps first in [16].

Condition (1) is too hard to check directly, and therefore we need simpler, if possible syntactic, conditions implying it. If we break (1) down into conditions for the meta-model and the rule component of $M_{\widehat{DSL}}$, we get the following two conditions:

$$MM_{\widehat{DSL}}|_{MM_{DSL}} \cong MM_{DSL} \tag{2}$$

$$\Pi\left(Rls_{\widehat{DSL}}\right)|_{MM_{DSL}} \cong_{\textit{stuttering}} \Pi\left(Rls_{DSL}\right) \tag{3}$$

We have again used the restriction operator $|_{MM_{DSL}}$, although in this case applied both to meta-models and traces, but with the same effect, namely, removing any non-MM_{DSL} elements (both from the meta-model $MM_{\widehat{DSL}}$ and rules in $Rls_{\widehat{DSL}}$). $\Pi\left(Rls_X\right)$ denotes the set of behaviours (possible executions, or traces) modelled by the transformation rules of a DSL X; that is, the set of all (potentially infinite) traces of model states as rewritten by these transformation rules. For traces, we use $\cong_{\mathit{stuttering}}$ to explicitly state that the traces in $\Pi\left(Rls_{\widehat{DSL}}\right)|_{MM_{DSL}}$ may in fact have more steps than those in $\Pi\left(Rls_{DSL}\right)$, but because of the restriction down to MM_{DSL}, these remaining extra steps should all be identities (stuttering steps).

The above conditions (2) and (3) could only be checked by performing the weaving and analysing the result. However, both the weaving and the checking on the resulting specification are potentially expensive. Instead, we want to check the safety of an observer model and a binding morphism simply by looking at the models themselves without having to perform the weave. We are, therefore, looking for conditions on M_{Obs} and the binding morphism, if possible syntactic, so that they can be automated, or at least, performed once and for all. We in fact claim that analysing M_{Obs}, and simple syntactic conditions on the parameter inclusions $M_{Par} \hookrightarrow M_{Obs}$ and on the instantiating binding $(\delta, \omega) : M_{Par} \to M_{DSL}$ are sufficient to imply the satisfaction of (2) and (3).

We first discuss conditions for the structural part encoded in MM_{Obs}, and then discuss the behavioural semantics.

Structural Conditions. In any adhesive category, the pushout of a monomorphism along any map is a monomorphism [17, Proposition 2.1]. Therefore, since $MM_{Par} \to MM_{Obs}$ is a monomorphism, the induced morphism $MM_{DSL} \to MM_{\widehat{DSL}}$ is also a monomorphism. Notice that in addition to new classifiers, attributes and associations involving observers, new attributes for classes in MM_{Par} and new associations between classes in MM_{Par} may be introduced in MM_{Obs}. This might be convenient for modelling some NFPs and does not cause problems from a semantic point of view.

Behavioural Conditions. We need to ensure that adding observers to a DSL specification does not prevent any behaviour of any instance of that DSL that was previously allowed, and, moreover, that no new behaviours are added.

It can be shown that we can have (3) by imposing a similar condition on the morphism $M_{Par} \to M_{Obs}$. More precisely, it can be shown that

$$\Pi\left(Rls_{Obs}\right)|_{MM_{Par}} \equiv_{\mathit{stuttering}} \Pi\left(Rls_{Par}\right) \tag{4}$$

implies

$$\Pi\left(Rls_{\widehat{DSL}}\right)|_{MM_{DSL}} \equiv_{\mathit{stuttering}} \Pi\left(Rls_{DSL}\right)$$

We still need methods for checking (4). The formalisation of the construction in [9] suggests some ideas in this direction, but we leave it as future work.

4 A Prototypical Implementation

For the implementation of our prototype we have used ATL [18], a hybrid model transformation domain specific language that contains a mixture of declarative and imperative constructs. ATL transformations are unidirectional, operating on read-only source

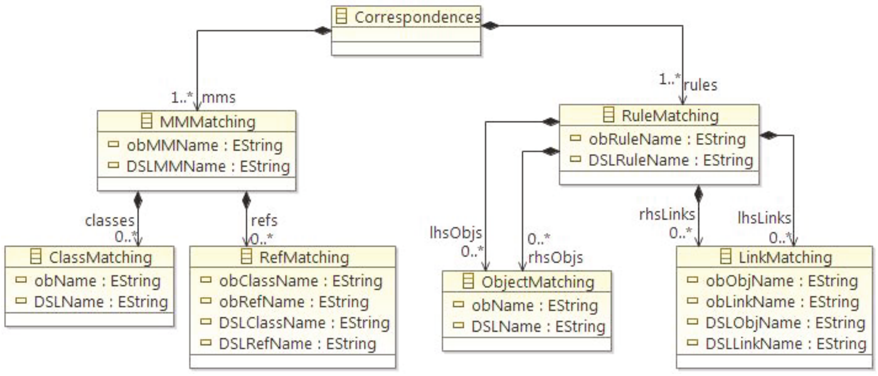


Fig. 8. Correspondences meta-model

models and producing write-only target models. During the execution of a transformation, source models may be navigated but changes are not allowed. Target models cannot be navigated.

Following the proposal presented in Figure 7, we have split the binding process in two ATL transformations: one for weaving the meta-models, MM_{DSL} and MM_{Obs}, and another one for weaving the behavioural rules, Rls_{DSL} and Rls_{Obs}. In our example, the former produces the meta-model shown in Figure 5, while the latter produces the woven rule depicted in Figure 6 (plus the remaining rules in Rls_{DSL}). For the remainder of this section, let us clarify that by *binding* we mean the relations established between two models. As for *correspondence(s)* and *matching(s)*, we use them indistinctly when we refer to one or more specific relations among the concepts in both models (either DSL and observer meta-models or DSL and observer behavioural rules).

The binding between M_{DSL} and M_{Obs} is given by a model that conforms to the correspondences meta-model shown in Figure 8. Thus, both bindings, between meta-models and between behavioural rules, are given in the same model. For the binding between meta-models (Figures 1(a) and 3(a) in our example), we have the classes `MMMatching`, `ClassMatching` and `RefMatching` that specify it. We will have one object of type `MMMatching` for each pair of meta-models that we want to weave. In our example, we have one object of this type, and its attributes contain the names of the meta-models to weave. Objects of type `MMMatching` contain as many `classes` (objects of type `ClassMatching`) as there are correspondences between classes in both meta-models. Each object of type `ClassMatching` stores the names of the classes in both meta-models that correspond. We have three objects of this type, as described in Section 2. Regarding the objects of type `RefMatching`, contained in the `refs` reference from `MMMatching`, they store the matchings between references in both meta-models. Attributes `obClassName` and `DSLClassName` keep the names of the source classes, while `obRefName` and `DSLRefName` contain the names of the references. Once again, and as described in Section 2, there are three objects of this type in our example.

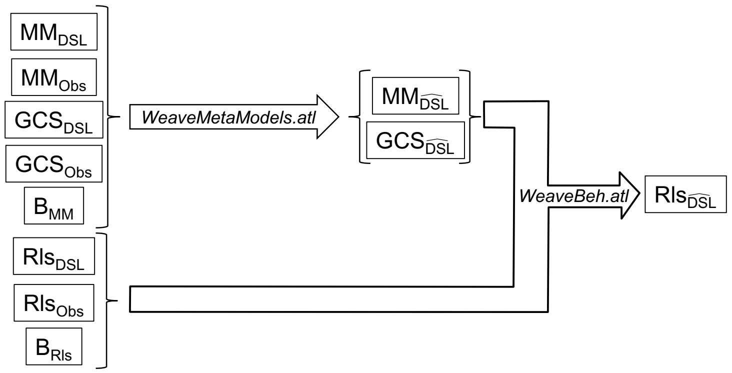


Fig. 9. Transformations schema

Regarding the binding between rules (Rls_{DSL} and Rls_{Obs}), there is an object of type RuleMatching for each pair of rules to weave, so in our example there is only one. It contains the names of both rules (Assemble and RespTime). Objects of types ObjectMatching and LinkMatching contain the correspondences between objects and links, respectively, in the rules. Concretely, our correspondence models differentiate between the bindings established between left- and right-hand side in rules, as we describe later. In our behavioural rules described within *e-Motions*, which conform to the Behavior meta-model (presented in [5]), the objects representing instances of classes are of type Object and they are identified by their id attribute, and the links between them are of type Link, identified by their name, input and output objects. Similar to the binding between meta-models, objects of type ObjectMatching contain the identifier of the objects matching, and instances of LinkMatching store information about matchings between links (they store the identifier of the source classes of the links as well as the name of the links). The correspondences between rules Assemble and RespTime are those described at the end of Section 2.

A detailed documentation of the weaving process, performed by two ATL transformations, is available in [19]. Here, we limit ourselves to a high-level overview of the transformation architecture. As shown in Figure 9, we have split the overall weaving function into two model transformations, one for weaving the metamodels and the other for weaving the rules. Apart from the models already presented in Figure 7, GCS models (graphical concrete syntax) also take part in the transformations. They store information about the concrete syntax of DSL and observer models. Both transformations work in two stages to ensure the original DSL semantics are preserved. First, they copy the original DSL model into the output model. Second, any additions from the observer model are performed according to the binding information from the weaving model.

The first transformation, named WeaveMetaModels.atl, deals with the weave of meta-models and GCS models. In the first step, the transformation copies both models from the DSL into the output models. Next, it decorates the models created with the concepts from the observer meta-model and GCS model. Regarding the output meta-model, it adds the classes, references and attributes representing observers. In our example this means inserting the `ResponseTime` class, adding its attributes and establishing the `respTime` reference among `Assemble` and `ResponseTime` classes. As for the output `GCS` file, it means adding all the necessary data regarding the concrete syntax of the `ResponseTime` class.

Between its inputs, the second transformation, `WeaveBeh.atl`, takes the models produced by the first transformation. It performs in a similar way. The first step is to copy all those rules from Rls_{DSL} in the output model with the behavioural rules. Next, those rules having correspondences with rules in Rls_{Obs} are decorated with observer objects, links and attributes.

5 Related Work

We discuss related work in two areas: modelling of non-functional properties and modular language definition.

5.1 Modelling of Non-Functional Properties

Modelling and analysis of non-functional properties has been an active research area for a substantial amount of time already. Our work is related to other work aiming to support specification of a wide range of non-functional properties—for example, languages such as QML [20], CQML [21], CQML$^+$ [22], or SLAng [23]. These languages take a meta-modelling approach to the specification of non-functional properties in a two-step process: In a first step, modellers specify *non-functional characteristics*—for example, performance. These characteristics are then used in a second step to express constraints over application models; that is, non-functional properties. This is similar to our approach: An observer model M_{Obs} effectively defines a non-functional characteristic. A woven DSL $M_{\widehat{DSL}}$ can then be used to model non-functional properties. The approaches mentioned above differ in their amount of formal rigor (increasing from QML to CQML$^+$ and SLAng) and the type of systems they support (all except SLAng are aimed at component-based systems; SLAng is meant for service-based systems). They typically do not provide extensive support for analysis of the models created.

More formal renderings of these concepts can be found in [16] and [7]. The former presents a formal encoding of real-time properties using so-called history-determined variables, which are then used to model non-functional characteristics that depend on time. [7] extends this to a formal framework for specifying non-functional properties of component-based systems. While these approaches can potentially enable proofs of non-functional properties, it is not clear how well they are suited to predictive analysis of system properties—for example through simulation.

The approach by Troya and Vallecillo [4] aims to address this issue by providing a specification based on observers and transformations. This enables predictive analysis through simulation based on an encoding in *e-Motions* [5,6], which is translated into Maude. However, their approach requires the details of a non-functional characteristic to be redefined completely for each DSL. Our proposal is an extension of this work using ideas from [7, 16] to separate the specification of non-functional characteristics from that of the functional behavioural semantics of a DSL.

5.2 Modular Languages, Models, and Transformations

We propose to weave two language definitions: One language enables the (abstract) specification of a set of non-functional properties while the second language focuses entirely on specifying relevant behaviours in a particular domain. Below we briefly review some related work in the general area of modular definition of languages, models, and transformations. We discuss selected related work in three areas:

1. Modular definition of languages;
2. Modular definition of models; and
3. Modular definition of model transformations.

Modular Definition of Languages. There is a large body of work on modularly defining computer languages. Most of this work (e.g., [24–26]) deals with textual languages and in particular with issues of composing context-free grammars. While the general idea of language composition is relevant for our work, this specific strand of research is perhaps less related and will, therefore, not be discussed in more detail.

For languages based on meta-modelling, there is much less research on language composition. Much of the work on model composition (see next sub-section) is of course of relevance as meta-models are models themselves. Christian Wende's work on role-based language composition [27] is an approach that specifically addresses the modularisation of meta-models. For a language module, Wende's work allows the definition of a composition interface by allowing language designers to use two types of meta-model concepts: meta-classes and meta-roles. Meta-classes are used as in normal meta-modelling to express the core meta-model concepts. Meta-roles are like meta-classes, however they actually represent concepts to be provided by another language—including definitions of operations and attributes, which are left abstract in the meta-role. Meta-roles are, thus, similar to our use of meta-model parameters in MM_{Obs}. However, Wende's work uses meta-class operations to provide an operational view on language semantics, while we use model transformations to encode language semantics.

Modular Modelling. Our notation for expressing parametrised meta-models is based on how UML expresses parametrised models. Similar notations have been used in aspect-oriented modelling (AOM) approaches—for example, Theme/UML [28] or RAM [29]. More generally, our language composition technique is based on the notion of model weaving from AOM. Theme/UML, RAM, or Reuseware [30] are examples of aspect-oriented modelling techniques, which are asymmetric [31]; that is, they make a distinction between a base model and an aspect model (the model that is parametrised) that is woven into the base model. This is also true of our approach: M_{DSL} is the base model and M_{Obs} is the model that is woven into it. There is an alternative approach to AOM that is more symmetric and considers all models to be woven as equal. This is typically based on identifying corresponding elements in different models and merging these. Examples are UML package merge or signature-based merging [32]. Most types of AOM also consider syntactic weaving only, disregarding the semantics of the modular models. In contrast, we explicitly consider the model semantics and povide formal notions ensuring that the composition does not restrict the set of behaviours modelled in the base DSL.

Modular Model Transformations. The semantics of the languages we are discussing are expressed using model transformations. As such, work on modularising model transformations is of relevance to our work. Generally, this work can be distinguished into work on external and on internal modularisation of model transformations: The former considers a complete model transformation as the unit of modularity, while the latter aims to provide modularity inside individual transformations [33]. As we are modifying the internals of the base transformation by adding in detail described in the observer transformation rules, our approach is an *internal* modularisation technique. Nonetheless, ideas from external composition approaches are of interest to us. In particular, the work on model typing and reusable model transformations presented in [34] shows how the set of meta-model concepts effectively used by a model transformation can be computed and how this can be used to make the transformations more reusable. This is similar to the way in which we use the parametrised part of MM_{Obs} to make the observer transformation rules more reusable and to adapt them to different DSLs.

6 Conclusions and Outlook

We have presented a formal framework for language components for the specification of non-functional properties (NFPs) in domain-specific languages (DSLs). Specifically, this enables language designers to encapsulate the semantics of particular NFPs in a reusable language specification that can be woven into a base DSL specification to produce a DSL that also enables the modelling and analysis of that particular NFP in the context of a specific domain. We have presented conditions for the consistency of such language components; in particular these ensure that weaving a language component with a DSL does not add neither remove valid behaviours from the semantics of any expressions in that DSL.

Our work makes a number of assumptions about the structure of the base DSL as well as about the NFPs to be specified. In the future, we aim to reduce these assumptions to provide a more general framework for the specification of NFPs in DSLs. Most importantly, we will further study the cases where there is no simple alignment between Rls_{Obs} and Rls_{DSL}. This will require more powerful pattern-expression constructs in $Rls_{Obs}|_{MM_{Par}}$ and a more complex weaving algorithm that allows observer rules to be bound to multiple DSL rules and vice versa. Our current formalisation also does not consider the effect of well-formedness rules defined for any of the DSLs involved, although their addition should be relatively straightforward.

Acknowledments. We would like to thank Antonio Vallecillo for fruitful discussions throughout the work on this paper, and to Fernando Orejas for his collaboration in the development of the formalisation of the proposal. This work has been partially supported by Spanish Government Project TIN2011-23795.

References

1. Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: Empirical assessment of MDE in industry. In: Taylor, R.N., Gall, H., Medvidovic, N. (eds.) Proc. 33rd Int'l Conf. on Software Engineering (ICSE 2011), pp. 471–480. ACM (2011)
2. Hemel, Z., Kats, L.C.L., Groenewegen, D.M., Visser, E.: Code generation by model transformation: A case study in transformation modularity. Software and Systems Modelling 9(3), 375–402 (2010); Published on-line first at www.springerlink.com
3. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
4. Troya, J., Rivera, J.E., Vallecillo, A.: Simulating domain specific visual models by observation. In: Proc. 2010 Spring Simulation Multiconference (SpringSim 2010), pp. 128:1–128:8. ACM, New York (2010)
5. Rivera, J.E., Durán, F., Vallecillo, A.: A graphical approach for modeling time-dependent behavior of DSLs. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2009, pp. 51–55. IEEE (2009)
6. Rivera, J.E., Durán, F., Vallecillo, A.: On the Behavioral Semantics of Real-Time Domain Specific Visual Languages. In: Ölveczky, P.C. (ed.) WRLA 2010. LNCS, vol. 6381, pp. 174–190. Springer, Heidelberg (2010)
7. Zschaler, S.: Formal specification of non-functional properties of component-based software systems: A semantic framework and some applications thereof. Software and Systems Modelling (SoSyM) 9, 161–201 (2009)
8. Katz, S.: Aspect Categories and Classes of Temporal Properties. In: Rashid, A., Akşit, M. (eds.) Transactions on AOSD I. LNCS, vol. 3880, pp. 106–134. Springer, Heidelberg (2006)
9. Durán, F., Orejas, F., Zschaler, S.: Behaviour protection in modular rule-based system specifications (submitted for publication, 2012)
10. Kühne, T.: Matters of (meta-) modeling. Software and Systems Modeling 5, 369–385 (2006)
11. Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J.: On the Use of Higher-Order Model Transformations. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 18–33. Springer, Heidelberg (2009)
12. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformations. Foundations, vol. 1. World Scientific (1997)
13. Ehrig, H., Ehrig, K., Habel, A., Pennemann, K.H.: Theory of constraints and application conditions: From graphs to high-level structures. Fundamenta Informaticae 74(1), 135–166 (2006)
14. Ehrig, H., Prange, U., Taentzer, G.: Fundamental Theory for Typed Attributed Graph Transformation. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 161–177. Springer, Heidelberg (2004)
15. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer (2005)
16. Abadi, M., Lamport, L.: An old-fashioned recipe for real time. ACM ToPLaS 16(5), 1543–1571 (1994)
17. Lack, S.: An embedding theorem for adhesive categories. Theory and Applications of Categories 25(7), 180–188 (2011)
18. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: a model transformation tool. Science of Computer Programming 72(1-2), 31–39 (2008)
19. Atenea: Reusable Specification of Observers (2012), http://atenea.lcc.uma.es/index.php/Main_Page/Resources/ReusableObservers

20. Frolund, S., Koistinen, J.: QML: A language for quality of service specification. Technical Report HPL-98-10, Hewlett-Packard Laboratories (1998)
21. Aagedal, J.Ø.: Quality of Service Support in Development of Distributed Systems. PhD thesis, University of Oslo (2001)
22. Röttger, S., Zschaler, S.: CQML$^+$: Enhancements to CQML. In: Bruel, J.M. (ed.) Proc. 1st Int'l Workshop on Quality of Service in Component-Based Software Engineering, pp. 43–56 (June 2003)
23. Skene, J., Lamanna, D.D., Emmerich, W.: Precise service level agreements. In: Proc. 26th Int'l Conf. on Software Engineering (ICSE 2004), pp. 179–188. IEEE Computer Society (2004)
24. Bravenboer, M., Visser, E.: Concrete syntax for objects: Domain-specific language embedding and assimilation without restrictions. In: Proc. 19th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004), pp. 365–383. ACM Press (2004)
25. Bravenboer, M., Visser, E.: Parse Table Composition Separate Compilation and Binary Extensibility of Grammars. In: Gašević, D., Lämmel, R., Van Wyk, E. (eds.) SLE 2008. LNCS, vol. 5452, pp. 74–94. Springer, Heidelberg (2009)
26. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: a framework for compositional development of domain specific languages. Int'l Journal on Software Tools for Technology Transfer (STTT) 12(5), 353–372 (2010)
27. 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)
28. Carton, A., Driver, C., Jackson, A., Clarke, S.: Model-driven Theme/UML. Transactions on Aspect-Oriented Software Development (2008)
29. Kienzle, J., Abed, W.A., Klein, J.: Aspect-oriented multi-view modeling. In: Proc. 8th ACM Int'l Conf. on Aspect-Oriented Software Development (AOSD 2009), pp. 87–98. ACM (2009)
30. Heidenreich, F., Henriksson, J., Johannes, J., Zschaler, S.: On Language-Independent Model Modularisation. In: Katz, S., Ossher, H., France, R., Jézéquel, J.-M. (eds.) Transactions on AOSD VI. LNCS, vol. 5560, pp. 39–82. Springer, Heidelberg (2009)
31. Harrison, W.H., Ossher, H.L., Tarr, P.L.: Asymmetrically vs. symmetrically organized paradigms for software composition. Technical Report RC22685, IBM Research (2002)
32. Reddy, Y.R., Ghosh, S., France, R.B., Straw, G., Bieman, J.M., McEachen, N., Song, E., Georg, G.: Directives for Composing Aspect-Oriented Design Class Models. In: Rashid, A., Akşit, M. (eds.) Transactions on AOSD I. LNCS, vol. 3880, pp. 75–105. Springer, Heidelberg (2006)
33. Kleppe, A.G.: 1st European workshop on composition of model transformations (CMT 2006). Technical Report TR-CTIT-06-34, Centre for Telematics and Information Technology, University of Twente (June 2006)
34. Sen, S., Moha, N., Mahé, V., Barais, O., Baudry, B., Jézéquel, J.M.: Reusable model transformations. Software and Systems Modeling (SoSyM), 1–15 (2010)

Modular Well-Definedness Analysis for Attribute Grammars[⋆]

Ted Kaminski and Eric Van Wyk

Department of Computer Science and Engineering
University of Minnesota, Minneapolis, MN, USA
{tedinski,evw@cs.umn.edu}

Abstract. We present a modular well-definedness analysis for attribute grammars. The global properties of completeness and non-circularity are ensured with checks on grammar modules that require only additional information from their dependencies. Local checks to ensure global properties are crucial for specifying extensible languages. They allow independent developers of language extensions to verify that their extension, when combined with other independently developed and similarly verified extensions to a specified host language, will result in a composed grammar that is well-defined. Thus, the composition of the host language and user-selected extensions can safely be performed by someone with no expertise in language design and implementation. The analysis is necessarily conservative and imposes some restrictions on the grammar. We argue that the analysis is practical and the restrictions are natural and not burdensome by applying it to the Silver specifications of Silver, our boot-strapped extensible attribute grammar system.

1 Introduction

There has been considerable interest in extensible language frameworks and mechanisms for defining language implementations in a highly modular manner. Many of these frameworks allow *language extensions* that add new syntax and new semantic analyses to be composed with the implementation of a so-called *host language*, such as C or Java, resulting in an extended language with new, possibly domain-specific, features. It is our contention that for these frameworks to be useful to programmers and more widely used, the language extensions need to be able to be developed independently and the process of composing the host language with a selected set of extensions needs to be easily doable by programmers with no knowledge of language design and implementation.

There are several systems that support, to varying degrees, the development of modular and composable language extensions. For example, JastAdd [6] is a system based on attribute grammars (AGs) extended with reference attributes [8] which has been used to develop extensible compilers for Java [5] and Modelica. MetaBorg [4] is based on term rewriting systems with strategies [21] and has been

⋆ This work is partially supported by NSF Awards No. 0905581 and 1047961.

used to build extensible specifications of Java and other languages. SugarJ [7] is an extension to Java in which new syntax to SugarJ can be defined in imported libraries written in SugarJ. Our system, Silver [17,9], is an attribute grammar system with *forwarding* [16], a mechanism for language extension that is useful for combining extensions developed independently, which has also been used to define an extensible specification of Java [18] and other languages.

Our work is motivated by two important questions related to how easy it is for a non-expert to use these frameworks to create new languages from independently developed language extensions.

1. *How easy is it* to compose the host and extension specifications? Must significant glue code be written to combine them, or can one basically direct the tools to compose the host and selected extensions automatically?
2. *What assurances are provided* to the user that the selected extensions will in fact be composable such that the composition defines a working compiler, translator, or interpreter for the specified extended language?

The formalisms on which the above systems are based (context free grammars (CFG), attribute grammars, and term rewriting systems) are all quite easily and naturally composed, and thus adequately address our first question above. The problem arises with the second question: the composition may not have desirable or required properties for that formalism. For example, CFGs compose but the resulting grammar may be ambiguous and thus not suited as a specification for a parser. Similarly, composed AGs may not be complete (i.e. missing required attribute equations) or it may contain circularities in the attribute definitions.

In 2009, Schwerdfeger and Van Wyk described a *modular determinism analysis* for context free grammar fragments that could be run by the language extension developer to verify that their extension CFG, when composed with the host language CFG and other independently developed and similarly verified extension CFGs, would result in a deterministic grammar, from which a conflict-free LR(1) parse table could be constructed [14]. Formally, this was expressed as

$$(\forall i \in [1,n].isComposable(CFG^H, CFG_i^E) \wedge \ \mathit{conflictFree}(CFG^H \cup \{CFG_i^E\}) \\ \implies \mathit{conflictFree}(CFG^H \cup \{CFG_1^E, \ldots, CFG_n^E\})$$

Note that each extension grammar CFG_i^E is tested, with respect to the host language grammar CFG^H, using the *isComposable* and *conflictFree* analysis. If all extensions *independently* pass this analysis, then their composition is a CFG from which a conflict-free deterministic LR parse table can be generated. Of course, this analysis puts some restrictions on the type of syntax that can be added to a language as a composable language extension, but we have found these restrictions to be natural and not burdensome [14]. This analysis is used in Copper, the integrated parser and context-aware scanner packaged with Silver.

One of the primary contributions of this paper is a modular analysis for attribute grammar completeness and circularity-detection that is meant to provide the sort of assurances described in the second question above. The modular completeness analysis, called *modComplete*, provides the following guarantee:

$$(\forall i \in [1,n].modComplete(AG^H, AG_i^E)) \implies complete(AG^H \cup \{AG_1^E, ..., AG_n^E\}).$$

This analysis has the same form as the modular determinism analysis described above in that it verifies the property of attribute grammar completeness independently on each of the attribute grammar specifications of language extensions. The guarantee is that the non-expert can direct Silver to compose host and extension specifications (that pass this analysis) knowing that *the resulting attribute grammar will be complete*. Thus an entire class of common attribute grammar errors can be solved by the extension developers, who have some understanding of language design and implementation, and this burden will not fall on the non-expert doing the composition of the extensions.

Additional contributions of the paper include the following.

– In specifying the modular completeness analysis we define a notion of *effective completeness* that is useful in higher-order attribute grammars [23] as well as with forwarding.
– Unlike the original (non-modular) completeness analysis for attribute grammars with forwarding [16,1], our analysis distinguishes between missing equations and cycles in the attribute dependencies resulting in better error messages to the developer.
– We extend the completeness analysis to a modular circularity analysis.
– We evaluate the restrictions imposed by the modular analysis on the Silver-language source specification of our (bootstrapped) Silver compiler. This highly-modular specification was written before the analysis was developed. We find that the restrictions are not overbearing.

Paper contents: Section 2 provides needed background on attribute grammars, defines a simplified AG language over which the analyses are described, and defines the mechanism for computing flow-types of nonterminals in the AG. In Section 3 we present a modular analysis for effective completeness which we then extend in Section 4 to allow for more flexible organization of grammars and to include additional AG features found in full-featured attribute grammar specification languages such as Silver. Section 5 describes our experience in applying this analysis on the specification of Silver. In Section 6 we augment the analysis to also ensure non-circularity. Related (Section 7) and future work (Section 8) are discussed, followed by concluding remarks (Section 9).

2 Background

We begin with a broad overview of attribute grammar analysis, and then get more specific, with an example grammar and flow graphs later in the section.

Attribute grammars [11] are a formalism for describing computations over trees. Trees formed from an underlying context-free grammar are attributed with synthesized and inherited attributes, allowing information to flow, respectively, up and down the tree. Each production in the grammar specifies equations that define the synthesized attributes on its corresponding nodes in the tree, as well as the inherited attributes on the children of those nodes. These equations defining the value of an attribute on a node may depend on the values of other attributes on itself and its children.

An attribute grammar is considered *complete* if there are no missing equations. That is, for all productions, there is an equation for every synthesized attribute that occurs on the nonterminal the production constructs, and for all children of the production, there is an equation for every inherited attribute that occurs on that child's nonterminal.

An attribute grammar is considered *non-circular* (and, if also complete, then *well-defined*) if on every possible tree allowed by the context-free grammar, there is no attribute whose value, as specified by the attribute equations, eventually depends upon itself. Knuth presents [11] an algorithm to ensure non-circularity in a complete attribute grammar. This algorithm is based upon constructing a graph for each production that describes how information flows around locally within that production. Alone, this is insufficient information: a production has no idea how its own inherited attributes might depend on its own synthesized attributes, nor can it know how its children's synthesized attributes might depend upon the inherited attributes it provides them. This global information is determined by a data flow analysis that results in a set for every nonterminal containing every possible flow of information between attributes on that nonterminal. Non-circularity can be checked using these sets.

Attribute grammars have been extended in a variety of ways. Higher-order attribute grammars [23] allow attributes to themselves contain trees that are as-yet undecorated by attributes. These trees are made useful by permitting productions to "locally anchor" a tree and decorate it, as if it had been a child. A child, however, is supplied when a tree is created, whereas these "virtual children" are defined by an equation during attribute evaluation that, of course, may have dependencies on the values of attributes, like any other equation. Higher-order attribute grammars amend the notion of *completeness* by requiring all inherited attributes to be supplied to these "virtual children," as well. The notion of *non-circularity* is also further extended, by requiring that the synthesized attributes of each "virtual child" have an implicit dependency on the equation defining that tree. These virtual children can potentially lead to the creation of an unbounded number of trees, and thus the nontermination of attribute evaluation.

Forwarding [16] was introduced to attribute grammars to allow for language extension. A language extension that introduces new productions combined with another that introduces new attributes on existing nonterminals presents a serious problem for completeness: the new attribute may not be defined on the new production. This is sometimes referred to as the *expression problem*.[1] If, however, those new productions *forward* requests for synthesized attributes without defining equations to *semantically equivalent* trees in the host language, the new attributes can simply be evaluated on that host tree instead and returned as the value of the attribute for the *forwarding* production, resolving the problem. Forwarding amends the notion of *completeness* by allowing a production that forwards to omit synthesized attribute equations, as they can instead be supplied by

[1] http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt

$$\begin{array}{rcl}
D & ::= & \texttt{nonterminal}\ n_{nt}\ \texttt{;} \quad |\ \texttt{synthesized attribute}\ n_s \texttt{::} T\ \texttt{;} \\
 & | & \texttt{inherited attribute}\ n_i \texttt{::} T\ \texttt{;} \quad |\ \texttt{attribute}\ n_a\ \texttt{occurs on}\ n_{nt}\ \texttt{;} \\
 & | & \texttt{production}\ n_p\ \ n_{lhs} \texttt{::} n_{nt}\ \texttt{::=}\ \overline{n_{rhs} \texttt{::} T}\ \{\ \overline{S}\ \} \\
 & | & \texttt{aspect production}\ n_p\ \ n_{lhs} \texttt{::} n_{nt}\ \texttt{::=}\ \overline{n_{rhs} \texttt{::} T}\ \{\ \overline{S}\ \} \\
S & ::= & n_{lhs}\,\texttt{.}\,n_s\ \texttt{=}\ E\ \texttt{;} \quad |\ n_{rhs}\,\texttt{.}\,n_i\ \texttt{=}\ E\ \texttt{;} \\
 & | & \texttt{local}\ n_{local} \texttt{::} T\ \texttt{=}\ E\ \texttt{;} \quad |\ n_{local}\,\texttt{.}\,n_i\ \texttt{=}\ E\ \texttt{;} \\
 & | & \texttt{forwards to}\ E\ \{\ \overline{A}\ \}\ \texttt{;} \\
A & ::= & n_i\ \texttt{=}\ E \\
E & ::= & n_{local}\ \ |\ n_{lhs}\,\texttt{.}\,n_a\ \ |\ n_{rhs}\,\texttt{.}\,n_a\ \ |\ n_{local}\,\texttt{.}\,n_a\ \ |\ \cdots \\
T & ::= & n_{nt}
\end{array}$$

Fig. 1. The language $\textsc{Ag}_f$

the forward tree. Forwarding's necessary modifications to *non-circularity* roughly follow those of higher-order attribute grammars: the forward tree appears as a "virtual child" and all synthesized attributes on a forward tree have a dependency on the equation defining this tree. Forwarding introduces implicit "copy" rules for synthesized attribute equations the production is missing, as well as for any inherited attributes not supplied to the forward tree.

A problem for completeness identified by the forwarding paper, but also existing for higher-order attribute grammars, is the inconvenience of requiring *all* inherited attributes to be supplied. Frequently, only a subset of synthesized attributes are demanded, which in turn only require a subset of inherited attributes. This shows up frequently for forwarding, where a production may only use its own children to synthesize a "pretty print" attribute, relying on the forward tree for everything else (such as "translation.") This production would be required to supply its children with inherited attribute equations that are never used, merely to pretty print the tree. An amended notion of *effective completeness* of inherited attributes can be used instead: we require that all inherited attributes *needed to compute any accessed synthesized attribute* be supplied, instead of simply all of them outright. An *effectively complete* attribute grammar can compute non-circularity in the same manner as a complete one: if these equations are never demanded, they will never have an effect on flow graphs of a nonterminal, and can be ignored in the flow graphs of a production.

2.1 The Language

The language defined in Fig. 1 describes a simplified attribute grammar language, based on our attribute grammar language, Silver, but with many orthogonal features (such as an indication of a starting symbol) omitted, and some introduced later in Section 4. It should generalize well to other attribute grammar languages that include forwarding. Declarations are represented by D. Attributes are declared separately from the occurs-on declarations that indicate what nonterminals (n_{nt}) the attribute (n_a) decorate. Semantic equations (S)

can be supplied separately from declaring a production via *aspect productions.* The possible equations include defining synthesized attributes (n_s) for the production's left hand side (n_{lhs}) and inherited attributes (n_i) for children (n_{rhs}) and locals (n_{local}). Local declarations allow for "locally anchoring" trees, as in higher-order attribute grammars. Finally, productions may forward, and provide equations to change the inherited attributes supplied to the forward tree, which are otherwise copied from the forwarding tree. Note that one restriction not reflected in the grammar is that a forward may not appear in an aspect production. Expressions (E) are largely elided from the language above. Only those expressions that induce dependencies in a production's flow graph are shown. As a result, even though referring to a child's tree (n_{rhs}) is a valid expression, it does not appear in E because a child tree is simply a value with no incurred flow dependencies. Similarly, any sort of function call expression induces no dependencies on its own, and simply aggregates dependencies from its component expressions.

We will write AG^H to indicate a host language, which should be a valid attribute grammar consisting of a set of declarations ($\overline{D}$.) We will write language extensions (also consisting of a set of declarations $\overline{D}$) as AG^E, and these grammars should be valid *in combination with* the host language they extend (i.e. $AG^H \cup AG^E$ is valid for each AG^E.) By validity, we mean certain properties about the grammars that we consider to be part of a "standard" environment and semantic analysis. For example, we will assume the grammars have all names bind properly and are type correct. We will assume that duplicate declarations of nonterminals, attributes, and productions are caught locally, and that if they occur in different grammars they are not duplicates but truly different symbols with different "fully-qualified" names based on the name of the grammar.

Fig. 2 shows an example of a small host language with two extensions. Note that one extension introduces a new production that forwards to its semantic equivalent in the host language (via De Morgan's laws), and that another extension introduce a new "translation" attribute for the productions in the host language. Both the `errors` and `java` attributes for the `or` production will be ultimately be computed by consulting the forwards tree.

The flow graphs for some of the productions of the composed grammars of Fig. 2 are shown in Fig. 3. Note that we use arrows to represent *dependencies* necessary to evaluate attributes, rather that using them to indicate the direction of information flow. A *flow type* [13] is a function $f_{nt} : syn \rightarrow \{inh\}$ that defines, for a nonterminal, what inherited attributes each synthesized attribute that occurs on that nonterminal may depend upon. A flow type can also be thought of as a graph, where edges are always from synthesized attribute nodes to inherited attribute nodes. A production's flow graph and the flow types for each nonterminal can be combined into a *stitched flow graph.* In the figure, the flow type for `Expr` is shown, along with the stitched flow graph for the production `not`. The flow types introduce edges between the attributes on the nonterminals of each child, virtual child, and forward in the stitched flow graph.

host grammar

```
nonterminal Expr;
synthesized attribute errors::[Msg];
inherited attribute env::Env;
attribute errors occurs on Expr;
attribute env occurs on Expr;

production and
e::Expr ::= l::Expr r::Expr
{ e.errors = l.errors ++ r.errors;
}
production not
e::Expr ::= s::Expr
{ e.errors = s.errors;
}
production var
e::Expr ::= n::Name
{ e.errors = lookup(e.env, n.lexeme);
}
```

or extension

```
production or
e::Expr ::= l::Expr r::Expr
{ forwards to
    not(and(not(l), not(r))); }
```

java extension

```
synthesized attribute java::String;
attribute java occurs on Expr;

aspect production and
e::Expr ::= l::Expr r::Expr
{ e.java = l.java ++ "&&" ++ r.java; }
aspect production not
e::Expr ::= s::Expr
{ e.java = "!" ++ s.java; }
aspect production var
e::Expr ::= n::Name
{ e.java = n.lexeme; }
```

Fig. 2. An example of a host grammar for boolean propositions, with two extensions

2.2 Flow Type Computation

Knuth's (corrected) algorithm for ensuring non-circularity can be thought of as computing a set of flow types for each nonterminal. For now, we are only concerned with computing a single flow type per nonterminal. The flow type computation is a function $\mathit{flowTypes}(\overline{D}) : nt \to syn \to \{inh\}$. Supplied with a set of declarations that form a valid attribute grammar, it results in a function mapping each nonterminal to a flow type. In principal, flow types could be computed using the standard non-circularity algorithm, by merging the set of flow types into just one for each nonterminal. It is significantly more efficient to compute them directly, with a slightly modified algorithm:

1. Local flow graphs for each production are computed.
2. Begin with an empty flow type for every nonterminal & synthesized attribute.
3. Iterate over every production. Produce a stitched flow graph for that production using the *current* set of flow types. If there are any paths from a synthesized attribute on the production's left hand side nonterminal to an inherited attribute on the same that are not yet present in the current flow type for that nonterminal, add them to the nonterminal's flow type.
4. Repeat until no new edges are introduced in a full pass over the productions.

We also have need to extend the domain of the flow type function ft_{nt} to track not just synthesized equation dependencies, but also those for forward equations. We will write $\mathit{ft}_{nt}(fwd)$ to refer to the dependencies potentially necessary to evaluate all forward equations for the nonterminal nt.

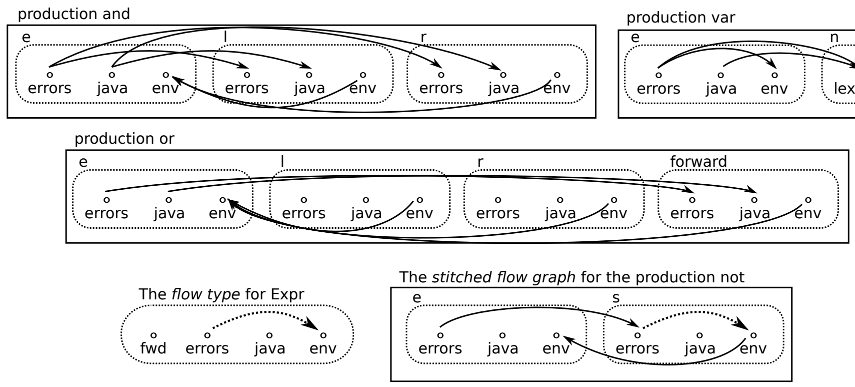


Fig. 3. Flow graphs for the grammar of Fig. 2. Also including the flow type of Expr, and an example stitched flow graph.

3 Modular Flow Analysis for Completeness

The modular completeness analysis *modComplete* checks six properties of the host and each extension attribute grammar individually to ensure that the composition of the host and these extensions will be effectively complete. Two of these properties require the flow types for host and extension grammars to have been computed. The analysis *modComplete* is defined as follows:

$$\begin{aligned}&modComplete(AG^H, AG^E) \triangleq\\ &\quad noOrphanOccursOn(AG^H, AG^E) \;\wedge\; noOrphanAttrEqs(AG^H, AG^E) \;\wedge\\ &\quad noOrphanProds(AG^H, AG^E) \;\wedge\; synComplete(AG^H, AG^E) \;\wedge\\ &\quad modularFlowTypes(flowTypes(AG^H), flowTypes(AG^H \cup AG^E)) \;\wedge\\ &\quad inhComplete(AG^H, AG^E, flowTypes(AG^H \cup AG^E))\end{aligned}$$

Each of these checks is defined in turn below. In these discussions, we will use the notation "n is exported by AG_1 or AG_2" to mean that the symbol n is declared in the grammars AG_1 or AG_2. This ensures that when `export` statements are introduced in the extended analysis (Section 4) the language used below will still be a correct description of the requirements. We will also say that something is "in scope" if the information is available in the standard environment for a grammar, as described in Section 2.

No orphan occurs-on declarations: The check *noOrphanOccursOn* ensures that there will be no duplicate occurs-on declarations in the composition of the host and all extension grammars, denoted AG^{all}.

noOrphanOccursOn(AG^H, AG^E) holds if and only if each occurs-on declaration "`attribute` a `occurs on` nt" in $AG^H \cup AG^E$ is exported by the grammar declaring a or the grammar declaring nt.

Every occurs-on declaration will have all potentially duplicate occurs-on declarations in its scope. This prevents, for example, an occurs-on relation being declared in an extension AG^E for an attribute and non-terminal that are both defined in the host AG^H. However, it still permits the occurs-on declaration if the nonterminal or the attribute are declared in the extension.

No orphan attribute equations: The check *noOrphanAttrEqs* ensures that there will be no more than one equation for the same attribute for the same production in AG^{all}.

noOrphanAttrEqs(AG^H, AG^E) holds if and only if each equation $n.a = e$ in a production p is exported by the grammar declaring the (non-aspect) production p or the grammar declaring the occurs-on declaration "`attribute` a `occurs on` nt" (where n has type nt.)

Similar to the orphaned occurs-on declarations, this rule ensure that each equation must have all potential duplicate equations in scope. This relies on the orphaned occurs check: if two extensions can independently make the same attribute occur on the same nonterminal, in such a way that the standard environment cannot catch the duplicate occurs, then it also cannot catch the duplicate equations. Also note that this rule applies equally to synthesized and inherited attribute equations, and that we have not yet ensured there exists *at least* one equation, only ruled out the possibility of more than one.

No orphan production declarations: The check *noOrphanProds* ensures that extension productions forward, in order to allow forwarding to solve the problem its introduction is intended to solve.

noOrphanProds(AG^H, AG^E) holds if and only if for each production declaration p in $AG^H \cup AG^E$ with left hand side nonterminal nt, the production p is either exported by the grammar declaring nt, or p forwards.

This rule is different from the previous two in that there's no choice of where a declaration can appear. The grammar declaring the nonterminal in effect declares a fixed set of productions that do not forward, and this set will be known to every extension grammar. As a result, a production is either in the host language AG^H, declared in the extension and forwards, or is declared in the extension and its left hand side is a nonterminal also declared in the extension.

Completeness of synthesized equations: The check *synComplete* ensures that for every production, an equation exists to compute every synthesized attribute that occurs on its left hand side non-terminal in AG^{all}.

> *synComplete*(AG^H, AG^E) holds if and only if for each occurs-on declaration `attribute` a `occurs on` nt, and for each non-forwarding production p that constructs nt, there exists a synthesized equation defining the value of that attribute for that production.

This rule relies on the previous orphaned productions rule to ensure that all non-forwarding productions are in scope at the occurs declaration. It further relies on the previous orphaned equations rule (and thus, the orphaned occurs rule) to ensure that all potential equations for those non-forwarding productions are in scope, and thus we can check for their existence. Any production that forwards will be able to obtain a value for this attribute via its forward tree if it lacks an equation, and therefore they do not need checking.

The rules up to this point ensure synthesized completeness in a modular way. No set of composed extensions that satisfy the above rules could result in a missing or duplicate synthesized equation for any production in the resulting composed attribute grammar, AG^{all}. These rules critically rely on forwarding. Without forwarding, nonterminals are no more extensible than standard datatypes in ML or Haskell, in that new synthesized attributes are possible, but not new productions. We now turn to inherited attributes and flow types.

Modularity of flow types: The check *modularFlowTypes* ensures that all grammars will agree on the flow types. For occurrences in AG^H, an extension is not allowed to change the flow types. For those in AG^E, it ensures they depend upon those inherited attributes needed to evaluate forward equations, at a minimum.

> *modularFlowTypes*$(\mathit{flowTypes}(AG^H), \mathit{flowTypes}(AG^H \cup AG^E))$ holds if and only if given $\mathit{ft}^H_{nt} \in \mathit{flowTypes}(AG^H)$ and $\mathit{ft}^{H\cup E}_{nt} \in \mathit{flowTypes}(AG^H \cup AG^E)$,
>
> 1. For all synthesized attribute occurrences `attribute` s `occurs on` nt declared in AG^H, $\mathit{ft}^{H\cup E}_{nt}(s) \subseteq \mathit{ft}^H_{nt}(s)$
> 2. For all nonterminals nt declared in AG^H, $\mathit{ft}^{H\cup E}_{nt}(fwd) \subseteq \mathit{ft}^H_{nt}(fwd)$
> 3. For all synthesized attribute occurrences, s, declared in AG^E where nt is declared in AG^H, $\mathit{ft}^{H\cup E}_{nt}(s) \supseteq \mathit{ft}^H_{nt}(fwd)$.

The first two properties prevent an extension from modifying the flow types of host language occurrences, including the forward flow type. The purpose of the requirement on extension attributes is less obvious, but boils down to potentially needing to be able to evaluate forwards to get to the host language production on which this attribute is defined (in particular for productions from other extensions.) Our implementation deals with this last requirement by simply modifying the flow types, rather than raising an error if one runs afoul.

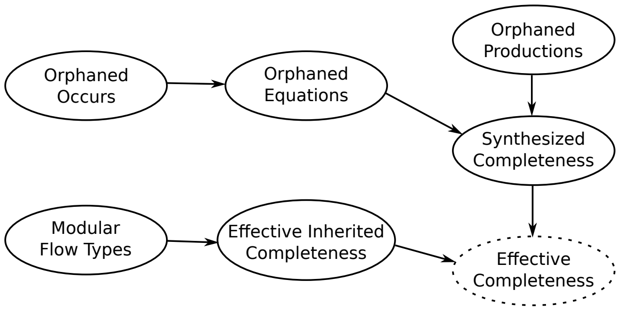


Fig. 4. The dependencies among the modularity analysis rules

Effective completeness of inherited equations: The check *inhComplete* ensures that no evaluation of attributes will demand an inherited attribute that is missing a defining equation. For each access of a synthesized attribute from a child or local, we ensure that a sufficient set of inherited attributes has been supplied to that child or local.

$inhComplete(AG^H, AG^E, flowTypes(AG^H \cup AG^E))$ holds if and only if for every production p in $AG^H \cup AG^E$ and for every access to a synthesized attribute $n.s$ in an expression within p (where n has type nt,) and for each inherited attribute $i \in ft_{nt}(s)$, there exists an equation $n.i = e$ for p.

This rule is sufficient to ensure that, when the host and extension are composed alone, no missing inherited attribute equations will be demanded. Together with the previous modularity rule for flow types, it's is also sufficient to ensure this property holds for AG^{all} because the flow type for host attribute occurrences cannot be changed by an extension. Finally, we can also be sure there are no duplicate inherited equations thanks to the earlier orphaned equations rule.

Fig. 4 summarizes the dependencies between the rules and how effective completeness is established. We achieve effective completeness for AG^{all} by ensuring modular effective inherited completeness and modular synthesized completeness hold for each extension individually, with no further checks necessary.

This analysis ultimately boils down to two major restrictions on extensions. First, the host language fixes a set of non-forwarding productions, and any extension to this language must ultimately be expressible in terms of these productions (via forwarding.) Second, for each host language synthesized attribute occurrence, the host language fixes a set of inherited dependencies. Our experience so far suggests that host languages are typically rich enough to express many interesting extensions, and many extensions that need inherited information can usually "piggy back" that information on already existing host language inherited attributes (e.g. the environment.) Further evaluation of the practicality of these rules will be presented in Section 5.

$$
\begin{array}{rcl}
G & ::= & \cdot \quad | \ \texttt{grammar} \ \ n_g \ \{ \ \overline{M} \ \ \overline{D} \ \} \ G \\
M & ::= & \texttt{import} \ \ n_g \ \ \texttt{;} \quad | \ \texttt{export} \ \ n_g \ \ \texttt{;} \quad | \ \texttt{option} \ \ n_g \ \ \texttt{;} \\
 & | & \texttt{export} \ \ n_g \ \ \texttt{with} \ n_{tg} \ \ \texttt{;}
\end{array}
$$

Fig. 5. Extending the language AG_f with a module system

4 Extending the Analysis

We extend AG_f in two ways: first, by introducing a module system for it, and second, by introducing several other features found in Silver to the language that affect the modularity analysis.

4.1 A Module System

So far we have referred to attribute grammars AG^H and AG^E as host and extension grammars, however this does not reflect the reality of developing large attribute grammars. In Fig. 5, we introduce a module system to AG_f. Each grammar is named, and consists of a set of module statements and a set of attribute grammar declarations. We will consider each module statement in turn.

The analysis presented in Section 3 is easily generalized to apply to acyclic import graphs between grammars. An import (`import` n_g) makes explicit the information available in the standard environment for a module, whereas we previously merely stated that extension grammars have in their environment their host language's symbols. To apply the modularity analysis, we consider every grammar to be an "extension" to the composition of all the grammars it imports, and we otherwise apply the analysis unchanged. The modularity rules are formulated such that being designated a "host" or "extension" confers no special status beyond what "is extending" and what "is being extended." A grammar that imports nothing satisfies all the modularity rules. In addition to allowing the host language to be broken up into multiple grammars, this allows extensions to make use of other extensions.

Exports (`export` n_g) allow grammars to be broken apart arbitrarily into different pieces, and be largely treated as a single grammar for the purpose of the analysis. For example, we might want to separate the concern of type checking from the host language, but type checking may not pass the analysis as an "extension." To allow this, we can have the host language grammar export the type checking grammar, essentially designating it part of the host. Again, the modularity rules do not require any changes, because we were already careful to word the rules to note when a symbol is "exported by" a grammar.

So far, however, this is still a limiting situation for host languages. Many languages have multiple potential configurations that cannot be reflected as extensions, for modularity reasons or simply because they conflict outright with alternative configurations. For example, GHC Haskell has many optional features

$$\begin{array}{rcl} E & ::= & \texttt{ref}\ n_{local} \mid \texttt{ref}\ n_{lhs} \mid \texttt{ref}\ n_{rhs} \mid E\,.\,n_a \\ & \mid & \texttt{case}\ E\ \texttt{of}\ \overline{p \to E_p} \mid n_v\,.\,n_a \mid \cdots \\ p & ::= & n_p(\overline{n_v}) \mid \_ \end{array}$$

Fig. 6. Extending the language AG_f with references and pattern matching

that can be enabled, some of which cannot be activated together. To support these configurations, we introduce *options* to the language. An *option* (`option` n_g) declaration behaves identically to an export, when computing any of the requirements imposed by Section 3, but has no effect on the standard environment. This allows, for example, new non-forwarding productions to be introduced in a grammar that is not necessarily the host language, but still allows the modularity rules to ensure that any extensions account for their existence. This in turn necessitates another new feature, *conditional exports*, to allow an extension grammar to optionally include support for a feature that may or may not be in the host language (because it is an optional component.) A conditional export (`export` n_g `with` n_{tg}) is identical to a normal export of n_g, so long as the importing module also imports its triggering grammar (n_{tg}), such as an optional component of the host language. If not, the conditional export is ignored.

Finally, our last modification to the module system is to account for import cycles. The above generalization to arbitrary import graphs works until a cycle is encountered. If there is a cycle, then a "host" grammar will actually include the "extension" in its flow type computations, and thus won't flag that extension's violations. To handle cycles (and as a bonus, to compute flow types much more efficiently,) we compute flow types just once, globally. To ensure we still generate the same flow types, production graphs are partitioned into standard edges and *suspect edges.* Only standard edges are used when stitching production graphs, and computing updates to flow types. Suspect edges are generated from synthesized (and forward) equations that are not permitted to affect their corresponding flow type. (Note that this includes implicit synthesized equations generated by forwards.) During the flow type computation, the dependencies suspect edges would introduce are considered, and only those direct edges to LHS inherited attributes that are already in their corresponding flow type are admitted as a standard edge. In this way, these edges' valid dependencies have effect on the rest of the graph, while avoiding affecting their own flow type.

4.2 Additional Language Features

Silver also supports a version of reference/remote attributes [8,3], and pattern matching [9]. References refer to trees that have already been given their inherited attributes elsewhere, whereas higher-order attributes refer to a tree that is "anchored" and supplied inherited attributes locally. References present a unique problem: the "decoration site" of the reference is unknown. It is not possible to know what inherited attributes where supplied, if any. We adopt an extremely conservative solution to this obstacle. We will refer to the set of *all* inherited attributes known to occur on a nonterminal nt **by the grammar that declares** nt as $ft_{nt}(ref)$. This particular choice isn't important, only that all grammars will agree on a consistent set. Whenever a reference is taken (`ref` n_n, where n_n has type nt), we consider it to depend on $ft_{nt}(ref)$. Whenever an attribute is demanded from a reference (E `.` n_a where E is a reference to a nonterminal nt), we ensure that the flow type $ft_{nt}(a) \subseteq ft_{nt}(ref)$. Thus, the set of inherited attributes on a reference type is fixed by the host language. This could be extremely limiting, however in Section 5 we present some evidence that it is still quite workable.

Silver allows for pattern matching on trees, in a manner that respects forwarding [9]. In order to perform the pattern matching, therefore, we must be able to evaluate the forward equation for any production, and therefore the scrutinee must be supplied $ft_{nt}(fwd)$. As a bonus, the interaction of pattern matching with forwarding, combined with the orphaned production rules of Section 3, allows pattern matching expressions to ensure all possible cases are covered. Pattern matching is capable of extracting references to the children of a production, but in a manner that constitutes a known "decoration site," and thus we do not have to fall back on treating them like references (so long as reference is not otherwise taken.) Given a set of inherited attributes known to be supplied to the scrutinee i, for each case matching a production p, we can flow i through the production flow graph for p, and determine the set of inherited attributes that will be supplied to each child of p extracted as a pattern variable. For each pattern variable attribute access (n_v `.` a where n_v has type nt) we can ensure the flow type $ft_{nt}(a)$ is a covered by this set.

Silver has a number of other features that have relatively trivial effects on the modularity analysis: collection attributes, autocopy attributes, and newly-introduced default equations, see Section 5. The later two present no complications, and in fact are greatly simplified by the analysis: for grammars that pass the analysis, all implicit equations they introduce are statically known. Our notion of collection attributes do not allow for remote contributions, unlike those of Boyland [3]. Collections attributes are synthesized attributes that have two different kinds of defining equations: base and contribution. Base equations are treated identically to ordinary equations for the modularity analysis. Contributing equations are allowed to exist in places that violate the orphaned equations rule (as there are allowed to be any number of them), but are still subject to the inherited completeness rule and the host language's flow type.

Finally, Silver supports an alternative composition model that more resembles that of classes in object-oriented languages. *Closed* nonterminals, instead of having a fixed set of non-forwarding productions, have a fixed set of attributes instead, and non-forwarding productions may appear in any grammar. As an example, many languages will have a concrete syntax tree with a single synthesized attribute to construct an abstract syntax tree. This poses a potential serious annoyance for extensions, given the modularity rules: the extension might have to duplicate the forwards for the concrete and the abstract productions. Making concrete syntax nonterminals closed resolves the issue.

5 Evaluation of Modular Completeness Analysis

To evaluate the analysis, and the practicality of writing specifications that satisfy the restrictions imposed, we have implemented the analysis and applied it to the Silver specifications for the Silver compiler. We chose to analyze Silver itself because it was one of the most complex attribute grammar specifications we have. The host language has an interesting type system, it includes several optional components that may or may not be included, it has several composable language extensions (some add new syntax, some add new attributes, some both), and it has a full translation to Java. It's also the Silver specification we use most, and as a result we believe it would have the fewest bugs.

The caveat for evaluating on Silver is that what is considered "host" vs "extension" is also under our control. To alleviate this concern somewhat, we briefly describe a few of the extensions, to demonstrate they are interesting and non-trivial. A "convenience" extension introduces new syntax that greatly simplifies making large numbers of similar occurs declarations, by allowing nonterminal declarations to be annotated with a list of them. A testing extension adds several constructs for writing and generating unit tests for the language specification. An "easy terminals" extension allows simple terminals to be referred to by their lexeme in production declarations (using `'to'` instead of `To_kwd`, for example.) Finally, the entire translation to Java is implemented as a composable language extension.

A technical report documents all issues raised by the analysis and the changes made to address them [10]. A brief summary of those results is reported here.

Silver focuses specifically on language extension, and as a result, we had chosen not to implement the monolithic analysis, to better enable separate compilation. Without the modular analysis, we had simply gotten by without a static completeness analysis. One set of changes made in response to the analysis were expected: we found (and fixed) several bugs. The analysis found several legitimately missing synthesized and inherited attribute equations. It also found several productions that should have been forwarding, but were not.

Another positive set of changes improved the quality of the implementation, even if they did not directly fix bugs. We discovered several extraneous attribute occurrences that simply never had equations, and were never used either. Many uses of references were found to be completely unnecessary and eliminated. One particularly interesting change has to do with how concrete syntax specifications are handled in Silver. Silver's host language supplies a "standard" set of declarations for concrete syntax, while Copper-specific declarations are kept in a separate optional grammar. The analysis raised a simple error: the Java translation attributes for parser declarations were being supplied by the Copper grammar, which is a violation of the rules. The decision was made to move this parser declaration out of the host language and into the Copper optional grammar, and that it actually belonged there all along: it does, after all, generate a Copper parser.

Some parts of the analysis were motivated by our attempts to get Silver to conform to the restrictions. The specification of the Silver host language is broken up into several grammar modules for standard software engineering reasons of modularity. Many of these modules are *not* meant to be considered composable language extensions. This motivated the introduction of the "option" module statement. We also found that we were abusing forwarding, using it as a way to define default values for attributes where the forwarded-to tree was not, in fact, semantically equivalent to the forwarding tree in the slightest. To resolve this, we introduced the notion of *default* attribute values as a separate concept, so these uses of forwarding could be eliminated.

There were two sorts of negative changes made to the Silver specification in order to make it pass the analysis. The first of these resulted from the conservative rules for handling reference attributes. Two sets of inherited attribute equations had to be supplied whose values are never actually used. In one case, a nonterminal representing concrete syntax information has two synthesized attributes, one for a normalization process and one for translation. These attributes have different inherited dependencies, but the analysis required the full set for both because the synthesized attributes internally used references.

The second sort of negative changes involved introducing workarounds for code that we already knew needed refactoring, but we did not want to fix, yet. In fact, in many of these cases, the analysis lead us to code that already had "TODO" comments complaining about a design for reasons unrelated to the analysis. The most significant of these is the use of a single nonterminal as a data structure to represent several different types of declarations in Silver (attributes, types, values, occurs-on, etc.) This is a legacy from when Silver did not have parametric polymorphism and needed to group all of these together into a single monomorphic type. To make the analysis pass, we introduced "error" equations for attributes that did not have sensible values otherwise (e.g. attributes for *value* declarations that do not apply to *type* declarations.) However, the use of "error" equations to make the analysis pass still provides a statically detectable indication that we are, in essence, making a temporary end-run around the analysis. These error equations are essentially a form of "technical debt" - legitimate problems that we will change later, but for various reasons decide not to do just yet. On the positive side, it lets the developer distinguish between bugs to fix now and changes to make later.

In our experience with Silver, the analysis found a few bugs, motivated us to fix a small number of design flaws, and introduced a relatively small amount of technical debt. The analysis also found problems that inspired some changes to Silver (*e.g.* options, defaults) to more easily write specifications that satisfied the restrictions. We found the use of "error" equations reasonable, as a way to document technical debt and refactorings that should be made at a later date (or in the worst case, an explanation of why the attribute really could never be demanded, despite the analysis indicating otherwise.) In the end, most of the changes necessary were to the host language itself, and the extensions then passed without further effort.

6 Extending the Modular Analysis to Circularity Analysis

So far we have focused only on ensuring *completeness* of the composed attribute grammar in a modular way. This involved making use of flow information that is typically used to ensure non-circularity, but we only computed a single flow type for each nonterminal instead of a set, as is normally done in circularity analyses. To ensure non-circularity, we will go back to calculating these flow sets once again.

In this section we define a modular non-circularity analysis. As in the modular completeness analysis, this analysis is performed independently on each extension to ensure non-circularity in the final composed grammar. This analysis, *modNonCircular*, is defined as follows:

$$\begin{aligned}&\mathit{modNonCircular}(AG^H, AG^E) \triangleq\\&\quad \mathit{modComplete}(AG^H, AG^E) \wedge \mathit{nonCircular}(AG^H \cup AG^E) \wedge\\&\quad \mathit{modFlowSets}(\mathit{flowSets}(AG^H), \mathit{flowSets}(AG^H \cup AG^E))\end{aligned}$$

The *modComplete* analysis is unchanged, and the analysis *nonCircular* is the standard non-circularity analysis for attribute grammars [11,23,16].

Modularity of flow sets: The check *modFlowSets* ensures that extensions do not introduce any flow types whose host language component is not already accounted for in the host language's flow sets. The extension can still introduce new flow types, so long as they differ only in edges from synthesized attribute occurrences declared in AG^E.

$\mathit{modFlowSets}(\mathit{flowSets}(AG^H), \mathit{flowSets}(AG^H \cup AG^E))$ holds if and only if for every $\mathit{ft}_{nt}^{H \cup E}$ in $\mathit{flowSets}(AG^H \cup AG^E)$, there is a ft_{nt}^{H} in $\mathit{flowSets}(AG^H)$ such that for every $s \in \mathit{dom}(\mathit{ft}_{nt}^{H})$, $\mathit{ft}_{nt}^{H \cup E}(s) \subseteq \mathit{ft}_{nt}^{H}(s)$.

This rule is justified by a well-known optimization for computing flow sets, where flow types that are "covered" by another flow type in the same flow set can be discarded. All this rule does is ensure that all flow types generated by the extension (restricted back to the host language) can be discarded in this fashion, and consequently do not affect the non-circularity check.

This analysis, too, can handle the extensions introduced in Section 4, with the caveat that the conservative rules introduced for handling references may result in false positive circularities. Finally, while we have not yet evaluated this analysis in practice, we believe the major potential problem for it is ensuring that the extension developer can understand the resulting requirements. The graphs that are contained in the flow sets are not necessarily subject to easy intuition the way flow types are.

7 Related Work

Knuth introduced attribute grammars [11] and provided a circularity analysis. In presenting higher-order attributes, Vogt *et al.* [23] extended Knuth's completeness and circularity analyses to that setting. Reference and remote attributes do not have a precise circularity analysis [3], as the problem is undecidable. Completeness in these settings is simply a matter of using occur-on relationships to check for the existence of equations for all of the required attributes. With forwarding, flow-analysis is used to check completeness and thus a *definedness analysis* that combines the check of completeness and circularity was defined [16,1]. This analysis used *dependency functions* instead of flow graphs in order to distinguish between synthesized attributes that depend on no inherited attributes and those that cannot be computed because of a missing equation or circularity, and thus conflate this two types of errors. All of these are non-modular analyses.

Saraiva and Swierstra [13] present *generic attribute grammars* in which an AG has grammar symbols marked as *generic* and not defined in the grammar. Composition in this model is the instantiation of these generic symbols with specific ones. Here, flow-types can be computed on the generic grammar being imported and then used when flow analysis is done on the instantiating/importing grammar. This composition model, however, is very different from the language extension model described in Section 1. It does not allow for multiple independent extensions to be composed, except by first merging them into a single extension, on which the analysis must then be performed, effectively making it monolithic.

In AspectAG, Viera *et al.* [20] have shown the completeness analysis can be encoded in the type system of Haskell. However, this analysis is again performed at the time of composition (by the type checker) and is thus a monolithic analysis.

Current AG systems such as JastAdd [6] and Kiama [15] do not do static flow analysis but, like previous versions of Silver, instead provide error messages at attribute evaluation time that indicates the missing equation or circularity. An extension writer can write test cases to test his or her specification and perhaps find any lurking problems, but this does not provide any assurances if independently developed grammars are later composed.

8 Current and Future Work

We would like to apply these analyses to other attribute grammars specified using Silver to further evaluate their usefulness. One is possibility is ableJ our extensible specification of Java with many different language extensions [18].

We plan to evaluate the practicality of the circularity analysis. We have not done this on the Silver compiler because Silver is a lazily evaluated language, and our compiler implementation has known "circularities" between attributes that are, in fact, well-defined thanks to laziness. To apply the non-circularity analysis to Silver, we will have to determine whether these can be eliminated or the analysis can be extended to somehow deal with these apparent circularities.

The completeness and circularity detection analyses do not check that an unbounded number of attributable trees will not be created. This is the third component of *well-definedness* identified by Vogt *et al.*in their original work on higher order attribute grammars [23], but dropped as a well-definedness requirement in his Ph.D. dissertation [22]. Other members of our group have separately explored a termination analysis based on term rewriting systems [12].

9 Conclusion

We have presented a *modular* analysis for completeness and non-circularity in attribute grammars. This differs from prior analyses in that languages extensions are checked prior to being composed together, independently of each other, with no checks necessary after composition to ensure these properties. In the extensible language scenario described here we do not have the luxury of a monolithic (composition time) analysis since the person composing the extension grammars is not expected, nor required, to understand attribute grammars, or even know what they are. For extensible languages and extensible language frameworks to be useful to most programmers we believe that these sort of *modular* analyses are critical in order to ensure that the composition of independently developed language extensions "just works."

The analyses do not, and cannot, check that the *forwards* tree on a production is semantically equivalent to the tree doing the forwarding. The language developer is stating, by using the forwarding mechanism that the two are semantically equivalent. If they are not then the attribute values returned from the forwarded-to tree may not be correct. However, a misuse of forwarding in this fashion is a problem with one specific extension, rather than a problem arising from the composition of extensions (though it may only be exposed by a composition of extensions.)

The two questions raised in Section 1 are related to ease of composition of grammars by the non-expert programmer, and not about ease of specification of the language or language extensions by the language developer. The restrictions imposed by the modular analysis are designed to ease the work of the non-expert programmer. That said, we certainly do want to provide as much support as possible to the language developer. The extensions to the analysis to cover all the language features of Silver and to provide new features such as options and attribute defaults are there to make conforming to the restrictions easier. In our experience with Silver, these imposition of the restrictions is more than offset by the strong guarantees that the analyses provide.

References

1. Backhouse, K.: A Functional Semantics of Attribute Grammars. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 142–157. Springer, Heidelberg (2002)
2. Bird, R.S.: Using circular programs to eliminate multiple traversals of data. Acta Informatica 21, 239–250 (1984)
3. Boyland, J.T.: Remote attribute grammars. J. ACM 52(4), 627–687 (2005)

4. Bravenboer, M., Visser, E.: Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In: Proc. of OOPSLA, pp. 365–383. ACM Press (2004)
5. Ekman, T., Hedin, G.: The JastAdd extensible Java compiler. In: Proc. of OOPSLA, pp. 1–18. ACM (2007)
6. Ekman, T., Hedin, G.: The JastAdd system - modular extensible compiler construction. Science of Computer Programming 69, 14–26 (2007)
7. Erdweg, S., Rendel, T., Kästner, C., Ostermann, K.: SugarJ: Library-based syntactic language extensibility. In: Proc. of OOPLSA. ACM (2011)
8. Hedin, G.: Reference attribute grammars. Informatica 24(3), 301–317 (2000)
9. Kaminski, T., Van Wyk, E.: Integrating Attribute Grammar and Functional Programming Language Features. In: Sloane, A., Aßmann, U. (eds.) SLE 2011. LNCS, vol. 6940, pp. 263–282. Springer, Heidelberg (2012)
10. Kaminski, T., Van Wyk, E.: Evaluation of modular completeness analysis on Silver. Tech. Rep. No. 12-024, University of Minnesota, Department of Computer Science and Engineering, `http://melt.cs.umn.edu/pubs/kaminski12tr`
11. Knuth, D.E.: Semantics of context-free languages. Mathematical Systems Theory 2(2), 127–145 (1968); corrections in 5, 95–96 (1971)
12. Krishnan, L., Van Wyk, E.: Termination Analysis for Higher-Order Attribute Grammars. In: Czarnecki, K., Hedin, G. (eds.) SLE 2012. LNCS, vol. 7745, Springer, Heidelberg (2012)
13. Saraiva, J., Swierstra, D.: Generic Attribute Grammars. In: 2nd Workshop on Attribute Grammars and their Applications, pp. 185–204 (1999)
14. Schwerdfeger, A., Van Wyk, E.: Verifiable composition of deterministic grammars. In: Proc. of PLDI. ACM (June 2009)
15. Sloane, A.M.: Lightweight Language Processing in Kiama. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2009. LNCS, vol. 6491, pp. 408–425. Springer, Heidelberg (2011)
16. Van Wyk, E., de Moor, O., Backhouse, K., Kwiatkowski, P.: Forwarding in Attribute Grammars for Modular Language Design. In: Nigel Horspool, R. (ed.) CC 2002. LNCS, vol. 2304, pp. 128–142. Springer, Heidelberg (2002)
17. 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)
18. Van Wyk, E., Krishnan, L., Schwerdfeger, A., Bodin, D.: Attribute Grammar-Based Language Extensions for Java. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 575–599. Springer, Heidelberg (2007)
19. Van Wyk, E., Schwerdfeger, A.: Context-aware scanning for parsing extensible languages. In: Intl. Conf. on Generative Programming and Component Engineering (GPCE). ACM Press (October 2007)
20. Viera, M., Swierstra, S.D., Swierstra, W.: Attribute grammars fly first-class: How to do aspect oriented programming in Haskell. In: Proc. of 2009 International Conference on Functional Programming, ICFP 2009 (2009)
21. Visser, E.: Stratego: A Language for Program Transformation Based on Rewriting Strategies System Description of Stratego 0.5. In: Middeldorp, A. (ed.) RTA 2001. LNCS, vol. 2051, pp. 357–361. Springer, Heidelberg (2001)
22. Vogt, H.: Higher order attribute grammars. Ph.D. thesis, Department of Computer Science, Utrecht University, The Netherlands (1989)
23. Vogt, H., Swierstra, S.D., Kuiper, M.F.: Higher-order attribute grammars. In: ACM Conf. on Prog. Lang. Design and Implementation (PLDI), pp. 131–145 (1989)

Meta-language Support for Type-Safe Access to External Resources

Mark Hills[1], Paul Klint[1,2], and Jurgen J. Vinju[1,2]

[1] Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
[2] INRIA Lille Nord Europe, France

Abstract. Meta-programming applications often require access to heterogeneous sources of information, often from different technological spaces (grammars, models, ontologies, databases), that have specialized ways of defining their respective data schemas. Without direct language support, obtaining typed access to this external, potentially changing, information is a tedious and error-prone engineering task. The Rascal meta-programming language aims to support the import and manipulation of all of these kinds of data in a type-safe manner. The goal is to lower the engineering effort to build new meta programs that combine information about software in unforeseen ways. In this paper we describe built-in language support, so called *resources*, for incorporating external sources of data and their corresponding data-types while maintaining type safety. We demonstrate the applicability of Rascal resources by example, showing resources for RSF files, CSV files, JDBC-accessible SQL databases, and SDF2 grammars. For RSF and CSV files this requires a type inference step, allowing the data in the files to be loaded in a type-safe manner without requiring the type to be declared in advance. For SQL and SDF2 a direct translation from their respective schema languages into Rascal is instead constructed, providing a faithful translation of the declared types or sorts into equivalent types in the Rascal type system. An overview of related work and a discussion conclude the paper.

1 Introduction

Software language engineers, such as those working in the grammarware, modelware, or ontologyware domains, write meta-programs. These programs, used for tasks such as calculating software metrics, performing static analysis, mining software repositories, and building IDEs, use information provided by a wide number of different sources. While some of these are internal to the application, some may be external, e.g., the contents of software repository commit messages, the values in a database storing bug reports, or an already-defined grammar for the language being manipulated. In general, this external information may be created using a number of different tools, each with its own data formats and methods of storing data. These data formats can be defined explicitly, such as with a table definition in a database or an SDF2 [7,16] definition for a grammar, but may also be defined implicitly, such as is found in RSF (Rigi Standard

Format) files, which provide a textual representation of binary relations (see Section 4.1), and in CSV (Comma-Separated Values) files, which provide a textual representation of tabular data.

Without direct language support for accessing external data sources, different libraries have to be developed, each targeted at a different data source. Each library provides a different mechanism for naming these data sources and determining the type of data stored in a resource, including how the external values and types map into the values and types of the host language. Actually writing out these types is a manual task, and synchronization in cases where the underlying types change (e.g., a database table that has been altered, a CSV file with added columns) also must be performed manually. The more sizable and complex data sources described by Bugzilla's entity/relationship (E/R) model are a case in point. This paper is about managing these repetitive and error-prone software language engineering tasks in the context of the Rascal meta-programming language [10].

We describe built-in Rascal language support for incorporating external sources of data (identified by URIs) and their corresponding types: *resources*. Resources allow the meta-programmer to generate type declarations statically, when a resource is imported, before type-checking takes place. This resources feature builds on essential Rascal language features, described in Section 2. This discussion of enabling features is then followed by the main contributions of this paper, given in Sections 3 and 4. Section 3 discusses the design of the Rascal resources language feature, while Section 4 demonstrates the use of resources through four case studies, including the application of resources to typical software engineering tasks. Sections 5 and 6 then close, presenting related work and a final discussion with ideas for future work, respectively.

2 Enabling Rascal Features

To explain Resources, we first discuss four key enabling Rascal features: *type literals* that allow types to be treated as values, *source location literals* that provide access to external resources via Uniform Resource Locators (URIs), *string templates* for code generation, and the *Rascal-to-Java bridge* to connect arbitrary Java libraries to Rascal.

2.1 Type Literals

The Rascal type system provides a uniform framework for both built-in and user-defined types, with the latter defined for both abstract datatypes and grammar non-terminals (also referred to as *concrete* datatypes). A built-in tree datatype (**node**) acts as an umbrella for both abstract and concrete datatypes. The type system is based on a type lattice with **void** at the bottom and **value** at the top (i.e., the supertype of all types). In between are the types for atomic values (**bool**, **int**, **real**, **str**, **loc**, **datetime**), types for tree values (**node** and defined abstract and concrete datatypes), and composite types with typed elements.

Examples of the latter are **list**[**int**], **set**[**int**], **tuple**[**int**,**str**], **rel**[**int**,**str**], and, for a given non-terminal type `Exp`, **map**[`Exp`,**int**]. Sub-typing is always covariant with respect to these typed elements; with functions, as is standard, return types must be covariant, while the argument types are instead contravariant. For example, for sets, **set**[**int**] is a subtype of **set**[**value**], while for functions, **int**(**value**) is a subtype of **value**(**int**).

Formal type parameters allow the definition of generic types and functions. All non-atomic types can have explicit *type parameters*, written either as `&T` or `&T <: Bound`. The former can be bound to any Rascal type, the latter only to subtypes of the type `Bound`. For example, **rel**[`&T`,`&T`] defines a generic binary relation type over elements of the same type, **list**[`&T <:` **num**] defines a list with elements that can only be one of the subtypes of the type **num**, and **list**[`&T`] `reverse(`**list**`[&T] L)` defines the type of a function with the name `reverse` that returns a list with the same element type as its argument `L`.

Reified types make it possible to manipulate types as ordinary values that can be passed around, queried and manipulated. Rascal's reification operator creates *self-describing* type values which contain both the reified type and all datatypes used in this type. A type can be reified using the prefix reification operator (#); we call such an expression a *type literal*. A reified type value contains a symbol to represent the type and a map of definitions for any abstract or concrete datatype dependencies. It is guaranteed to have the type **type**[`&T`], where the type parameter `&T` is bound to the type that was reified. For example:

– **#int** produces a literal value `type(\int(),())` of type `type[`**int**`]`.
– **#rel**[**int**,**str**,**bool**] produces `type(\rel([\int(),\str(), \bool()]),())` of type **type**[**rel**[**int**,**str**,**bool**]].

The **type** data constructor used to build type literals is built in to Rascal; the representations for type symbols and their definitions are defined as Rascal datatypes in a library module. Above, the map of definitions was empty: `()`. For abstract or concrete datatypes this map will contain the complete (possibly recursive) abstract datatype or grammar. Assume a definition for Boolean connectives:

```
data Bool = and(Bool l, Bool r) | t() | f();
```

then the reified type `#Bool` will produce the following term of type **type**[`Bool`] (some details have been elided):

```
type(adt("Bool"),
     (adt("Bool"):choice(...,constructor(adt("Bool"),"and",
          [label("l",adt("Bool")),label("r",adt("Bool"))]),...)))
```

Such self-describing type values are particularly useful in the context of defining resources, where we want to import, and compute types for, otherwise untyped or unknown data from outside of Rascal. Using type literals we can write library functions that bind arbitrary (external) data to specific types.

2.2 Source Locations

Rascal provides built-in support for location literals (values of type **loc**) that are Uniform Resource Identifiers[1] (URIs) optionally followed by text coordinates that allow the identification of specific text ranges in the information the URI points at. Location literals are quoted with bars, such as `|http://www.rascal-mpl.org|`.

In addition to the standard schemes like `file` (local file access) and `http` (remote file access), a number of Rascal-special schemes are supported such as `cwd` (current working directory), `home` (the user's home directory), `std` (the Rascal standard library), `jar` (an entry in a jar file), and `project` (an Eclipse project). The collection of schemes is openly extensible – the extension implements a contribution interface in Java.

The location datatype conveniently provides direct access to parts of the URI and gives short-hands to interact with file systems and web pages. Source locations in Rascal are very versatile and are, for instance, used for tasks such as accessing source code locations in editors and providing hyperlinking functionality in the IDE. In the context of the resources feature, we use them to identify external sources of type information.

2.3 String Templates and Concrete Syntax Templates

Rascal provides both string templates and concrete syntax templates for code generation, a frequently occurring operation in meta-programming. String templates are multi-line string literals with a left-margin, interpolation of arbitrary expressions, auto-indentation, and structured control flow. For example, the following code generates the definition of a Java class named `name` with a number of fields (given as `name`×`type` pairs in relation `fields`), all indented by 2 spaces:

```
str class(str name, rel[str,str] fields) =
  "class <name> {
  '  <for (<f,t> <- fields) {><t> <f>;
  '<}>
  '}";
```

Concrete syntax templates are parsed fragments of code, used for pattern matching and pattern construction. Concrete syntax fragments are supported for languages that have a grammar defined in Rascal. For example:

```
import lang::rascal::syntax::Rascal;

Module m = `module M imports N; ...`;
```

The fragment within the backquotes will be parsed using the grammars defined in the current scope (here, the imported grammar of Rascal). Concrete syntax fragments allow for anti-quoting to expand variables or to match and bind parts using pattern matching. The benefit of concrete syntax fragments is that both generated code and patterns are statically guaranteed to be syntactically correct.

[1] See `http://www.ietf.org/rfc/rfc3986.txt`.

2.4 Rascal-to-Java Bridge

The Rascal-to-Java Bridge makes it possible to call Java functions from Rascal code and to build Rascal data values in Java code. Rascal users can extend their library reusing existing Java code or building on top of the Java standard library. This enables, for example, reuse of JDBC libraries, open Java compilers, SMT solvers, and the Apache Math library. The author of a library written in Java is responsible for producing Rascal data of the right type. Consider the `size` function for lists:

```
@javaClass{org.rascalmpl.library.Prelude}
public java int size(list[&T] lst);
```

The modifier `java` indicates that the function `size` is written in Java and the annotation `javaClass` defines in which class the method `size` can be found. The function is then implemented by the following Java code:

```
public class Prelude {
  IValueFactory vf;
  ...
  IInteger size(IList lst) {
    return vf.integer(lst.length());
  }
}
```

The Java API `IValueFactory` makes it possible to construct arbitrary Rascal values. If the returned type does not match the return type of the associated Rascal function, a run-time type exception will occur, ensuring this mechanism cannot be used to break type safety.

3 Rascal Resources

Given the above motivation and prerequisites, we explain the core contribution of this paper: the design and realization of user-defined resources in Rascal.

3.1 Design of Resources

There are four requirements for resources in Rascal. First, resources should be accessible with a uniform naming scheme. Second, access to resources should be statically typed. Third, the types of resources should be transparently observable to the Rascal programmer. Fourth, where possible, resources should be implemented directly in Rascal. We discuss the first three requirements below.

Uniform Naming Scheme: Many of the libraries for accessing external sources of data in Rascal use their own naming schemes to refer to the sources of the data. For instance, file-based resources tend to use the location of the file, encoded as a Rascal location. By contrast, JDBC resources use JDBC *connect strings*, strings encoding the information needed to connect to the database (host, database, user id, etc), and built either directly as strings or using driver-specific functions that accept the proper parameters. Resources based on data retrieved in JSON format over HTTP may encode the query URL directly.

To obtain a uniform naming scheme, we took as inspiration work on both the Unix [14] and Plan 9[2] operating systems. Unix introduced a major innovation in the handling of I/O by providing a uniform interface for many input and output sources. This allowed special devices, such as terminals, and pseudo-devices to be treated like files, given they provided implementations of operations such as `read` and `write`. Plan 9 took this even further and allowed each resource (including processes and network resources) to have a unique path name to be accessed uniformly. This goal of providing a uniform addressing mechanism can also be realized by the URIs that we already use in Rascal locations, as described above.

It is natural to use Rascal's existing location values to identify and locate external sources of data and types. These external sources introduce new schemes, which bind to functionality for interpreting specific kinds of external data sources (like comma separated values or JDBC data sources). The authority, path and query components serve to identify, unambiguously, which particular source of data is imported and to provide all necessary parameters to do so. We use a `+` sign to split the scheme into a logical and a physical part. Two examples of locations that identify resources are:

- `|csv+file:///Users/foo/projects/data.csv|` uses a CSV resource to access a file in a folder of user foo.
- `|sdf:///languages/ansi-c/syntax/Main|` uses an SDF resource to import a grammar with main module `languages/ansi-c/syntax/Main` (using an implicit search path, see below).

The format of the information in the location is dependent on the resource—the only requirement is that it be representable as a URI.

Access to Resources is Statically Typed: As already discussed, the Rascal type system provides ways to introduce type-checked identifiers, such as names of abstract datatypes and their fields (**data** `Person = person(`**str** `name,` **int** `age)`) and relations with named columns (**rel**`[AST class,` **int** `NCLOC]`). Types not only provide safety, they also provide access via meaningful identifiers to project, select and update parts of datatypes.

Without the resources feature we could already (easily) provide general access to external data. In that case, all imported data is of type **value**. To analyze and manipulate such data, pattern matching is needed extensively, as well as indexing into containers using anonymous "magic" constants. The next best thing is to use type literals (described above). If the client code of a library that imports external data provides a specification of the expected type, then at least the client code can be made type correct and use the appropriate API. For example, we can write: `readCSV(#`**rel**`[`**str** `name,` **int** `age], |home:///people.csv|)` to obtain a typed API to a person/age database stored in a CSV file. The first step

[2] See `http://cm.bell-labs.com/plan9/`.

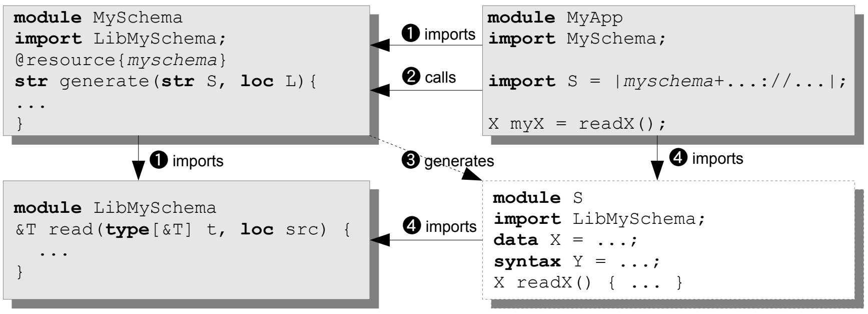
in providing a Rascal resource is to create such a generic library with read functions that are parameterized with the expected return type.

Type literals may solve the type safety issue with external data sources, but the heavy-lifting is in the client code that provides a complete type specification for the external data. For a CSV file with twenty columns, one needs to manually infer a type literal from the CSV file that reflects their types. For an SQL database, one needs to come up with corresponding relation types for every table (imagine a relatively simple situation with 20 tables with 5 columns each). For an external grammar formalism, one needs to port each non-terminal into a Rascal syntax definition. Moreover, all these schemas are subject to evolution and maintenance, leading to cumbersome co-evolution between a data source and its reflection in the Rascal type system. Finally, the semantics of the mapping from one type system to another may require a sizable intellectual effort—recall the "impedance mismatch" between object-oriented (OO) and E/R representations of data.

Our solution to this remaining problem is to apply *code generation* at module import time. One code generator is needed for each kind of foreign data, e.g., CSV or JDBC. Such a code generator is needed in the standard library for each common kind of foreign data, and is reusable for every external data source of this kind. The generated code typically makes use of generic library code accepting type literals as arguments. A designated user-defined code generator generates all required type definitions and interface functions automatically. Since what is generated is standard Rascal code, like a a user would themselves write, the code can then be type checked by the Rascal type checker when the module is loaded, before execution. Public definitions in the generated module provide the module signature used during type checking inside the importing module, ensuring the resource is used safely. Since resources are treated as standard Rascal values, the type checker has the same limitations as with other Rascal code: most type errors are caught statically, while a few (e.g., missing fields on constructor values) are caught using dynamic checks that throw exceptions in error cases.

The reuse of code generators solves the problem of inferring complex type definitions for every new data source as well as their co-evolution. Type-safety is provided as well as an appropriate API. What remains is the specialized, one-time design of a code generator for every new kind of external data. This design influences how many static guarantees can be made and which kinds of co-evolution will go detected. For example, if a CSV resource generator does not support column names, then swapping two columns with the same inferred types in a file will go undetected by Rascal's type system.

Resource Types Are Transparent: The types of external resources can be provided implicitly by way of type-inspection APIs, or explicitly by providing them as complete type definitions. We opt for the latter, since we believe that this is easier to understand for the programmer. By design, the resource generators produce the source code of full Rascal modules, containing all type definitions and access functions. The user can read the generated code like any other Rascal module, debug it if necessary, and use the same IDE support for browsing and



```
module MySchema
import LibMySchema;
@resource{myschema}
str generate(str S, loc L){
...
}
```

```
module MyApp
import MySchema;

import S = |myschema+...://...|;

X myX = readX();
```

```
module LibMySchema
&T read(type[&T] t, loc src) {
  ...
}
```

```
module S
import LibMySchema;
data X = ...;
syntax Y = ...;
X readX() { ... }
```

Fig. 1. Processing the import of an external resource using a URI scheme

querying Rascal modules, yet does not need to maintain it manually. Because of the complexity of the mapping between external and internal types and values, we believe that the ability to interact with the generated source code is an essential engineering prerequisite.

3.2 Syntax and Semantics of Resources

As depicted in Figure 1, to extend Rascal with a new resource type (say `myschema`) the following ingredients are needed:

– A library, e.g. `LibMySchema`, that handles the basic access to and communication with the external resource, using type literals.
– A Rascal module, e.g. `MySchema`, that contains the code generator for the `myschema` resource. The code generator function takes two arguments and is declared to work for a specific schema using a tag: `@resource{myschema}`.
– Rascal client code, e.g. `MyApp`, which imports a to-be-generated resource module using this notation: **import** `S = |myschema+...://...|;` Since it is a `myschema` resource, it starts with `myschema`. This schema name connects the use of a resource with the proper generator that has been defined for that resource.

Given these ingredients, the following steps are executed when the `MyApp` application is loaded (Figure 1):

❶ First all non-resource module dependencies are loaded. This means the `MySchema` module that implements the `myschema` resource is loaded and its declarations are executed. When the declaration of the `generate` function is evaluated, the `@resource` tag triggers the binding of `myschema` to this function. Typically the generator depends on a `LibMySchema` module to get access to the external type information, which is imported now transitively.
❷ Then the resource modules are loaded. For every specific schema used, in this case only `myschema`, its associated generator function is called providing

the name of the module to be generated (`S`) and the user-provided URI (`L`) as arguments. Resource modules are loaded after non-resource modules to ensure that the generator functions have been registered; this restriction could be relaxed, but only at the cost of a more detailed analysis to ensure that all needed generator functions are imported before the resources that use them.

❸ As a result of the call, the code generator uses the information provided in the URI to acquire and manipulate external data sources, eventually generating the source code of a statically correct module with the given name.

❹ Finally, the generated module `S` is imported, like any normal module, by the client application, which may trigger more imports if the generated module imports library modules. Note that generated modules may contain resource imports themselves, which may be useful in cases of importing resources that are themselves structured modularly.

After these steps module `MyApp` can use the freshly created API of module `S`.

4 Sample Resources for Software Language Engineers

When a software language engineer is studying a large software system he/she wants to tap into as many relevant information sources as possible and wants to integrate their contents into a unified result. Examples are bug reports that are stored in a relational database, metrics stored in a spreadsheet, a call graph stored in some textual format, or a complete grammar stored in an external grammar file. In the following sections we demonstrate that all these needs can be addressed by Rascal's resources concept.

4.1 Rigi Standard Format (RSF)

Rigi Standard Format[3] is the main file exchange format for the Rigi system [13] and is used to describe binary relations. Although RSF has been superseded by richer formats such as GXL[4] and GraphML[5] we use it here for its simplicity. An RSF file contains triples of the form `verb subject object` and can simultaneously define several binary relations. The verb part of a triple determines the relation to which the triple contributes. Here is an example:

```
call    main          printf
call    main          listcreate
data    main          FILE
data    listcreate    List
file    listcreate    list.c
lineno  listcreate    10
lineno  main          150
...
```

[3] See `http://www.rigi.cs.uvic.ca/downloads/rigi/doc/node52.html`.
[4] See `http://www.gupro.de/GXL/`.
[5] See `http://graphml.graphdrawing.org/`.

```
@resource{rsf}
public str generate(str moduleName, loc src) {
  map[str, type[value]] rels = getRSFTypes(src);

  return "module <moduleName>
         'import lang::rsf::IO;
         '<for(rname <- rels) {>
         'public <rels[rname]> <rname>() {
         '  return readRSFRelation(#<rels[rname]>, \"<rname>\", <src>);
         '}<}>"';
}
```

Fig. 2. Resource generator for RSF

These triples define the relations `call`, `data`, `file` and `lineno`. The first three will correspond to Rascal relations of type **rel**`[`**str**`,`**str**`]`, while `lineno` will get type **rel**`[`**str**`,`**int**`]`. In order to create support for RSF resources, two steps are needed. First, basic I/O functionality is needed to support the resource format itself. In the case of RSF, we have extended the existing RSF library `lang::rsf::IO` to support resources. There are two essential functions:

– **map**`[`**str**`, `**type**`[`**value**`]] getRSFTypes(`**loc**` src)` returns a map of relation names and their inferred types. By default, all elements in the relations have type `str`, but consistent use of `bool`, `int`, or `real` values at fixed positions in the triples will lead to more precise typing (for instance, with `lineno` above).
– `&T readRSFRelation(`**type**`[&T] result, `**str**` name, `**loc**` src)`[6], given an expected type and a relation name, returns the typed relation with the given name for the RSF resource at location `src`.

Using this, we can create a generator that supports RSF. A simplified version is shown in Figure 2. It essentially processes the given location, extracts the relation names and types from the RSF triples at that location, and uses string templates to generate a Rascal module with declarations for typed functions to extract the various relations from the RSF triples.

Next, we illustrate RSF resources using extracted facts from JHotDraw[7]:

```
CALL    AbstractConnector_2354  Figure_1715
CALL    AbstractConnector_2354  Geom_3544
INHERITANCE AbstractConnector_2354  Connector_1478
CONTAINMENT AbstractConnector_2354  Figure_1715
CALL    AbstractFigure_2788 ChopBoxConnector_2286
...
```

An RSF resource for this data is created by:

```
import RSF;
import JHotDraw52 = |rsf+file:///Users/.../JHotDraw52.rsf|;
```

[6] `&T` could be replaced with **rel**`[&T1,&T2]` to enforce that the return type is a relation; this change will be made in a future version of the code.

[7] See `http://code.google.com/p/crocopat/source/browse/tags/crocopat-2.1.4/examples/projects/JHotDraw52.rsf`.

```
module JHotDraw52
import lang::rsf::IO;

public rel[str, str] CALL() =
  readRSFRelation(#rel[str, str], "CALL", |file:///Users/.../JHotDraw52.rsf|);

public rel[str, str] INHERITANCE() =
      readRSFRelation(#rel[str, str], "INHERITANCE", |file:///Users/.../JHotDraw52.rsf|);

public rel[str, str] CONTAINMENT() =
      readRSFRelation(#rel[str, str], "CONTAINMENT", |file:///Users/.../JHotDraw52.rsf|);
```

Fig. 3. Generated module `JHotDraw52`

This will generate the module `JHotDraw52` shown in Figure 3. Finally, one can use this resource, for instance, by defining a function that reads the `CALL` relation from it (accessible using function `CALL`) and computes its transitive closure:

```
rel[str, str] indirectCalls() = CALL()+;
```

The bottom-line is that Rascal can handle the untyped data in an RSF file in a fully type-safe manner.

4.2 Comma-Separated Values (CSV)

The CSV format was originally intended for exchanging information between spreadsheets and databases but is today used as an exchange format in many other application domains as well. A CSV file has the following structure:

– a header line consisting of field names separated by commas;
– one or more lines consisting of values separated by commas.

The CSV format differs in various respects from the RSF format:

– RSF can define several relations at once; CSV can define only one relation.
– RSF only supports binary relations; CSV supports relations of arbitrary arity.
– The RSF format is fixed; in a CSV file, the header line is optional and the default separator (comma) can be redefined.

The Rascal `lang::csv::IO` library supports the standard CSV format[8] and has been extended to support CSV resources.

The major challenge compared to RSF resources is to handle the variability mentioned above. Our solution is to use the standard query parameters in the URI that describes the location of the CSV data. In the following example we want to process the metrics collected by the Eclipse Metrics Plugin[9]. The relevant data are collected in the file `methods.csv` (a run of the metrics plugin on the source code of the Rascal system itself) in the user's home directory, and we want the function for reading the metrics resource to be named `METHOD_METRICS`:

[8] See `http://tools.ietf.org/html/rfc4180`.
[9] See `http://eclipse-metrics.sourceforge.net/`.

```
module METHODS
import lang::csv::IO;

alias METHOD_METRICSType = rel[str PACKAGE, str TYPE, str METHOD, int LINE, int NOL, int NOS,
                               int FE, int NLS, int NOP, int CC, int LOCm];

public METHOD_METRICSType METHOD_METRICS() =
  return readCSV(#METHOD_METRICSType, |home:///methods.csv|, ());
```

Fig. 4. Generated module `METHODS`

```
  import CSV;
  import METHODS = |csv+home:///methods.csv?funname=METHOD_METRICS|;
```

This will generate the `METHODS` module shown in Figure 4. For convenience, the alias `METHOD_METRICSType` is created as an abbreviation for the actual relation type. The functions in the generated module `METHOD` can, for instance, be used in the following comprehension to compute the methods with the largest cyclomatic complexities (field `CC`):

```
{ <m.PACKAGE, m.METHOD, m.CC> | m <- METHOD_METRICS(), m.CC > 50 };
```

In this particular example, four methods were found, two of which are shown:

```
rel[str, str, int]: {
  <"org.rascalmpl.interpreter","reify",53>, ...
  <"org.rascalmpl.library.vis.util","unPrintableKeyName",59>
}
```

The full power of the relational calculus that is embedded in Rascal can now be used to further explore these metrics data in a type-safe manner.

4.3 Java Database Connectivity (JDBC)

A number of systems, such as Bugzilla, use relational databases to store information useful in language engineering tasks. For instance, information about bug reporters is stored in Bugzilla in a table named `profiles`, which contains data that conforms to the schema shown in Figure 5.

Column	Type	Nullable
userid	mediumint(9)	N
login_name	varchar(255)	N
cryptpassword	varchar(128)	Y
realname	varchar(255)	N
disabledtext	mediumtext	N
mybugslink	tinyint(4)	N
extern_id	varchar(64)	Y
disable_mail	tinyint(4)	N

Fig. 5. Schema for table `profiles`

One popular way to gain access to this information in Java is to use JDBC, a standard Java API for querying, updating, and exploring the metadata of databases. Using Rascal's ability to call Java functions, we have written a JDBC library that allows JDBC calls to be made from within Rascal code. For instance, to connect to a Bugzilla database, select all the records from `profiles`, and then close the connection, the following code would be run:

```
module Profiles
import JDBC;

alias profilesType = rel[int userid, str login_name,
  Nullable[str] cryptpassword, str realname, str disabledtext,
  int mybugslink, Nullable[str] extern_id, int disable_mail];

public profilesType resourceValue() {
  registerJDBCClass("com.mysql.jdbc.Driver");
  con = createConnection("jdbc:mysql://host/bugs?user=usr&password=pass");
  profilesType res = loadTable(#profilesType, con, "profiles");
  closeConnection(con);
  return res;
}
```

Fig. 6. Generated JDBC resource for the `profiles` table

```
registerJDBCClass(mysqlDriver);
con = createConnection("jdbc:mysql://host/bugs?user=usr&password=pass");
res = loadTable(con,"profiles");
closeConnection(con);
```

The first line registers the proper JDBC driver, in this case for MySQL. The second line then actually creates the connection, using a JDBC connect string formatted according to the requirements of the MySQL JDBC driver. The third line loads the data in the table into `res`; since no type information is provided, the data is loaded as a set of values, which can then be de-constructed using pattern matching. Finally, the fourth line closes the connection.

It is possible to instead load a typed representation of the data with `loadTable`, which returns the data in a relation with named fields of the proper type. However, this requires computing the type manually. As was discussed in Section 3, determining the correct type literal is a non-trivial task, here made more difficult by the need to map from native MySQL types, to JDBC types, and then to Rascal types, along with the need to properly account for `null` values (Rascal has no equivalent of `null`, so a datatype `Nullable`, parameterized by the actual column type, is used instead). For this simple table, this process would derive the following type literal representing a row in the table:

```
#tuple[int userid,str login_name,Nullable[str] cryptpassword,str realname,
str disabledtext,int mybugslink,Nullable[str] extern_id,int disable_mail]
```

Two JDBC Resources are currently defined to provide type safe access to JDBC tables. The first, `jdbctable`, provides access to a specific table, while the second, `jdbctables`, provides access to all tables in a database. The following two `import` statements import the `profiles` table and all Bugzilla tables, respectively:

```
import Profiles=|jdbctable+mysql://host/bugs/profiles?user=u&password=p|;
import AllTables=|jdbctables+mysql://host/bugs?user=u&password=p|;
```

The first of the resource imports generates a module, `Profiles`, containing the code shown in Figure 6. The second import creates similar code for each table

```
P = profiles(); B = bugs(); S = bug_status();

deltaDurations = { < p.login_name, b.bug_id,
  createDuration(ts, b.delta_ts).days > | s <- S, s.id==5, p <- P, b <- B,
  b.assigned_to==p.userid, b.bug_status==s.value,
  notnull(ts) := b.creation_ts };

perPerson = { < p,
  ( 0 | it + d | <_,d> <- deltaDurations[p] ) / size(deltaDurations[p]) > |
  p <- deltaDurations<0> };
```

```
rel[str, int]: { <"person1",1>, <"person2",17>, <"person3",7>, ... , <"personN",8> }
```

Fig. 7. Compute average days/person to resolve a bug, using the JDBC resource

in the database, with the table name used to give a name to the function used to retrieve the resource (e.g., `resourceValue` in Figure 6). Using the imported tables, one can then perform queries over the data. For instance, one may want to find the average number of days, per person, it takes from when a bug is assigned to when it is resolved[10]. This is done using the code shown in Figure 7.

The first line in Figure 7 extracts the relations stored in tables `profiles`, `bugs`, and `bug_status` into variables `P`, `B`, and `S`, respectively. Relation `deltaDurations` is then created using a comprehension, which enumerates all bug statuses; filters these to only include tuples with `id` 5 (status "resolved" in this Bugzilla database); enumerates all profiles; enumerates all bugs; and filters the bugs to include only those bugs assigned to the user represented by the profile, with a status the same as the current status, and with a non-null creation timestamp (checked using pattern matching, with a non-null timestamp represented as timestamp `ts` wrapped in the `notnull` constructor). For each matching combination of status, profile, and bug (after accounting for all the conditions just mentioned), a tuple is added to the computed relation containing the login name of the profile, the id of the bug, and the number of days between the creation timestamp and the timestamp of the last change to the bug information, which we assume here represents the date when the bug was resolved – i.e., a relation between logins, bug ids, and days to resolve the bug. `perPerson` is built in a similar way: for each login name `p`, the days related to `p` are summed, with the result divided by the number of records to compute the average. The resulting relation is shown, in part, in Figure 7 below the code, with login names made anonymous.

4.4 Syntax Definition Formalism

The Syntax Definition Formalism (SDF) [7,16] is an EBNF-like grammar formalism extended with disambiguation constructs. It is generally used to define both the concrete and the abstract syntax of software languages in the same definition. There are many open-source grammars available written in SDF[11]. These

[10] Technically, this shows the average from the creation date to the delta date, which is the date of the last change to the bug information.

[11] See `http://www.syntax-definition.org`.

```
@resource{sdf}
public str generate(str name, loc at) {
   def = loadSDF2Module(at.host, [|rascal:///|]);
   gr = fuse(sdf2grammar(name, def));
   return "module <name>
          '
          '<grammar2rascal(gr)>
          ";
}
```

Fig. 8. Resource generator for SDF

grammars are complex engineering artifacts [9], especially if they are written with the intention to generate a syntactically and semantically correct parser. Since it is appealing to reuse such SDF grammars, we have implemented an SDF resource that can, for instance, import an SDF syntax definition of `Java5` into a Rascal module as follows:

```
import lang::sdf2::utils::Resource;
import Java5 = |sdf://languages/java/syntax/Java5|;
```

The resource generator is implemented as shown in Figure 8. This implementation deserves some explanation:

– We use the search path for Rascal modules to search for SDF files. The URI `|rascal:///|` represents the root of the entire Rascal search path which is passed to a function that will traverse all the imports of a modular SDF specification and produce a single syntax tree listing all relevant modules. Adding an SDF grammar to any Rascal project in the Eclipse workspace will make it available for use.
– Since SDF's module system has an entirely different semantics from Rascal's module system, modules in SDF can not map to modules in Rascal. The *fuse* function flattens the internal grammar such that SDF's module semantics are implemented[12].
– The `sdf2grammar` function (800 LOC in Rascal), in particular, attempts to maintain the semantics of SDF's disambiguation features. Bouwers et al. have described some of the intricacies of the semantics of disambiguation [3]. Some disambiguation features in SDF are more powerful than their counterparts in Rascal. Their semantics have been limited in the design of Rascal to make them easier to understand and debug. At the same time, Rascal has additional disambiguation features that can replace the earlier "mis-uses" of the power of SDF. The translation is intentionally *not* complete, such that features that do not map are documented in the resulting Rascal grammar. `sdf2grammar` is not fast, it was written for brevity and clarity first, with optimization as a later goal.

The complexity of such a translation from one EBNF-based formalism to another may be daunting, but being able to reuse it transparently via the resource feature

[12] SDF's renaming and module parameter features are not yet implemented.

adds all the more value. It is particularly useful to be able to read the source code of the resulting Rascal module like any normal module. We also expect that further maintenance may take place in the generated Rascal modules. This can be achieved by replacing the above import of the Java5 resource by a direct import of the generated module `Java5`.

5 Related Work

Rascal resources fit in the field of interfacing data sources and programming languages. A large amount of related work exists on interfaces (libraries, code transformations, language extensions) for accessing external (especially database) resources. Because of space, we only discuss the most directly related work.

5.1 Scripting Languages

Scripting languages, like Rascal, often serve as "glue" between systems that need to be combined. Rascal has a static type system, while most other scripting languages (Python, Ruby, Perl) have dynamic type systems. Such languages have the advantage that data conversion between two systems can be limited to the shape of the data and not much time has to be spent on bridging type systems. Dynamically typed languages serve well as glue because they pose no a priori, static, limitations on the kind of data that can be processed. Our resources concept is unnecessary for dynamically typed languages, since there is no static type system to use in the first place.

One can use Rascal as a mostly dynamically typed language when no optional type declarations are used and all data is simply of type **value**, **list**[**value**], **set**[**value**] or **node**. XML documents can, for instance, be represented in this way. In this style, one has to use pattern matching to analyze and transform these untyped data structures, effectively encoding the type system into the program.

Compared to this dynamically typed programming model using pattern matching, Rascal resources are at the other end of the spectrum and use external datatype definitions, bringing the external data into the typed world. Note that the generated code should be type-safe unless the underlying schema has changed since the last generation of the interface. In this unusual case, Rascal produces a run-time type error if the schema change results in a type change.

5.2 Object/Relational Mapping

There have been many attempts, in many language paradigms, at solving the *impedance mismatch* [11] between the representation of data in a relational database and in program values. This includes PLAIN [15], a Pascal-like programming language extended with statements to query a database, and the Microsoft ADO.Net Entity Framework [12], a data access framework for Microsoft's .Net platform. [5] provides a comprehensive overview of the problems involved in integrating programming languages and databases.

In our work on Rascal resources we have not focused specifically on access to databases, and we have not yet attempted to optimize access, instead working with a model where all data of interest is loaded into Rascal and then manipulated using Rascal expressions (see Figure 7 for an example). However, there is nothing inherent to our solution that would prevent this optimization. Since we are most interested in working with existing data, we also do not yet support writing to databases, and would need a stateful interaction library, such as that developed for working with Maude-based analysis tools [8], to do so efficiently.

5.3 XML Binding and AST Generators

The problem of being able to use typed interfaces on data that is serialized in an untyped or otherwise foreign notation can be found in many places. In the XML domain, this is called *binding'* and generation of typed interfaces from XSD or DTD schemas is common practice [2,4,1]. In the (compiler) front-end domain we, and many others, have generated APIs for abstract syntax trees from grammars [6]. This is also a kind of data-binding.

5.4 LINQ

LINQ provides language support for type-safe SQL-like query syntax on external data-sources. By implementing a Provider, library authors can relatively easily add support for foreign data representations. A LINQ provider may even examine the syntax tree of a query and decide how to implement it.

Rascal resources share a similar goal, but the design is quite different. Instead of enabling an adapter to access the remote data, Rascal resources are about transforming external data into local data representations. LINQ provides, like Rascal, guarantees for type safety of the client code.

5.5 F$^{\#}$3.0 Type Providers

Rascal's resources also resemble the feature of *Type Providers* in F$^{\#}$3.0[13]. We briefly discuss commonalities and some significant differences.

Type Providers in F$^{\#}$provide a hook into the type system. The user may declare, using source code annotations, certain extensions to the type system, which may manipulate the set of declared types. Implementing a Type Provider entails the implementation of an interface to produce literal representations of new types, new properties and new methods. The net effect is that after a programmer has declared the intention to use a certain kind of type provider, at every use site of the generated types the provider mechanism will query external data sources for type information and bind new type names "dynamically" at type-checking time to the relevant scope in the F$^{\#}$program.

[13] See `http://msdn.microsoft.com/en-us/library/hh156509(v=vs.110).aspx`.

Granularity. A Type Provider is a module (dll) which provides functions to add types to a relevant scope. These functions may be parameterized, for example by a location URI, such that the generated types are specific for the call site. Rascal's resources are generators of modules, which statically contain all the type information that is present via a statically known URI. They are "module providers" rather than type providers and thus have a larger granularity. They will bind the new type names for the entire module that imports the resource.

Dynamic & Lazy versus Static & Eager. By design, Type Providers make every change in external definitions of schemas immediately visible, and by using dynamic and lazy retrieval of type information exploratory programming[13] can be supported. Still, if a schema changes the user does need to rebuild the F$^{\#}$project[13]. Rascal's design requires a regeneration of the type definitions, which it does every time an importing module is (re)loaded. It does so eagerly—a resource must produce a complete set of type definitions for the data that will later be loaded dynamically. For large database schemas the Rascal programmer would have to wait until an entire E/R schema is translated or individually select which tables to import, while in F$^{\#}$the schema can be explored on a table-by-table basis. However, the Rascal programmer always gets a complete overview of the available types as a literal Rascal program, while the F$^{\#}$programmer would need to exercise the generated structure to discover these types, using IDE features such as auto-completion.

Definition. F$^{\#}$'s type providers are written in F$^{\#}$. A new type provider consists of functions that return lists of type literals. To implement a type provider, the programmer needs access to and understanding of F$^{\#}$'s reflection API. The construction of the types is type safe, calls to the reflection API are type-checked and the type-correctness of the generated type definitions can be guaranteed.

For Rascal resources, we currently use string templates to generate a new Rascal module. The *generating* program thus has no static guarantees of correctness regarding the generated program, while the *generated* program is fully type-checked before execution. Note that both Rascal and F$^{\#}$use type parameterized type literals to link generic data to specific types.

Example. For comparison, Figure 9 shows Rascal code for a resource generator that produces a set of 100 datatypes, each with three kinds of constructors of which one has 100 fields and 2 functions defined on each type. This example mimics precisely the example F$^{\#}$type provider called `HelloWorldTypeProvider`.[14]

Rascal's and F$^{\#}$'s type declarations are not entirely compatible, but the two example snippets generate definitions of the same size and complexity. The size of the F$^{\#}$example is much larger (67 non-commented non-empty lines), compared to Rascal (14). F$^{\#}$also needs 5 library modules to access the reflection API and the Type Provider API, while Rascal needs no API. The benefit of F$^{\#}$'s added explicitness is the type-safety that Rascal does not guarantee at generation time, while Rascal's solution is simpler to understand (we claim).

[14] See `http://msdn.microsoft.com/en-us/library/hh361034(v=vs.110)`.

```
module HelloWorldResource
@resource{helloworld}
str generate(loc uri, str name)
  = "module <name>
    '<for (i <- [1..100]) {>
    '@doc{This is an example generated type definition}
    'data Type<i> = unit_<i>()
    '             | data_<i>(str x)
    '             | nested_<i>(str sp_1, <for (j <- [2..100]) {>, str sp_<j><}>)
    '             ;
    '@doc{This computes some property}
    'int property(Type<i> arg) = ...;
    '@doc{This computes some function}
    'str method(Type<i> arg, int i) = ..."
    '<}>";
```

Fig. 9. The Rascal Resource generator to mimic F$^{\#}$'s HelloWorldTypeProvider

Rascal does, however, support the construction of statically syntax correct modules using its concrete syntax feature. For example:

```
@resource{hello}
Module generator(loc uri, str name) = `module <[Name] name> ...`;
```

5.6 OData

Rascal resources share some of the same goals as the Open Data Protocol[15], or OData. OData also uses URIs to identify resources, and is focused on providing a standard interface to resource data, but is a protocol, not a programming language mechanism. In the future, an `OData` Rascal resource would provide a clean way for Rascal to access information shared using the OData standard.

6 Discussion

We have presented the design and implementation of typed, uniform access to external resources in Rascal. This brings the flexibility of managing external datatypes in dynamically typed languages to the world of statically typed languages. Although we believe that the examples in Section 4 demonstrate that we have made good progress towards satisfying the requirements listed earlier in Section 3, some comments are in order.

First, it could be beneficial to check the static safety of the generated code *at generation time*, instead of as part of the module load process. Currently, generation can (if using concrete syntax) only guarantee that the generated code is syntactically correct. This would require invoking the checker as part of the generation process. Second, we have focused so far on resources that can easily be read fully into Rascal and manipulated using Rascal code. For larger resources, this is not practical, and we instead would need a method of gradually reading in resource

[15] See `http://www.odata.org/`.

data. Along with this, it would be useful to leverage the optimization and search capabilities of external systems. For instance, the not-null check shown in the JDBC example in Section 4 is currently performed using pattern matching in Rascal, but could also be done directly by the database. Third, we have focused mainly on reading resources, and would like to provide better support for writing data back to external data sources after making changes within Rascal.

We intend to further explore and extend the possibilities of resources for other resource types that are relevant for the software language engineer.

Acknowledgments. We would like to thank the anonymous reviewers, whose helpful feedback has assisted us in improving the quality of this paper.

References

1. The Enhydra Project: The Zeus Java-to-XML Data Binding tool (2002), http://zeus.ow2.org/
2. The ExoLab Group: Castor (2002), http://www.castor.org
3. Bouwers, E., Bravenboer, M., Visser, E.: Grammar Engineering Support for Precedence Rule Recovery and Compatibility Checking. In: Proceedings of LDTA 2007. ENTCS, vol. 203, pp. 85–101. Elsevier (2008)
4. XML/Java Data Binding and Breeze XML Binder. Technical report, The Breeze Factor (2002), http://www.breezefactor.com/whitepapers.html
5. Cook, W., Ibrahim, A.: Integrating Programming Languages and Databases: What is the Problem? ODBMS.ORG (September 2006) Expert Article
6. de Jong, H., Olivier, P.: Generation of abstract programming interfaces from syntax definitions. Journal of Logic and Algebraic Programming 59, 35–61 (2004)
7. Heering, J., Hendriks, P., Klint, P., Rekers, J.: The syntax definition formalism SDF - reference manual. SIGPLAN Notices 24(11), 43–75 (1989)
8. Hills, M., Klint, P., Vinju, J.J.: RLSRunner: Linking Rascal with K for Program Analysis. In: Sloane, A., Aßmann, U. (eds.) SLE 2011. LNCS, vol. 6940, pp. 344–353. Springer, Heidelberg (2012)
9. Klint, P., Lämmel, R., Verhoef, C.: Toward an engineering discipline for Grammarware. ACM TOSEM 14(3), 331–380 (2005)
10. Klint, P., van der Storm, T., Vinju, J.: EASY Meta-programming with Rascal. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2009. LNCS, vol. 6491, pp. 222–289. Springer, Heidelberg (2011)
11. Maier, D.: Representing Database Programs as Objects. In: Advances in Database Programming Languages, Papers from DBPL-1, Roscoff, France, pp. 377–386 (September 1987)
12. Melnik, S., Adya, A., Bernstein, P.A.: Compiling Mappings to Bridge Applications and Databases. In: Proceedings of SIGMOD 2007, pp. 461–472. ACM (2007)
13. Müller, H., Klashinsky, K.: Rigi - A System for Programming-in-the-Large. In: Proceedings of ICSE 10, pp. 80–86 (April 1988)
14. Ritchie, D., Thompson, K.: The UNIX Time-Sharing System. The Bell System Technical Journal 57(6) (July-August 1978)
15. van de Riet, R.P., Wasserman, A.I., Kersten, M.L., de Jonge, W.: High-Level Programming Features for Improving the Efficiency of a Relational Database System. ACM Trans. Database Syst. 6(3), 464–485 (1981)
16. Visser, E.: Syntax Definition for Language Prototyping. PhD thesis, University of Amsterdam (1997)

Approaches and Tools for Implementing Type Systems in Xtext

Lorenzo Bettini[1], Dietmar Stoll[2], Markus Völter[3], and Serano Colameo[2]

[1] Dipartimento di Informatica, Università di Torino
[2] itemis Schweiz GmbH
[3] independent/itemis AG

Abstract. With the Xtext framework, building domain specific languages (DSLs) integrated into the Eclipse IDE has become increasingly popular and viable even for non-trivial domains. However, sophisticated DSLs may require advanced type checking capabilities, since they usually include expressions, types and the notion of type conformance. In this paper we compare a number of approaches and frameworks for implementing type systems for Xtext languages regarding flexibility, required effort and usability. We use a common case study to illustrate the trade-offs between the various tools.

1 Introduction

Developing languages is time consuming: it involves defining grammars, parsers, constraints, type systems and generators or interpreters, as well as an IDE. With the rising popularity of domain-specific languages (DSLs), there is an increasing need for better tool support for building languages and their IDEs. Traditional parser generators (such as Flex/Bison [25] or ANTLR [28]) help only with the concrete and abstract syntax of a language. Modern language workbenches (such as Xtext [3], MPS [37] or Spoofax [24]) generate editors, complete with syntax highlighting, code completion and static error highlighting, based on a grammar definition. They significantly reduce the effort for implementing a language and its IDE.

Judging by the uptake in industry, Xtext [3] is currently one of the most popular language workbenches. It is based on Eclipse, and, from an enriched grammar, generates an Eclipse-based editor, a parser and an abstract syntax tree based on EMF Ecore [35]. It also provides APIs and languages for defining scopes, constraint checks and generators as well as additional IDE aspects that cannot be automatically derived from the grammar. *Type checking* is a specific case of constraint checking. In order to check if a type is valid, that type (and all other types in the program) first has to be calculated. For non-trivial languages, type calculation can be quite elaborate.

In Xtext, both constraints and transformations or generators are by default implemented in Java (or Xtend [2], a modernized, Java-like language that comes with Xtext). However, Xtext provides no specific support for defining *type systems*, beyond implementing typing rules in Java/Xtend as part of the validation. As languages become more sophisticated, there is a need for dedicated support for implementing type systems that lets users express typing rules concisely. The main feature of such type system implementation tools should be the calculation of the type of program elements. They should also integrate with the constraint checking facilities to perform the actual type checks. More specifically, a type system definition framework has to support the following features, in addition to the obvious requirement of being able to calculate the types for all typed language concepts. The framework must . . .

F1 allow to express typing rules in a concise way,
F2 support common cases for typing rules (e.g., assignment of fixed types to elements, the derivation of the type of one element from the type of one of its properties and the computation of common (super-)types), but not prevent the user from implementing manually more sophisticated (corner) cases,
F3 report type specifications that are incomplete or refer to AST concepts that do not exist (or have been deleted after a typing rule has been specified),
F4 integrate with the validation framework of the language development tool so typing errors can be reported together with other constraint errors,
F5 support test and debugging of typing rules.

Note that in the context of this paper, type calculation and checking does not include name resolution (linking and scoping in Xtext terminology) or simple constraint checking. Both are supported reasonably well by writing Xtend code (which supports, for instance, functional abstractions).

Contribution. Our primary contribution is a comparison between various approaches for implementing type systems for Xtext-based languages. We start out by showing how to implement type systems with Xtext only and then compare three alternative approaches: Xsemantics, the Xtext Type System (XTS) and the type system introduced by Xbase. Our secondary contribution is a brief look at other language workbenches and the ways they defined type systems for DSLs.

The paper is structured as follows: in Section 2 we introduce a case study that is used to compare the various type system definition approaches discussed in this paper. In Section 3 we then present the implementation of the type system for this language using the default way suggested by Xtext: an implementation in plain Java/Xtend. We then illustrate three alternative approaches. In Section 4 we will show how to integrate a DSL type system with Xbase, a reusable expression language, integrated with Java, and shipped with Xtext. We then implement the type system for our case study in Xsemantics (Section 5) and in XTS (Section 6), two DSLs for implementing type systems for Xtext-based languages. In Section 7 we evaluate and compare these approaches based on their support for the features introduced above. We also provide recommendations on when to use which approach. We conclude the paper with a review of related work (Section 8). The code of all examples in this paper is available at `https://github.com/markusvoelter/typesystemcomparison`.

2 Case Study

The comparison of the various tools for type system definition are based on the example language discussed in this section. It is a language for describing data entities and GUI forms to edit them. The example language is not intended to have all features relevant in practice: it is only used as a case study to illustrate the implementation of type systems, and we will concentrate on the features that are more interesting to this end.

Fig. 1 shows an overview over the abstract syntax of the language. An `Entity` can have `Attributes` of primitive types such as `boolean`, `string`, `int` and `float`, or `EntityTypes`, which represent the type of an `Entity`. `Attributes` can have a type and an initialization expression. If both the type and the init expression are specified, we require that the (inferred) type of the expression is a subtype of the `Attribute`'s type. If no `Attribute` type is specified, then the initialization expression is mandatory and used to infer the type of the `Attribute`. Entities can specify a base entity they extend, so there is a subtyping relationship on `EntityTypes` implied by the transitive closure of the `extends` relation.

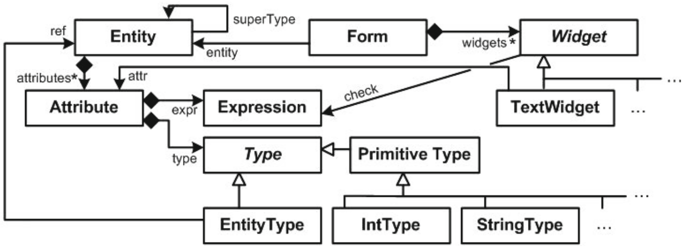


Fig. 1. The essential parts of the meta model (AST) generated from the grammar

A `Form` references an `Entity` and owns widgets such as text fields and checkboxes. Widgets refer to a specific attribute of the entity. Widgets may also contain a `check` clause, which verifies the content of the widget (for example, the length of the input). In check expressions one can access the current content of the widget using the `wid-getcontent` expression, which will must be typed to the type of the corresponding attribute. Listing 1 shows an example program with a `Person` entity and a `PersonForm` to edit it. Note that the attribute `isAdult` has an explicit type, and its initialization expression conforms to that type; *greeting* has no explicit type, and its type is inferred from its initialization expression.

The part of the example grammar that is used by all variants of type system implementation is shown in Listing 2. The `Type` and `Expression` part of the grammar is the same for the the plain Xtext grammar and the Xsemantics and Xtext/TS grammars (for lack of space we do not show the complete grammar of expressions). In the Xbase scenario, the rule `Expression` is replaced with the Xbase's `XExpression`.

```
entity Person {
  name      : string;
  firstName : string;
  age       : int;
  weight    : float;
  isAdult   : bool = age > 18;
  greeting  = "Hello " + firstName
            + " " + name + "!";
}
```

```
form PersonForm edits Person {
  text(20) -> name check
           lengthOf(widgetcontent) >= 2;
  text(20) -> firstName;
  text(5)  -> age check 12.2>widgetcontent;
  text(5)  -> weight check 0<widgetcontent;
  checkbox -> isAdult;
  text(30) -> greeting;
}
```

Listing 1. Forms and Entities DSL

```
Model: (entities+=Entity | forms+=Form )*;
Entity: "entity" name=ID ('extends' superType=[Entity])? "{" (attributes+=Attribute)* "}";
Attribute: name=ID ( ((":" type=Type)? "=" expr=Expression) | (":" type=Type) )";";
Form: "form" name=ID "edits" entity=[Entity] "{" (widgets+=Widget)* "}";
Widget: TextWidget | CheckBoxWidget;
TextWidget: "text" "("length=Number")" "->" attr=[Attribute] ("check" check=Expression)?";";
CheckBoxWidget: "checkbox" "->" attr=[Attribute] ("check" check=Expression)? ";" ;

Type: PrimitiveType | EntityType;
EntityType: ref=[Entity];
PrimitiveType: NumberType | BooleanType | StringType;
NumberType: FloatType | IntType;
FloatType: {FloatType} "float";
IntType: {IntType} "int";
BooleanType: {BooleanType} "bool";
StringType: {StringType} "string";

Expression: BooleanExpression;
/* skipped full Expression grammar */
Atomic returns Expression: '(' Expression ')' | {WidgetContent} "widgetcontent" |
   {LengthOf} "lengthOf" "(" expr=Expression ")" | {EntityType} "new" ref=[Entity] |
   {BooleanLiteral} value=("true"|"false") | {FloatLiteral} value=Float |
   {IntLiteral} value=INT | {StringLiteral} value=STRING | {AttributeRef} attr=[Attribute];
```

Listing 2. Grammar of case study DSL

Xtext grammars are essentially EBNF with the following differences. For each grammar rule, an EMF `EClass` is generated in the meta model (e.g., there will be a metaclass `TextWidget`). The grammar specifies property names, so they can be generated into the metaclass (`TextWidget` will have a property `length`). Finally, Xtext grammars support expressing references natively. The `attr=[Attribute]` syntax in the `TextWidget` rule expresses that in the generated `EClass` there will be a property `attr` that is a non-containing reference to `Attribute`. Scoping rules (to be written separately) determine which instances of `Attribute` are valid targets for the reference. Also, constraints and typing rules have to be expressed separately. This paper is about expressing the typing rules efficiently.

To compare the different type system variants, we implement the following checks in each of them: the `Attribute`'s initialization expression (if present) must conform to the `Attribute`'s declared type (if specified); the check expression must be `boolean`; the text widgets must not refer to `boolean Entity` attributes; the checkbox widgets must refer only to `boolean` attributes. The `+` operator can be used both for arithmetic addition (in this case its type is numeric) and as string concatenation: if one of the operands is a `string`, the type of `+` will be `string`.

3 Plain Xtext

The plain Xtext implementation requires the implementation of three kinds of operations: one to calculate the actual type of an expression, one to calculate the expected type (which depends on the context in which the expression is used), and one that checks whether one type is assignable to another. The first two are implemented by

```
 1  class GuiDslTypeProvider {
 2     @Inject extension TypeConformance conformance // Xtend "extension" and google @Inject
 3     Type _bool = GuiDslFactory::eINSTANCE.createBooleanType // local var for easy use
 4     Type _float = GuiDslFactory::eINSTANCE.createFloatType // local var for easy use
 5     @Inject CyclicDependencyType cyclicType // ... similar for other basic types
 6
 7    def Type getType(EObject e) { getType(e, newHashSet()) }
 8    def Type getType(EObject e, Collection<EObject> visited) {
 9      if (visited.contains(e)) return cyclicType else visited.add(e)
10      switch e {
11        Widget : e.attr.getType(visited)
12        Attribute case e.expr != null && e.type != null
13           && e.type.isAssignable(e.expr.getType(visited)) : e.type
14        Attribute case e.expr != null : e.expr.getType(visited)
15        Attribute case e.type != null : e.type
16        AttributeRef : e.attr.getType(visited)
17        AndOrExpression : _bool
18        Comparison : _bool
19        // type is the most general, e.g. int + float => float
20        Plus : mostGeneral(e.left.getType(visited), e.right.getType(visited))
21        Minus : mostGeneral(e.left.getType(visited), e.right.getType(visited))
22        // ... similar for other expressions
23        default: null
24    } }
25    def Type getExpectedType(EObject e) { internalGetExpectedType(e.eContainer, e.
          eContainingFeature) } ...
```

Listing 3. Type provider in Xtend

GuiDslTypeProvider[1], partially shown in Listing 3. The method getType(EObject expr) determines the actual type of an expression expr. It avoids endless loops in case of cyclic dependencies (typically due to a malformed cyclic entity hierarchy) by caching already visited elements (line 9). The type of primitive types BooleanType as well as the type of an EntityType (see grammar in Listing 2) is defined to be itself (not shown). The type of a Comparison is always boolean (line 18), there is no need for recursive computation. The method getExpectedType() expects that the elements in a Comparison container are boolean as well.

The type of an Attribute (lines 12-15) is the type of its type feature, if present. If there is only an init expression, its type is taken. If there is both a type feature and a conforming init expression, the type of the former is taken. If the init expression is non-conformant, getType() returns the type of the init expression, while getExpectedType() will return the type of the Attribute's type feature. This will lead to an intended type error when the validator checks expected types against actual types. References to attributes (line 16) have the type of the referenced attribute.

The operation getExpectedType() in Listing 3 (line 25) returns the expected type of an EObject by checking its container. It calls another operation (not shown here) with the container of the EObject and the feature of the container that contains itself. For instance, if the container of an expression is a Widget and the feature of the widget containing the expression is the check clause, the expected type is boolean. An expression is also expected to be boolean if its container is a Comparison. However,

[1] This class is written in Xtend [2]. Xtend compiles to Java and provides functional abstractions (useful for interpreters and AST navigation), templates (for code generators), and syntactic sugar like extension methods and a concise getter/setter syntax. For instance, isAssignable(a, b) may be written as a.isAssignable(b), and a.getType() as a.type.

```
 1  class TypeConformance {
 2    def dispatch isAssignable(Type left, Type right) {
 3      left.eClass == right.eClass || right.eClass.EAllSuperTypes.contains(left.eClass) }
 4    def dispatch isAssignable(EntityType left, EntityType right) {
 5      internalIsAssignable(left.ref, right.ref, newHashSet() }
 6    def internalIsAssignable(Entity left, Entity right, Collection<Entity> visited) {
 7      if (visited.contains(right)) return false; // cycle detected
 8      visited.add(right)
 9      left == right || (right.superType != null && internalIsAssignable(left, right.superType,
            visited) ) }
10    def dispatch isAssignable(FloatType left, IntType right) { true }
11    def dispatch isAssignable(StringType left, NumberType right) { true }
```

Listing 4. Type conformance specification (Xtend code)

```
public class GuiDslJavaValidator extends AbstractGuiDslJavaValidator {
  @Check // slightly simplified code for publication
  def checkType(Expression object) {
   val expectedType = object.getExpectedType
   val actualType = object.getType
   if (expectedType == null) return;
   if (!expectedType.isAssignable(actualType)) {
     error("Type error, expected '" + expectedType + "' but was '" + actualType + "'");
  // ... other validation methods
}
```

Listing 5. Xtext validator

if an expression is contained in a `-`, `*` or `/`, it is expected to be of a `NumberType`. For a `+`, a `string` is expected, unless there is a common type of the arguments that is more specific. The method returns `null` to indicate that there is no expected type.

The `GuiDslTypeProvider` uses the Xtend code in Listing 4 to compute whether another type can be provided where a certain type is expected, also known as type conformance or subtyping, which is an important part of a type system. The method `isAssignable(left, right)` returns `true` if an element of type `right` can be used where an element of type `left` is expected. Thanks to polymorphic dispatch, a call to `isAssignable()` will be dispatched to the method having the most specific parameter types given the run-time types of the arguments. The first method (line 2) accepts `Types` and reuses the type hierarchy of the EMF metamodel generated from the DSL grammar. For instance, the EMF metamodel class `FloatType` subclasses `NumberType`, and thus `eClass().getEAllSupertypes()` called on a `FloatType` instance contains `NumberType` which in turn means that `FloatType` can be used where `NumberType` is expected. The methods in lines 4 to 9 specify that one `EntityType` is a subtype of another one if the contained `Entity` (e.g. `Person`) are the same, or if the first is a supertype of the second (e.g. `Teacher extends Person`). Here, the reference `superType` is not an EMF reference, but the one specified in the grammar for `Entity`. As in Listing 3, the visited entities are cached to avoid endless loops in case of cyclic hierarchies. The other methods deal with the remaining special cases.

Finally, the validator in Listing 5 uses the classes shown above to implement the checks on the widgets (see Section 2). A method annotated with `@Check` is called by Xtext for every model element that fits the parameter type. In the example, the actual and expected types of expressions are checked for assignability (using concise Xtend syntax). In case there is no expected type, e.g. for an attribute whose type is only defined by the derivation expression, there is no need to issue an error.

4 Xbase

To leverage type checking and scoping of Xbase in this scenario, the `Expression` rule in the grammar is replaced with the Xbase `XExpression`, which allows to write a specification on how Xbase should generate Java classes from *Entities* and *Forms*. Listing 6 shows the complete code (without comments) necessary to let Xbase automatically generate Java classes for entities and forms. For each `Entity`, a Java class is inferred with supertype references (line 8), getters (line 10), fields (line 12) and setters (line 13).

Also, for each `Widget` of a `Form`, a method with return type *boolean* is added to the `Form`'s Java class, whose name is derived from the attribute's name the widget refers to (line 20). Of course, the method can only be inferred if there was an attribute and a check clause defined for the widget in the DSL file (line 19).

```
1  class XGuiDslJvmModelInferrer extends AbstractModelInferrer {
2      @Inject extension JvmTypesBuilder // Xtend "extension" keyword
3      @Inject extension IQualifiedNameProvider
4      @Inject extension GuiTypeProvider guiTypeProvider
5
6      def dispatch void infer(Entity element, IJvmDeclaredTypeAcceptor acceptor, boolean
           preIndexingPhase) {
7          acceptor.accept(element.toClass(element.fullyQualifiedName)).initializeLater [
8              if (element.superType != null) superTypes += element.superType.cloneWithProxies
9              for (a : element.attributes) {
10                 val getter = a.toGetter(a.name, a.getJvmType)
11                 if (a.expr != null) { getter.body = a.expr } else {
12                     members += a.toField(a.name, a.getJvmType)
13                     members += a.toSetter(a.name, a.getJvmType) }
14                 members += getter
15             }]}
16
17     def dispatch void infer(Form form, IJvmDeclaredTypeAcceptor acceptor, boolean
           preIndexingPhase) {
18         acceptor.accept(form.toClass(form.fullyQualifiedName)).initializeLater [
19             form.widgets.filter[check != null && attr != null].forEach[ w |
20                 members += w.toMethod('check'+w.attr.name.toFirstUpper, form.newTypeRef(
                       Boolean::TYPE)) [
21                     parameters += w.toParameter('widgetcontent', w.attr.getJvmType)
22                     body = w.check
23                 ]]]} ...
```

Listing 6. JvmModelInferrer written in Xtend

With this inference specification, Xbase will take care of type checking the `check` clause of a `Widget`. The case study also requires that the keyword `widgetcontent` in the `check` clause must refer to the attribute the widget refers to. This is done in Xbase by adding a parameter `widgetcontent` with the attribute's type (line 21).

The validator in the Xbase scenario is shown in Listing 7. The validation methods are similar to the plain Xtext scenario (e.g. for a `TextWidget`, line 3), but now `JvmTypes` are compared (line 6), which are supplied by a type provider (line 2) which delegates to the built-in Xbase type provider.

```
class XGuiDslJavaValidator extends XbaseJavaValidator { // google guice @Inject
  @Inject private extension GuiTypeProvider typeProvider; // and Xtext @Check
  @Check def void checkTextWidgetForNonBoolean(TextWidget widget) {
    val jvmTypeReference = widget.attr.getJvmType
    if (jvmTypeReference == null) return;
    if (jvmTypeReference.getQualifiedName().equals(Boolean::TYPE.getName()))
      error("Textbox may NOT refer to boolean attributes.", ...
  } ...
```

Listing 7. Xtext validator in the Xbase scenario

5 Xsemantics

Xsemantics [7] (the successor of Xtypes [6]) is a DSL (written in Xtext) for writing type systems, reduction rules, interpreters (and in general relation rules) for languages implemented in Xtext. In this paper we illustrate its use for defining type systems. Xsemantics is intended for developers who are at least a little familiar with formal type systems and operational semantics since it uses a syntax that resembles deduction rules in a formal setting [12,20,31]. A type system definition in Xsemantics is a set of *judgments* (formally, assertions about the typing of programs) and a set of *rules* (formally, implications between judgments, i.e., they assert the validity of certain judgments, possibly on the basis of other judgments [12]) which have a conclusion and a set of premises; these rules can act on any Java object, though, typically, they will act on `EObjects` which are elements of the metamodel of a language implemented in Xtext. Xsemantics relies on Xbase to provide a rich syntax for defining rules, thus giving full access to Java types. Starting from the definitions of judgments and rules, Xsemantics generates Java code that can be used in a language implemented in Xtext for scoping and validation (it also generates a validator in Java).

An Xsemantics judgment consists of a name, a *judgment symbol* (which can be chosen from some predefined symbols) and the *parameters* of the judgment; these parameters are separated by a *relation symbol* that can be chosen from some predefined symbols. The parameters can be either input parameters (default) or output parameters (using the `output` keyword followed by the Java type). The judgment definitions for our case study are shown in Listing 8.

```
judgments {
  type |- EObject typable : output Type
  isAssignable |- Type left <~ Type right // whether right is assignable to left
  mostGeneral |- Type first ~~Type second |> output Type // most general between 1st and 2nd
}
```

Listing 8. Judgment definitions in Xsemantics

Once the judgments are declared, we can start declaring the rules of the judgments. Each rule consists of a name, a *rule conclusion* and the *premises* of the rule. The conclusion consists of the name of the *environment* of the rule, a *judgment symbol* and the *parameters* of the rules, which are separated by a *relation symbol* that can be chosen from some predefined symbols. To enable better IDE tooling and a more "programming"-like style, Xsemantics rules are written in the other direction with respect to standard deduction rules: the conclusion comes before the premises.

```
axiom BooleanLiteralType
  G |- BooleanLiteral lit :
      XsemGuiDslFactory::eINSTANCE.createBooleanType

rule AttributeRefType
  G |- AttributeRef attrRef : Type type
from { G |- attrRef.attr : type }

rule LengthOfType
  G |- LengthOf len : XsemGuiDslFactory::eINSTANCE.createIntType
from { G |- len.expr : var StringType stringType }

rule WidgetContentType
  G |- WidgetContent widgetContent : Type type
from { G |- env(G, 'widgetcontent', Attribute) : type }
```

$$\Gamma \vdash \mathtt{true} : \mathtt{boolean}$$

$$\frac{\Gamma \vdash \mathtt{attr} : \mathtt{T}}{\Gamma \vdash \mathtt{ref\ attr} : \mathtt{T}}$$

$$\frac{\Gamma \vdash \mathtt{exp} : \mathtt{string}}{\Gamma \vdash \mathtt{lengthOf(exp)} : \mathtt{int}}$$

$$\frac{\Gamma \vdash \Gamma(\mathtt{widgetcontent}) : \mathtt{T}}{\Gamma \vdash \mathtt{widgetcontent} : \mathtt{T}}$$

Listing 9. Some examples of rules and axioms in Xsemantics

The things that make a rule belong to a specific judgment are the judgment symbol and the relation symbols (which separate the parameters); moreover, the types of the parameters of a rule must be (Java) subtypes of the corresponding types of the judgment (or exactly the same Java types). Two rules belonging to the same judgment must differ for at least one input parameter's type.

The premises of a rule, which are specified in a `from` block, can be any Xbase expression, or a *rule invocation*. If one thinks of a rule declaration as a function declaration, then a rule invocation corresponds to a function invocation, thus one must specify the environment to pass to the rule, as well as the input and output arguments. The premises of an Xsemantics rule are considered to be in a *logical and* relation and are verified in the same order they are specified in the block. If one needs premises (or blocks of premises) in a *logical or* relation, the operator `or` can be used to separate blocks of premises. If a rule does not require any premise, we can use a special kind of rule, called *axiom*, which only has a conclusion. In the premises one can assign values to the output parameters; and when another rule is invoked, upon return, the output arguments will have the values assigned in the invoked rule.

The rule *environment* (taken from the type theory where it is usually denoted by Γ) allows to pass additional arguments to rules (e.g., contextual information, bindings for specific keywords, like `widgetcontent` as in Listing 12, etc.). An empty environment can be passed using the keyword `empty`. When passing an environment during a rule invocation, one can specify additional *environment mappings*, using the syntax `key <- value`. In general, environment mappings concern free variables occurring in the rule (like `widgetcontent`).

At runtime, upon rule invocation, the generated Java system will select the most appropriate rule according to the runtime types of the passed argument (using the *polymorphic dispatch* mechanism provided by Xtext). If one of the premises fails, then the whole rule will fail. In particular, if the premise is a `boolean` expression, it will fail if the expression evaluates to `false`. If the premise is a rule invocation, it will fail if the invoked rule fails.

In Listing 9 we present some rules for the judgment `type` (see Listing 8, recall that in the rules of these judgments the second parameter is an output parameter). In Listing 9 we also show on the right the typical type deduction rules in a possible formalization

```
rule MinusType                                    rule PlusType
  G |- Minus minus : NumberType type                G |- Plus plus : Type type
from {                                            from {
  // require number types                           // deal with any type
  G |- minus.left : var NumberType leftType         G |- plus.left : var Type leftType
  G |- minus.right : var NumberType rightType       G |- plus.right : var Type rightType
  // get the most general                           // get the most general (may be string)
  G |- leftType ~~rightType |> type                 G |- leftType ~~rightType |> type
}                                                 }
```

Listing 10. Some rules for binary expressions

```
rule IsAssignableBase G |- Type left <~ Type right from { left.eClass == right.eClass }

axiom BooleanAssignableToString G |- StringType left <~ BooleanType right
axiom IntAssignableToString G |- StringType left <~ NumberType right
axiom IntAssignableToFloat G |- FloatType left <~ IntType right

rule EntityTypeAssignable G |- EntityType left <~ EntityType right
from {
  left.ref == right.ref or
  getAll(right.ref, XsemGuiDslPackage::eINSTANCE.entity_SuperType,
     XsemGuiDslPackage::eINSTANCE.entity_SuperType, typeof(Entity)).contains(left.ref) }
```

Listing 11. Some rules for the `isAssignable` judgment

of our case study, to show how the specifications are similar (apart from the order of the conclusion and the premises). For typing a literal (in the example a boolean literal) we write an axiom (since there is no premise) and the result is a `BooleanType` (created through the EMF factory for our language). The rule for typing an `AttributeRef` can be read as follows: the type of an `AttributeRef` is the type resulting from typing the corresponding referred attribute (the feature `attr`, refer to Listing 2). The type of a `LengthOf` expression is an integer type, provided that the expression argument of `LengthOf` has string type. Finally, for typing `widgetcontent` we make use of the rule environment: we access the environment with the predefined function `env`, by specifying the key and the expected Java type of the corresponding value. If no key is found in the environment or the value cannot be assigned to the specified Java type the premise will fail. We will show how an environment is passed later in Listing 12. Thus, this rule will type `widgetcontent` with the type of the corresponding attribute.

The rules for `Minus` and `Plus` are shown in Listing 10 (the rules for `mostGeneral` are not shown). The typing rule for `Minus` requires that the two subexpressions have a numeric type (recall that since we specify a `NumberType` as the output argument in rule invocation, its invocation will succeed only if the result is assignable to `NumberType`); the resulting type will be the most general type, thus, for instance, if one of the two subexpressions has the type `FloatType` and the other one `IntType`, the resulting type will be `FloatType`. Recall that we use + not only as the arithmetic operator, but also for string concatenation; in particular, if one of the subexpression is a string, the whole expression is considered a string concatenation. Thus, the rule for `Plus` computes the types of the two subexpressions, and then gets the most general; if one of them is a string type, the whole expression will have string type (see also subtyping rules in Listing 11).

The rules of the judgment `isAssignable` (Listing 8), which basically implements subtyping, do not have an output parameter, they accept two types as parameters: they succeed if the right parameter is assignable to the left parameter. In Listing 11 we show some rules for this judgment. The first rule is the most general and states that types of the same kind are assignable to each other (in this context "kind" corresponds to `EClass`); moreover, we have axioms saying that booleans and integers are assignable to strings (for instance, like in Java, by an implicit conversion through `toString` method). Finally, an integer can be assigned to a float. For the subtyping between two `EntityTypes` the idea is that `right` can be assigned to `left` either if they are the same type (entity subtyping is reflexive) or if `left` is a "super entity" (possibly indirectly) for `right`. To avoid dealing with possible malformed cyclic hierarchies, we use a predefined function of Xsemantics, to compute the "closure" of a graph in a cycle-proof way:

```
getAll(eObject, feature to collect, feature to follow, expected type)
```

In Listing 11, it will return all the superclasses of `right`.

In an Xsemantics system we can specify some special rules, `checkrule`, which do not belong to any judgment. They are used by Xsemantics to generate a Java validator for the Xtext language. A `checkrule` has a name, a single parameter (which is the `EObject` which will be checked by the validator) and the premises (but no rule environment). The syntax of the premises of a `checkrule` is the same as in the standard rules. Xsemantics will generate a Java validator with a `@Check` method for each `checkrule`; just like in Java validators for Xtext languages, one can have many checkrules for the same `JavaType` (provided the rule name is unique).

```
checkrule AttributeTypeChecks for Attribute attribute from {
    empty |- attribute : var Type type
}
checkrule CheckMustBeBoolean for Widget widget from {
    widget.check == null or
    'widgetcontent' <- widget.attr |- widget.check : var BooleanType boolType
}
checkrule CheckTextWidgetAttributeNotBoolean for TextWidget widget from {
    'widgetcontent' <- widget.attr |- widget.attr : var Type attrType
    !(attrType instanceof BooleanType)
}
checkrule CheckCheckBoxWidgetAttributeBoolean for CheckBoxWidget widget from {
    'widgetcontent' <- widget.attr |- widget.attr : var BooleanType attrType
}
```

Listing 12. Some checkrules for the Validator

In Listing 12 we present some checkrules for validating the elements of our language (see Section 2). The first checkrule basically states that an `Attribute` is correct if we can give it a type (in the empty environment). The second one accepts a `Widget` and ensures that either its check part is not specified or it has a boolean type; note that in this case we pass to the type rule invocation an explicit environment so that we are able to type possible occurrences of `widgetcontent`. These first two rules also show an important use of the environment: since the rule for typing `widgetcontent` (Listing 9) requires that the string 'widgetcontent' is bound to an attribute in the environment, and since when typing an attribute we provide an empty environment, then a possible occurrence of `widgetcontent` in an attribute's initialization expression (which is accepted by the grammar) will be automatically (and correctly) rejected. The third checkrule requires that the TextWidget's attribute is not of boolean type, while the fourth one requires that the checkbox's attribute has a boolean type (by implicitly trying to assign the result type, in the rule invocation, to a boolean type).

6 XTS

XTS [38] was originally developed as a framework with a Java API to declaratively specify type system rules. Later, a DSL was added on top of the framework. The DSL provides several usability advantages such as code completion into the target metamodel and static consistency checks for the type system definition. From the type system expressed with the DSL, Java code is generated that uses the original Java API. The DSL only covers the most important cases of type system specification. For the remainder, Java code has to be written against the framework API. XTS has been used in several commercial projects, by the author and by independent third parties.

Each XTS type system specification starts with a header. It specifies the fully qualified class name of the type system implementation class generated from this specification file. It also specifies the platform URI of the Ecore file for the target language as well as the fully qualified name of the EMF-generated package class for that Ecore file:

```
typesystem org.typesys.xts.guidsl.typesys.GuiDlsTypesystem
  ecore file "platform:/resource/or...esys/xts/guidsl/GuiDsl.ecore"
  language package org.typesys.xts.guidsl.guiDsl.GuiDslPackage
```

Type system specifications mostly consists of typeof clauses, which define how the type for a given EClass is calculated. It can optionally specify constraints on the types of properties of these metaclasses. In the code below we specify that the type of instances of Type (and all its subclasses, indicated by a +) is a clone of itself. In other words, types are their own types. We also specify the subtyping relationship between FloatType and IntType: wherever a FloatType is expected, an IntType can be used as well (but not the other way around). IntType is more specialized.

```
typeof Type+ -> clone
subtype IntType base FloatType
```

The type of string and boolean literals is fixed (it does not depend on any further context), so it can be assigned statically. However, for number literals it depends on its value whether it is an integer or float type. This cannot be be expressed declaratively in the DSL. So we declare the type for NumberLiteral to be calculated with Java code.

```
typeof StringLiteral -> StringType
typeof BooleanLiteral -> BooleanType
typeof NumberLiteral -> javacode
```

javacode leads to the generation of an abstract method in the generated type system class, which we have to override in a manually written subclass:

```
public EObject type( NumberLiteral s, TypeCalcTrace trace ) {
    if ( s.getValue().equals(s.getValue().intValue())) return create(cl.getIntType());
    return create(cl.getFloatType()); }
```

Before we define the type system rules for expressions, we define two characteristics. A *characteristic* is essentially a named set of types. Instead of listing the set of types over and over again, we can use the characteristic as a shortcut:

```
characteristic COMPARABLE { IntType, FloatType, BooleanType, StringType }
characteristic NUMERIC { IntType, FloatType }
```

Then we define the type for expressions. `Expression` itself has no type, since it is abstract, and all its subclasses specify their own typing rules. It is useful to make this explicit in the specification since the type system DSL editor will then check that all subconcepts of `Expression` are actually covered by their own type system rules:

```
typeof Expression -> abstract
```

We can now take a look at some of the more interesting cases (Listing 13). For `Comparison` (one of the binary expressions not shown in the grammar in Listing 2), the type will be `BooleanType` and the `left` and `right` arguments have to be `COMPARABLE` (see the characteristic above). In addition, they also have to be *compatible*. For instance, while boolean and string types are both `COMPARABLE`, they cannot be compared *to each other*. This is why we need this explicit compatibility check using the `:<=:` operator. It represents *ordered* compatibility, meaning that the type on the left must be the same or a subtype of the type specified on the right. In contrast, `:<=>:` represents *unordered* compatibility where the left must be the same or a subtype of the right, *or vice versa*. An example is in the typing rule for `Plus`. The `Plus` also shows the use of `common` as a type specification. This expresses that the type is the common supertype of the two arguments. This only works if the two types are either the same or part of a subtype relationship (such as `FloatType` and `IntType`).

```
typeof Comparison -> BooleanType {
  ensureType left :<=: char(COMPARABLE)
  ensureType right :<=: char(COMPARABLE)
  ensureCompatibility left :<=>: right
 }
typeof Plus -> common left right {
  ensureType left :<=: StringType, char(NUMERIC)
  ensureType right :<=: StringType, char(NUMERIC)
  ensureCompatibility left :<=>: right
}
```

```
typeof Equality -> BooleanType {
  ensureType left :<=:
     char(COMPARABLE), BooleanType
  ensureType right :<=:
     char(COMPARABLE), BooleanType
  ensureCompatibility left :<=>: right
}
```

Listing 13. Some rules for binary expressions

```
typeof Widget -> abstract

typeof CheckBoxWidget -> feature attr {
   // 3) checkboxes must refer to boolean attrs
   ensureType attr :<=: BooleanType
   // 1) the check expression must be boolean
   ensureType check :<=: BooleanType
}
```

```
typeof TextWidget -> feature attr {
   ensureType length :<=: IntType
   // 2) text widgets may only
   // refer to non-boolean attributes
   ensureType attr :<=:
        StringType, IntType, FloatType
   ensureType check :<=: BooleanType
}
```

Listing 14. Types for widgets

Let us now look at some more special cases: the type of the `AttributeRef` is the type of the referenced `Attribute`, reachable via the `attr` reference:

```
typeof AttributeRef -> feature attr
```

The type of the widgets (they are not expressions, but as we will see below, it is useful for them to have a type) are shown in Listing 14. They implement the checks for our case study (see the comments in Listing 14).

We can now calculate the type of the `widgetcontent` expression, which has the type of the `Attribute` referenced by the parent widget. This is expressed by (relying on the fact that we have provided a type for the `Widget`, as mentioned above):

```
typeof WidgetContent -> ancestor Widget
```

Finally, the type of an `Entity` also has to be calculated with Java code, because it has to be an `EntityType` that references the corresponding entity. While a declarative way for specifying such a rule is easily imaginable (e.g. `typeof Entity -> EntityType(ref=this)`), the DSL currently does not support it:

```
protected EObject type(Entity element, TypeCalcTrace trace) {
   EntityType et = (EntityType)create(cl.getEntityType());
   et.setRef(element);
   return et; }
```

There is one more interesting open issue. We have to implement the subtyping relationship between entities. This is not so simple, because we compare `EntityTypes`, and the subtyping depends on whether their corresponding referenced entities are subtypes of each other (covariance). So instead of declaring a subtype relationship directly in the XTS specification, we implement a type comparison method in Java. The method `internalCompareTypesOrdered()` checks the entity subtype relation with cycle detection as seen in the plain Xtext case (Listing 4, method `internalIsAssignable()`).

```
protected boolean compareTypes( EntityType t1, EntityType t2, CheckKind k, TypeCalcTrace t
     ){
   if ( k == CheckKind.same ) return t1.getRef() == t2.getRef();
   if ( k == CheckKind.ordered ) return internalCompareTypesOrdered(
      t1.getRef(), t2.getRef(), Sets.<Entity>newHashSet());
   return false; }
```

Once again, the DSL could easily be extended to support this feature, for example using `subtype Entity covariant ref`.

7 Evaluation

7.1 When to Use Which Approach

The choice of which type system framework to use to implement the type system for a DSL in Xtext mainly depends on the context of the DSL itself. In this section we summarize our experiences, and evaluate the features of the mechanisms and frameworks described in this paper.

Plain Xtext. The plain Xtext strategy (Section 3) is always feasible, and, by relying on the powerful features of Xtend, it is even easier to deal with complex model visits. However, if the DSL has to rely on an involved type system, implementing such functionalities in plain Java/Xtend might still be a big effort. The complete control on all the parts of the implemented type system comes at the cost of having to deal with all the internal details, without relying on any abstraction. For instance, as shown in Listing 3, loop protection for type computations has to be implemented manually.

Xbase. If the DSL has to be tightly integrated with Java, there is basically one single sensible choice: rely on Xbase. By "integration with Java" we mean that the DSL must reuse Java types not only in declarations but also in the actual operation code (for instance, it must create objects in a Java-like style and invoke methods on such instances).

From Xbase the DSL "inherits" a rich Java-like syntax for expressions (including closures and type inference) and the complete Java type system. In particular, the expression parts of the DSL will be dealt with directly by Xbase, relieving the programmer from the big burden of having to reimplement repetitive type checks. It is also possible to customize several aspects of Xbase expressions, including the syntactic shape (though it might not be possible to change radically such syntax without experiencing grammar ambiguities) and the typing and semantics. The latter scenario, though, might require some deeper knowledge of Xbase internals (for which, in most cases, the code of Xbase is the only documentation). For instance, in Xsemantics itself, Xbase is used for the syntax of the premises of rules; however, single boolean Xbase expression statements in the premises of an Xsemantics rule have a different semantics: if the expression does not evaluate to true the whole rule must fail. In Xbase, a boolean expression used as a statement is not considered valid (it represents a statement with no side effect). To deal with that, in Xsemantics, a custom validator is implemented to "intercept" the checks in the Xbase validator in order not to issue an error in these situations. Similarly, a custom Xbase compiler is implemented in Xsemantics in order to wrap the generated Java code for boolean expressions with Java code that deals with possible failures of such expressions. Thus, the more the DSL is similar to Java, the easier it will be to reuse Xbase. Otherwise, things might get more complicated, though it is still possible to customize the typing and semantics of Xbase expressions.

Xsemantics. Xsemantics can be a useful framework for implementing a language which has been formalized using standard meta-theory mechanisms: define the type system of the language, its semantics, and then prove that the language is sound by showing that the semantics is consistent with the type system. Xsemantics aims at providing a rich syntax for defining any kind of rules: relations among elements (e.g., *subtyping*), *static semantics* (i.e., type systems) and *dynamic semantics* (i.e., reduction rules that can be used for interpreting a program). Xsemantics syntax aims at resembling the way deduction rules are written in a formal setting (see Listing 9). Thus, it is easy to implement the formal definition of type system and operational semantics in Xsemantics, since the gap between formal systems syntax and the actual implementation is reduced. Moreover, when implementing type systems in Xsemantics, the details of the original formal type system are not lost and spread through several lines of Java (or Xtend) code, and the programmer is relieved from many implementation details, thanks to the declarative style of Xsemantics rule syntax. For instance, Xsemantics was used to implement the type system of *Featherweight Java* [22] (a minimal Java core, used to prove properties of Java-like languages) which was previously implemented manually in Xtext [5], and to implement type inference with unification for computing the most general type for a simple *lambda calculus* (see the examples in [7]). Furthermore, it is being employed to re-engineer the implementation of other languages implemented in Xtext, which have a solid theoretical foundation ([33,8]).

XTS. XTS offers a concise syntax for the most common tasks when defining type systems. These include specifying that the type of an element is a copy of itself, subtype relationships and compatibility constraints, grouping several elements (*characteristic*) to save constraint code, and access to EMF features of model elements which allows to specify that a model element has the type of one of its features. XTS has been used in several real-world Xtext DSLs and has proven to be useful and stable. While a few additional features would be worth adding to the DSL (e.g. instantiating structured types such as the `EntityType`), the fact that only a few types have to be calculated in Java is not a big problem in practice.

7.2 Evaluation Regarding the Features

In this section we evaluate the four approaches regarding their ability to support the features introduced in Section 1.

F1, Conciseness. Since XTS only targets type systems, the specification of a type system in XTS is more compact than Xsemantics (where judgments for typing and subtyping have to defined explicitly). Xbase can be compact if only Xbase-provided typing is needed. The plain Xtext-approach is the most verbose because Xtext itself does not directly support typing rules and because users have to express typing rules with a GPL. However, this GPL, Xtend, can be quite concise, alleviating this problem to some extent.

F2, Support for Common Cases. XTS is certainly optimized towards the most common cases, they can be expressed very concisely. As a trade-off, rarer cases have to be expressed in Java, relying on the Generation Gap pattern [36]. In Xsemantics the need of providing custom Java code is rare since it relies on Xbase for rule implementation, thus it already has a richer syntax; on the other hand, the common case is not quite as concise as in XTS. Type system specification with Xbase relies on the fact that the common cases are already implemented in Xbase itself, so users get them almost ”for free”. However, implementing corner cases may require deep knowledge of the inner workings of Xbase. The plain Xtext case does not provide any direct support for typing; so support for the common cases has to be built manually.

F3, Consistency Checking. All approaches refer to program elements using the EClasses of the Ecore model and detect when there are typing rules that refer to no longer existing elements of the AST. XTS however also provides additional static checks that ensure, for example, that each subconcept of an abstract concept has a type specification.

F4, Integration with Validation. All four alternatives can be integrated with Xtext validation; in Xbase and Xsemantics this is fully automatic. In XTS, the user has to write a single 3 line stereotypical validation method manually. The plain Xtext case relies on the developer to provide such integration.

F5, Test and Debugging. An important feature that we believe type system frameworks should provide is the ability to keep a trace of the computation that brings to the derivation of a type (or of its failure). This is crucial both for debugging and for testing the type system. Both Xsemantics and XTS provide mechanisms for accessing such traces.

In XTS, a popup window is available for the Xtext editor that shows the type of program elements and the trace in the editor itself. Xbase does not come with a trace facility for type computations, neither is the plain Xtext case. For debugging, all alternatives rely on debugging (generated) Java code and testing can be done with the regular means of testing constraints in Xtext. Xbase also allows to debug Xbase expressions, thus it is also possible to directly debug Xtend and Xsemantics code.

Summarizing, both Xsemantics and XTS provide all the functionalities that we outlined in the Introduction, thus they complement Xtext for the implementation of DSLs which require an involved type system. In particular, the Java code generated by these frameworks seamlessly integrates in the validation mechanisms of Xtext. The table below shows a summary.

	Plain Xtext	Xbase	Xsemantics	XTS
F1, Conciseness	-	0	+	++
F2, Common Cases	-	0/+	+	+
F3, Consistency Checking	0	0	+	+
F4, Integration with Validation	0	++	++	+
F5, Test and Debugging	0	0	+	+

8 Related Work

This section discusses related work concerning Xtext (we also refer to [30] for a wider comparison) and other language workbenches regarding type system implementations.

Tools like IMP (The IDE Meta-Tooling Platform) [13] and DLTK (Dynamic Languages Toolkit) [1] only deal with IDE functionalities and do not address type system definition specifically. The same is true for TCS (Textual Concrete Syntax) [23] and EMFText [19]. However, the latter two rely on a metamodel (abstract syntax) expressed using EMF Ecore, so the XTS and Xsemantics type system frameworks discussed in this paper could be used with EMFText and TCS. This does not hold for MPS [37] and Spoofax [24] since they are not built on top of EMF. However, they both come with their own support for type system specification.

In MPS, language developers specify declarative type system equations with a special DSL. A solver then tries to solve all the type system equations relevant to a given program in order to compute and check types for program elements. The following is the type equation for a local variable declaration of the form `int x = 3;`. It expresses that the type of the `init` expression (the `3` in the example) must be the same or a subtype of the type of the `dtype` property (`int` in the example).

```
typeof(lvd.init) :<=: typeof(lvd.dtype);
```

The following code shows the type calculation of an array initialization expression (as in `{1, 2, 3.5}`). The second line computes the type `T` as the common supertype of all the types of the elements in the initialization expression (`float` in the example). The third line then makes the type of the complete `arrInitExpr` an array of `T`.

```
type var T
foreach ( e: init.elements ) T :<=: typeof(e)
typeof(arrInitExpr) :==: new ArrayType(T);
```

Spoofax relies on the Stratego [11] term transformation language for defining most language aspects beyond the grammar. For typing, Spoofax predefines a transformation rule `type-of` which returns the type of a program element passed in as an argument. It has to be overridden for any language concept that needs to be typed. Below are the rules that type the `Int` by assigning `IntType`:

```
type-of: Int(value) -> IntType()
```

The type of `Add` depends on the type of its arguments, so `where` clauses must be used to distinguish the cases:

```
type-of: Add(exp1, exp2) -> IntType()
  where <type-of> exp1 => IntType() ; <type-of> exp2 => IntType()
type-of: Add(exp1, exp2) -> StringType()
  where <type-of> exp1 => StringType() ; <type-of> exp2 => StringType()
```

Both in MPS and in Spoofax, the typing rules are declarative. If language extensions are added, the type system for the additional language concepts can be defined by just *adding* additional cases.

Many systems (for example, SILVER [40], JastAdd [16] and LISA [26]) describe type systems using attribute grammars. Attribute grammars associate attributes with AST elements. These attributes can capture arbitrary data about the element (such as its type). Xsemantics and XTS can be seen as special-purpose attribute grammars in the sense that they associate a type attribute with program elements. As a consequence of the fact that Xsemantics and XTS are purpose-built for typing, they provide a more concise notation, more specific error reporting regarding the consistency of the type system and a custom Xtext-based editor. Of course, in contrast to general-purpose attribute grammar systems, Xsemantics and XTS cannot be used for anything else except type system specification.

OCL [39,27] is an expression language for declaring constraints on (meta)models. It is not particularly well suited for typing, however, since it lacks the necessary typing-specific high-level abstractions available in Xsemantics and XTS.

EriLex [41] is a software tool for embedded DSLs and it supports specifying syntax, type rules, and dynamic semantics of such languages but it does not generate any artifact for IDE functionalities. On the contrary, Xtext shares with older tools (like the Synthesizer Generator [32] and Centaur [9]) the philosophy that a tailored editor is crucial for the usability of a language (in particular the immediate feedback about problems in the program being edited) and thus a framework for implementing languages should address also this functionality.

In contrast to many other approaches (such as Centaur [9], MPS [37], ASF+SDF [10], Ruler [15], PLT Redex [17], and EriLex [41]), Xsemantics and XTS specifications do not refer to the grammar elements of the language but only to the structure of the EMF model representing the AST. Thus, they might be used with any other framework that uses EMF metamodels (though this still requires some investigation).

In contrast to for example Ott [34], neither Xsemantics nor XTS aim at providing mechanisms for formal proofs for the language and the type system. They do no generate versions of the type system for proof assistants like Coq [4], HOL [18] or Isabelle [29]. Maude [14], a language based on rewriting logic, can be used as a semantic framework, with formal analysis capabilities, to specify and prototype other languages. Whether Xsemantics and XTS could be used for formal proofs is still under investigation.

As the focus on Xtext suggests, our paper addresses only *external* DSLs, where DSL programs are separate from programs written in programming languages. With *internal* DSLs, the DSL programs are embedded in host programs written in a suitable programming language. Also, the DSL is defined and implemented with the means of that host language. In case the host language supports static typing, the host language's type system is used for typing the DSL. This is illustrated, for example by [21] which describes embedding of DSLs into Scala. Internal DSLs are however most often used with languages that use dynamic typing (such as Groovy or Ruby), so static typing, as discussed in this paper, does not apply.

9 Conclusion and Future Work

In this paper we presented approaches for developing type systems for DSLs implemented in Xtext. While a type system can always be implemented in plain Java or Xtend, still it is crucial for productivity to have a dedicated framework with specific functionalities for implementing type systems. Xtext provides the expression language Xbase that can be reused in a DSL. This is useful in cases where the DSL is tightly coupled with Java, as Xbase provides a full integration with the Java type system.

Xsemantics and XTS both provide a framework and a DSL to make the implementation of type systems more concise and more maintainable, regardless of whether the DSL is tied to Java or not. Conciseness is achieved via the *declarative* specification of type system rules, focusing on type computation and subtyping while hiding implementation details. Improved *maintainability* is a consequence of conciseness and the static checks of type system completeness in the type system specification editor.

Future work will focus on the support for expressing extensible type systems. Xtext allows a language to extend another language, making the concepts defined in that language available to the new language. Both XTS and Xsemantics currently do not support the definition of typing rules in a way that is similarly extensible.

References

1. DLTK, http://www.eclipse.org/dltk
2. Xtend, http://xtend-lang.org
3. Xtext, http://www.eclipse.org/Xtext
4. Bertot, Y., Castéran, P.P.: Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions. Texts in theoretical computer science. Springer (2004)
5. Bettini, L.: An Eclipse-based IDE for Featherweight Java implemented in Xtext. In: ECLIPSE-IT, pp. 14–28 (2010)
6. Bettini, L.: A DSL for Writing Type Systems for Xtext Languages. In: PPPJ, pp. 31–40. ACM (2011)

7. Bettini, L.: Xsemantics (2012), http://xsemantics.sourceforge.net/
8. Bettini, L., Damiani, F., Schaefer, I., Strocco, F.: TraitRecordJ: A programming language with traits and records. Science of Computer Programming (to appear, 2012)
9. Borras, P., Clement, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B., Pascual, V.: CENTAUR: the system. SIGPLAN 24(2), 14–24 (1988)
10. Van Den Brand, M.G.J., Heering, J., Klint, P., Olivier, P.A.: Compiling language definitions: the ASF+SDF compiler. ACM TOPLAS 24(4), 334–368 (2002)
11. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A language and toolset for program transformation. Science of Computer Programming 72(1-2), 52–70 (2008)
12. Cardelli, L.: Type Systems. ACM Computing Surveys 28(1), 263–264 (1996)
13. Charles, P., Fuhrer, R., Sutton Jr., S., Duesterwald, E., Vinju, J.: Accelerating the creation of customized, language-Specific IDEs in Eclipse. In: OOPSLA. ACM (2009)
14. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: The Maude 2.0 System. In: Nieuwenhuis, R. (ed.) RTA 2003. LNCS, vol. 2706, pp. 76–87. Springer, Heidelberg (2003)
15. Dijkstra, A., Swierstra, S.D.: Ruler: Programming Type Rules. In: Hagiya, M. (ed.) FLOPS 2006. LNCS, vol. 3945, pp. 30–46. Springer, Heidelberg (2006)
16. Ekman, T., Hedin, G.: The JastAdd system – modular extensible compiler construction. Science of Computer Programming 69(1-3), 14–26 (2007)
17. Felleisen, M., Findler, R.B., Flatt, M.: Semantics Engineering with PLT Redex. The MIT Press, Cambridge (2009)
18. Gordon, M.: From LCF to HOL: a short history. In: Proof, Language, and Interaction: Essays in Honour of Robin Milner, pp. 169–186. The MIT Press (2000)
19. 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)
20. Hindley, J.R.: Basic Simple Type Theory. Cambridge University Press (1987)
21. Hofer, C., Ostermann, K., Rendel, T., Moors, A.: Polymorphic embedding of DSLs. In: Smaragdakis, Y., Siek, J.G. (eds.) Proceedings of the 7th International Conference on Generative Programming and Component Engineering, GPCE 2008, Nashville, TN, USA, October 19-23, pp. 137–148. ACM (2008)
22. Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. ACM TOPLAS 23(3), 396–450 (2001)
23. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: GPCE, pp. 249–254. ACM (2006)
24. Kats, L.C.L., Visser, E.: The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In: OOPSLA, pp. 444–463. ACM (2010)
25. Levine, J.: flex & bison. O'Reilly Media (2009)
26. Mernik, M., Lenič, M., Avdicauševic, E., Zumer, V.: LISA: An Interactive Environment for Programming Language Development. In: Nigel Horspool, R. (ed.) CC 2002. LNCS, vol. 2304, pp. 1–4. Springer, Heidelberg (2002)
27. Object Management Group. Object Constraint Language, Version 2.2 (2010), http://www.omg.org/spec/OCL/2.2
28. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Programmers (2007)
29. Paulson, L.C.: Isabelle: A Generic Theorem Prover. LNCS, vol. 828. Springer, Heidelberg (1994)
30. Pfeiffer, M., Pichler, J.: A comparison of tool support for textual domain-specific languages. In: DSM, pp. 1–7 (2008)

31. Pierce, B.C.: Types and Programming Languages. The MIT Press (2002)
32. Reps, T., Teitelbaum, T.: The Synthesizer Generator. In: Software Engineering Symposium on Practical Software Development Environments, pp. 42–48. ACM (1984)
33. Schaefer, I., Bettini, L., Damiani, F.: Compositional Type-Checking for Delta-oriented Programming. In: AOSD, pp. 43–56. ACM (2011)
34. Sewell, P., Nardelli, F.Z., Owens, S., Peskine, G., Ridge, T., Sarkar, S., Strnisa, R.: Ott: Effective tool support for the working semanticist. J. Funct. Program 20(1) (2010)
35. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley (2008)
36. Vlissides, J.: Generation Gap Pattern. C++ Report 8(10):12, 14–18 (1996)
37. Voelter, M.: Language and IDE Development, Modularization and Composition with MPS. In: 4th Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2011). LNCS, Springer (2011)
38. Völter, M.: Xtext/TS - A Typesystem Framework for Xtext (May 2011), http://code.google.com/a/eclipselabs.org/p/xtext-typesystem/
39. Warmer, J., Kleppe, A.: The Object Constraint Language: Precise Modeling with UML. Addison Wesley (1999)
40. Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: an Extensible Attribute Grammar System. Electronic Notes in Theoretical Computer Science 203(2), 103–116 (2008)
41. Xu, H.: EriLex: An Embedded Domain Specific Language Generator. In: Vitek, J. (ed.) TOOLS 2010. LNCS, vol. 6141, pp. 192–212. Springer, Heidelberg (2010)

Author Index