Anthony Sloane
Uwe Aßmann (Eds.)

LNCS 6940

Software Language Engineering

4th International Conference, SLE 2011
Braga, Portugal, July 2011
Revised Selected Papers

Springer

Anthony Sloane   Uwe Aßmann (Eds.)

Software Language Engineering

4th International Conference, SLE 2011
Braga, Portugal, July 3-4, 2011
Revised Selected Papers

Springer

Volume Editors

Anthony Sloane
Macquarie University, Department of Computing
Sydney, NSW 2109, Australia
E-mail: anthony.sloane@mq.edu.au

Uwe Aßmann
Technische Universität Dresden, Fakultät Informatik
Institut für Software- und Multimediatechnik (SMT)
Nöthnitzer Straße 46, 01069 Dresden, Germany
E-mail: uwe.assmann@tu-dresden.de

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-28829-6 e-ISBN 978-3-642-28830-2
DOI 10.1007/978-3-642-28830-2
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012932856

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

LNCS Sublibrary: SL 2 – Programming and Software Engineering

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

Printed on acid-free paper

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

Preface

We are pleased to present the proceedings of the 4th International Conference of Software Language Engineering (SLE 2011). The conference was being held in Braga, Portugal, during July 3–5, 2011, with several sessions on research papers, tool/language demonstrations, a doctoral symposium, and an industry track. SLE 2011 was co-located with the 4th Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2011) and two workshops: the Industry Track of Software Language Engineering (ITSLE 2011) and the Coupled Software Transformations Workshop (CSXW 2011).

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 response to the call for papers for SLE 2011 was lower than in 2010. We received 45 full submissions from 50 abstract submissions. From these submissions, the Program Committee (PC) selected 20 papers: 16 full papers and 4 tool/language demonstration papers, resulting in an acceptance rate of 44%.

SLE 2011 would not have been possible without the significant contributions of many individuals and organizations. We are grateful to the organizers of GTTSE 2011 for their close collaboration and management of many of the logistics, in particular our host João Saraiva, who took great care that everything worked out extremely well. We also wish to thank our financial supporters: Centro de Ciências e Tecnologias de Computação (CCTC), Fundação para a Ciência e a Tecnologia (FCT), Luso-American Foundation (FLAD), Multicert, and Software Improvement Group (SIG).

The SLE 2011 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 also thank the authors for their efforts in writing and then revising their papers, and we thank the Springer team for the final proceedings.

January 2012

Anthony Sloane
Uwe Aßmann

Organization

SLE 2011 was hosted by the Departamento de Informática, Universidade do Minho, Braga, Portugal.

General Chair

João Saraiva	Universidade do Minho, Braga, Portugal

Program Co-chairs

Anthony Sloane	Macquarie University, Australia
Uwe Aßmann	Dresden University of Technology, Germany

Organizing Committee

José Creissac Campos	University of Minho (Finance Chair), Portugal
Joost Visser	SIG (GTTSE/SLE Students' Workshop Co-chair), The Netherlands
Eric Van Wyk	University of Minnesota (GTTSE/SLE Students' Workshop Co-chair), USA
Jurgen Vinju	CWI (Workshop Selection Chair), The Netherlands
João Paulo Fernandes	University of Porto and University of Minho (Web and Publicity Co-chair), Portugal
Vadim Zaytsev	CWI (Publicity Co-chair), The Netherlands

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 Gasevic	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

Adrian Johnstone	University of London, UK
Aldo Gangemi	Semantic Technology Laboratory, Italy
Alexander Serebrenik	Eindhoven University of Technology, The Netherlands
Ana Moreira	FCT/UNL, Portugal
Anthony Cleve	University of Namur, Belgium
Anthony Sloane	Macquarie University, Australia
Anya Helene Bagge	University of Bergen, Norway
Bernhard Rumpe	Aachen University, Germany
Bijan Parsia	University of Manchester, UK
Brian Malloy	Clemson University, USA
Bruno Oliveira	Seoul National University, South Korea
Chiara Ghidini	FBK-irst, Italy
Daniel Oberle	SAP Research, Germany
Eelco Visser	Delft University of Technology, The Netherlands
Eric Van Wyk	University of Minnesota, USA
Fernando Pereira	Federal University of Minas Gerais, Brazil
Fernando Silva Parreiras	University of Koblenz-Landau, Germany
Friedrich Steimann	University of Hannover, Germany
Görel Hedin	Lund University, Sweden
Ivan Kurtev	University of Twente, The Netherlands
James Power	National University of Ireland, Ireland
Jeff Gray	University of Alabama, USA
Jeff Z. Pan	University of Aberdeen, UK
João Paulo Fernandes	University of Minho and University of Porto, Portugal
John Boyland	University of Wisconsin-Milwaukee, USA
John Grundy	Swinburne University of Technology, Australia
Jordi Cabot	École des Mines de Nantes, France
Jurgen Vinju	CWI, The Netherlands
Laurence Tratt	Middlesex University, UK
Marjan Mernik	University of Maribor, Slovenia
Mark van den Brand	Eindhoven University of Technology, The Netherlands
Michael Collard	University of Akron, USA
Nicholas Kraft	University of Alabama, USA
Paul Klint	CWI, The Netherlands
Paulo Borba	Federal University of Pernambuco, Brazil
Peter Haase	University of Karlsruhe, Germany
Peter Mosses	Swansea University, UK

Ralf Möller	Hamburg University of Technology, Germany
Silvana Castano	University of Milan, Italy
Steffen Zschaler	Lancaster University, UK
Uwe Aßmann	Dresden University of Technology, Germany
Zhenjiang Hu	National Institute of Informatics, Japan

Additional Reviewers

Mauricio Alferez
Christoff Bürger
Robert Clarisó
Jácome Cunha
Chiara Di Francescomarino
Luc Engelen
Arne Haber
Florian Heidenreich
Christoph Herrmann
Roland Hildebrandt
Nophadol Jekjantuk
Ted Kaminski
Sven Karol
Markus Look
Karsten Martiny
Stefano Montanelli
Rainer Marrone
Ruth Raventos
Sebastian Richly
Mirko Seifert
Daniel Spiewak
Massimo Tisi
Arjan Van Der Meer
Tijs Van Der Storm
Jeroen Van Den Bos
Naveneetha Vasudevan
Sebastian Wandelt
Michael Welch

Sponsoring Institutions

CCTC	Centro de Ciências e Tecnologias de Computação
FCT	Fundação para a Ciência e a Tecnologia
FLAD	Luso-American Foundation
Multicert	
SIG	Software Improvement Group

Table of Contents

Towards a One-Stop-Shop for Analysis, Transformation and Visualization of Software

Paul Klint[1,2], Bert Lisser[1], and Atze van der Ploeg[1]

[1] Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
[2] INRIA Lille Nord Europe, France

Abstract. Over the last two years we have been developing the meta-programming language RASCAL that aims at providing a concise and effective language for performing meta-programming tasks such as the analysis and transformation of existing source code and models, and the implementation of domain-specific languages.

However, meta-programming tasks also require seamlessly integrated visualization facilities. We are therefore now aiming at a ”One-Stop-Shop” for analysis, transformation and visualization. In this paper we give a status report on this ongoing effort and sketch the requirements for an interactive visualization framework and describe the solutions we came up with. In brief, we propose a coordinate-free, compositional, visualization framework, with fully automatic placement, scaling and alignment. It also provides user interaction. The current framework can handle various kinds of charts, trees, and graphs and can be easily extended to more advanced layouts. This work can be seen as a study in domain engineering that will eventually enable us to create a domain-specific language for software visualization. We conclude with examples that emphasize the integration of analysis and visualization.

1 Introduction

Over the last two years we have been developing the meta-programming language RASCAL[1] [6] that aims at providing a concise and effective language for performing meta-programming tasks such as the analysis and transformation of existing source code and models, and the implementation of domain-specific languages. RASCAL is completely written in Java, integrates with Eclipse and gives access to programmable IDE features using IMP, The IDE Meta-Tooling Platform[2].

Given the large amounts of facts that emerge when analyzing software, meta-programming tasks also require seamlessly integrated visualization facilities We are therefore now aiming at a ”One-Stop-Shop” for analysis, transformation and visualization. In this paper we give a status report on this ongoing effort and sketch the requirements for an interactive visualization framework and describe the solutions we came up with. In brief, we propose a coordinate-free, compositional, visualization framework, with fully automatic placement, scaling and alignment. It also provides user interaction.

[1] `http://www.rascalmpl.org/`
[2] `http://www.eclipse.org/imp/`

Software visualization is a relatively young and broad domain [3] and there are several ongoing efforts to create software vizualization frameworks, for instance, Rigi [14], Bauhaus,[3] and the Mondrian [10] visualization component of Moose[4] to mention just a few. In addition there is much excellent work in creating and exploring very specific visualizations ranging from version and process histories to application overviews, module interconnections and class structures, see, for instance, [3] or the proceedings of SoftVis.[5]

Software visualization systems can be positioned in a spectrum of visualization approaches and tools that differ in the level of automation and specialization for a specific domain. At the high end of the spectrum are domain-specific visualization systems. For instance, in the business domain, a limited number of visualizations can cater for the majority of visualization needs. As a case in point, Excel[6] provides a dozen chart types including line charts, bar charts, area charts, pie charts, histograms, Gantt charts and radar charts. Although all these charts are customizable, it is complex—and requires explicit low-level programming—to add a completely new chart type to Excel. Other examples in this category are GnuPlot[7] that aims at graph plotting in the scientific domain and Many Eyes[8] that aims at charts and data visualization.

At the low end of the automation and specialization spectrum are graphics package like AWT[9], Java2D[10], SWT[11] and Processing[12] that provide low level graphics. Everything is possible but has to be implemented from scratch using low-level primitives. On a slightly higher automation level is a system like Protovis[13] [1] that uses a declarative, data-driven, approach for data visualization. There is no global state and visual attributes are combined based on inheritance. However, mapping of measures to graphical attributes has to be programmed explicitly and scaling them requires code changes.

The domain of visualizing software facts exhibits too much variability and is too diverse to be covered by a limited set of standard visualizations from the high end of the spectrum. Unfortunately, using low-level primitives to implement software visualizations from scratch is time-consuming, error-prone and requires manual integration with the fact extractor. What is needed is a software visualization framework that takes the middle ground.

As an analogy, consider the field of layouts in user-interfaces: there is no set of layouts that can cater for all layout needs but manually programming them involves tedious computations for alignments, sizing and resize behavior as well as manual integration of the user interface elements with the layout. This has been solved by automatic

[3] `http://www.bauhaus-stuttgart.de/bauhaus/index-english.html`
[4] `http://www.moosetechnology.org/news/moose-4-0`
[5] `http://www.softvis.org`
[6] `http://office.microsoft.com/en-us/excel-help/available-chart-types-HA001034607.aspx`
[7] `http://www.gnuplot.info/`
[8] `http://www-958.ibm.com/software/data/cognos/manyeyes/`
[9] `http://java.sun.com/products/jdk/awt/`
[10] `http://java.sun.com/products/java-media/2D/index.jsp`
[11] `http://www.eclipse.org/swt/`
[12] `http://processing.org/`
[13] `http://vis.stanford.edu/protovis/`

layout managers such as the ones in Tcl/TK [11]. We aim to do the same for software visualization: alleviate programmers of tedious tasks by providing just the right level of abstraction and automation. Thus we aim to develop a software visualization framework that:

- enables non-experts to easily create, combine, extend and reuse interactive software visualizations;
- integrates seamlessly with existing techniques for software analysis (parsing, pattern matching, tree traversal, constraint solving) and software transformation (rewriting, string templates).

Our main objective is to liberate the creator of visualizations from many low-level chores, such as programming of explicit coordinates, mapping metrics related to software properties to sizes of shapes and figures, and to provide high-level features instead, like figure composition, fully automatic figure placement and symbolic links between visualizations elements. Since RASCAL already provides excellent facilities for software analysis and transformation, the main challenge is therefore to design a software visualization framework that integrates well with and provides full access to what RASCAL has to offer. Our contributions can be summarized as follows:

- A compositional, coordinate-free, visualization framework that provides primitives for drawing elementary shapes and composite figures.
- Mechanisms to associate numeric scales with arbitrary figures.
- The first attempt we are aware of to decompose charts into reusable primitives.
- The integration of this framework with the RASCAL language and infrastructure thus creating a true ”One-Stop-Shop” for software analysis, transformation and visualization.
- An analysis of the software visualization domain that can form the basis for a domain-specific language for software visualization.

The paper is organized as follows. In Section 2 we identify requirements, elaborate on the design principles we have adhered to and describe the actual design we came up with. In Section 3 we present some examples and in Section 4 we draw conclusions.

2 Requirements, Design and Architecture

We will now first summarize our requirements and global design (Section 2.1) and describe our global architecture (Section 2.2). In subsequent sections we will provide more details.

2.1 Requirements

In scientific visualization, the data of interest represent objects or processes in the physical world and thus mapping of these data to visual attributes (e.g., location, size, color, line width, line style, gradient, and transparency) is mostly dictated by physical properties or appearance. In software visualization, and in information visualization in general, the data is more abstract and the mapping of the data to visual attributes is not prescribed and not easy to design. A key problem when designing a software visualization is to find good visual abstractions for software artifacts. Our software visualization framework should therefore make it easy to describe such mappings and to enable experiments to find the most appropriate ones. As a consequence the framework should provide reusable primitives for expressing such mappings. Typical use cases to be supported are the visualization of

- hierarchical and non-hierarchical software structures,
- (multi-dimensional) metrics associated with software components,
- functions that express software properties over time.

The visualization framework should not only promote creating new visualizations but should also include standard visual layouts (e.g., graphs, trees, tree maps) and primitives for common chart types (e.g., bar charts, scatter plots, Gantt charts). To achieve this goal, we reason that the software visualization framework should be *automatic and domain-specific*, *reusable*, *compositional* and *interactive*.

Automatic and Domain-specific. We aim for *as much as possible* automation and specialization for our software visualization framework. This implies eliminating low-level representation chores, such as layout and size computations, and introducing concepts that are specialized for the software visualization domain. There is no fixed border between general information visualization and software visualization and it is fruitful to exchange ideas and concepts between the two. We aim for very general solutions that have direct application in the software visualization domain.

Reusable. Creating visualizations is difficult and their reuse should therefore be enabled as much as possible. This implies that visualizations are treated as ordinary values and can, for instance, be used as arguments of a function or be the result computed by a function. The same applies to visual attributes. We want to be able to resize, parameterize and combine existing visualizations into new ones. A corollary is that arbitrary combinations and nesting of visualizations should be allowed and that the composition of visual attributes should be well-defined.

Compositional. Many graphics approaches are highly imperative. The color or line style of a global *pen* can be set and when a line is drawn these pen properties are used. As a consequence, drawing sequences that assume a different global state cannot be easily combined. We aim for declarative visualizations in which each visualization is self-contained, can be drawn independently, is easily composable and thus contributes to reusability. For some related work we refer to [5,4], HaskellCharts[14] and Degrafa[15]. Compositionality is a desirable goal but limits the solution space. The main challenge is to cover the whole spectrum of possible software visualizations with compositional primitives.

Interactive. In the use cases we envisage, overwhelming amounts of data have to be understood and Schneiderman's *Overview First, Zoom and Filter, then Details-on-demand* mantra [12] applies. We need interaction mechanisms to enable the user to start with

[14] `http://dockerz.net/twd/HaskellCharts`
[15] `http://www.degrafa.org/`

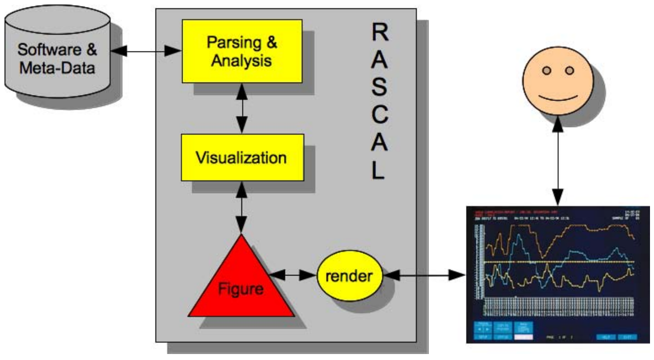


Fig. 1. Architecture of Figure visualization framework

an overview visualization and to zoom in on details. This may include mixing visualization with further software analysis and requires tight integration with an Interactive Development Environment (IDE). For instance, given an initial overview of a software system, a user may interactively select a subsystem, fill in a dialog window to select the metrics to be computed, and inspect a new visualization that represents the computed metrics. We aim for a mix of automatic, out-of-the box, interactions and programmatically-defined ones. Since we will support various interaction elements (e.g., buttons, text fields) as well as layouts we achieve integration of pure visualization and user-interface construction.

2.2 Architecture

The technical architecture of our visualization framework is shown in Figure 1. The given *Software & Meta-Data* is first parsed and then relevant analyses are performed (*Parsing & Analysis*). Parsing and analysis can be completely defined and arbitrary languages and data formats can therefore be parsed and analyzed. The analysis results are then turned into a figure (*Visualization*) and the result is an instance of the `Figure` data type, an ordinary RASCAL datatype that is used to represent our visualizations. Note, for later reference, that another data type, `FProperty`, is used to represent all possible visual properties of Figures. Figures are interpreted by a `render` function that transforms them in an actual on-screen display with which the user can interact. There is *two-way communication* between visualization and user: the visualization functions create a figure that is shown to the user, but this figure may contain call backs (RASCAL functions) that can be activated by user actions like pointing, hovering, selecting or scrolling.

2.3 Figures and Properties

As already mentioned, visualizations are ordinary values and we use the datatypes `Figure` and `FProperty` to represent them. All primitives and properties will be

Table 1. Primitives and Sample Containers

Operator	Description
`text`	A text string
`outline`	Rectangular summary (with highlighted lines) of a source code file
`box`	A rectangle (may contain nested figures)
`ellipse`	An ellipse (may contain nested figures)

Table 2. Sample Properties

Operator	Description
`id`	Name of subfigure (used for cross references)
`grow`	Horizontal (`hgrow`) and vertical (`vgrow`) size relative to children
`shrink`	Horizontal (`hshrink`) and vertical (`vshrink`) size relative to parent
`resizable`	Resizable in horizontal (`hresizable`) and vertical (`vresizable`) direction
`align`	Horizontal (`halign`) and vertical (`valign`) alignment
`lineWidth`	Width of lines
`lineColor`	Color of lines
`fillColor`	Fill color for closed figures

represented as constructor functions for the datatypes `Figure` and `FProperty`.[16] In order to be able to give meaningful examples, we give some samples of both.[17]

Figure Primitives. The primitive figures and some containers are listed in Table 1. The primitives `text` and `outline` are atomic in the sense that they cannot contain subfigures: `text` defines, unsurprisingly, text strings and `outline` is a rectangle with a list of highlighted lines inside that can be used to summarize findings on a specific source code file. The primitives `box` and `ellipse` are actually non-atomic containers that may contain a subfigure (e.g., box in a box).

Figure Properties. Some figure properties are listed in Table 2, and they can define size, spacing, and color. A figure may have a name which is defined by the `id`-property (e.g., `id("A")`) and is used to express dependencies between figures and between properties.

Several properties are related to size, alignment, and space assignment when figures are being composed. We discuss these size-related properties together in later sections. Other properties define visual aspects such as color, line style and font, or specific properties of shapes. Properties for associating interactive behavior with a specific figure are given later in Table 5.

[16] In Section 4, we speculate on using the syntax definition facilities of RASCAL and giving a textual syntax to them, thus creating a true visualization DSL.

[17] Our complete framework provides dozens of primitives and properties. In this and following tables we only list items relevant for the current paper.

Example. The expression `box(fillColor("red"), lineColor("green"))` will create a red rectangle with a green border. When rendered, this rectangle will occupy all available space in the current window.

Property Inheritance. Since figures and their subfigures usually have the same settings for most of their properties it is cumbersome to set all the properties of every figure individually. It should therefore be a possible to inherit properties from a parent figure, but a model where all properties are inherited is cumbersome as well: if a property should only apply to the current figure and not to its children then the programmer has to explicitly reset that property for all the children. Therefore we have opted for a model does not introduce this chore, but does allow inheritance:

- All properties are initialized with a standard (default) value.
- A property only applies to the figure in which it is set.
- A property can redefine the standard value of a property: that new standard value is then used in all its direct and indirect children (unless some child explicitly redefines that property itself). For example, `lineColor` defines the line color for the current figure, `std(lineColor)` defines it for all direct and indirect children of the current figure.

This model is similar in goal but simpler and more uniform than the inheritance model in cascading stylesheets in HTML, since in our model the inheritance behavior is the same for all properties, while in CSS this may differ per property as explained by Lie [8].

2.4 Figure Composition and Layout

In many approaches to graphics and visualization *coordinates* and *explicit sizes* are the primary mechanisms to express the layout of a visual entity. For instance, a rectangle is specified by its upper-left corner, width and height, or alternatively, by its upper-left and lower-right corner. Although this is common practice, there are disadvantages to this approach [2]:

- Explicit coordinates and sizes lead both to tedious computations to achieve a simple layout and to manual programming of resize behavior. This conflicts with our goals of automation and re-use.
- Explicit coordinates and sizes do not show the spatial relationships between the elements in a layout, this makes re-use and interaction more difficult.

Therefore, we have chosen to avoid explicit coordinates and provide layouts based on figure composition. The position of a figure is described by nesting figures, for example an ellipse inside a box is written as `box(ellipse())` or by using the composition operators listed in figure Table 3. The most fundamental operators provide horizontal, vertical and overlayed (stacked) composition of figures. As an example, a resizable Dutch flag (Figure 2) can be created as follows: `vcat([ box(fillColor(c)) | c <- ["red","white","blue"] ])`.

The size of a figure should depend on its context. In this way we can use the same figure in different contexts, even though these contexts have different sizing requirements. One way of expressing the size of a figure in terms of its context is through the

Table 3. Composition Operators

Operator	**Description**
`hcat`	Horizontal composition, grid with one row
`vcat`	Vertical compostion, grid with on collumn
`hvcat`	Horizontal and vertical compostion (resembling placing words in a text paragraph)
`overlay`	Stacked composed, i.e., figures are overlayed in the z-dimension.
`grid`	Place figures in a grid
`pack`	Place figures as close together as possible (bin packing)
`graph`	Place figures and edges in graph layout
`tree`	Place figures and edges in tree layout
`treemap`	Place figures in a treemap layout

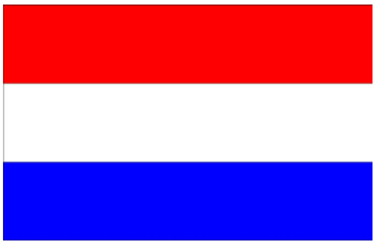


Fig. 2. Dutch Flag

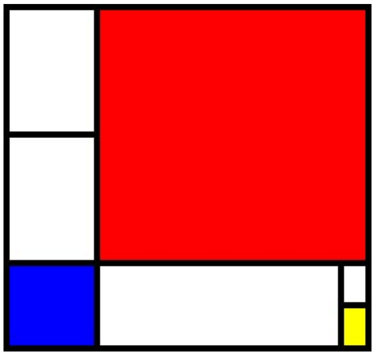


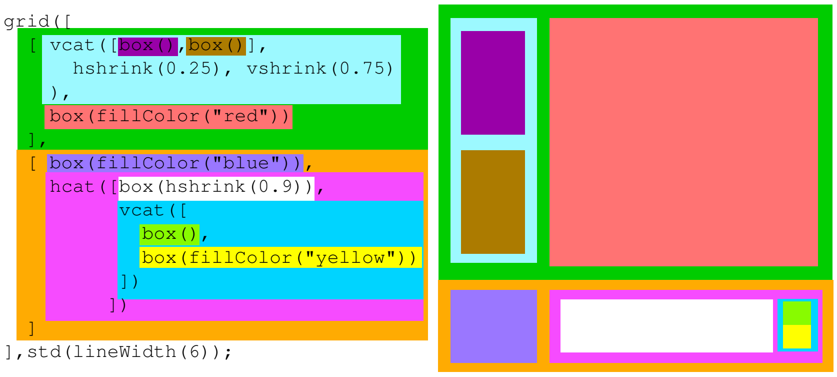
Fig. 3. Our version of *Composition II in Red, Blue, and Yellow* by Piet Mondriaan, 1930

`hshrink` and `vshrink` properties, which declare how much a figure shrinks relative to the horizontal or vertical size of its parent. The property `shrink` is a shorthand for setting both `hshrink` and `vshrink`.

As an example, consider our version of the painting *Composition II in Red, Blue, and Yellow* by Piet Mondriaan in Figure 3. Using our layout and sizing mechanisms this painting can be concisely described by the code in Figure 4. The sizes of the boxes are described in terms of their parents. If no sizing properties are given, the figure is given the size that is available.

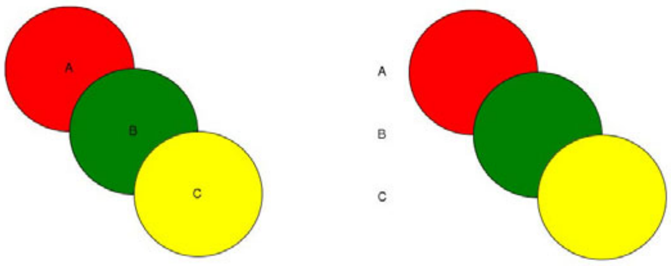
This wellknown way of defining sizes is, for instance, also used in HTML tables. Another, novel, way of defining sizes in term of their context is through the `hgrow` and `vgrow` properties: which declares how much a figure grows relative to the horizontal or vertical size of its children. For example if we want a box containing the text “Rascal” that is twice as wide and three times as high as the enclosed text we could define this by `box(text("Rascal"),hgrow(2.0),vgrow(3.0))`.

To specify where in the available space a figure is positioned the `halign` and `valign` properties are used. Here 0.0 means completely on the left side or top side



```
grid([
  [ vcat([box(),box()],
      hshrink(0.25), vshrink(0.75)
    ),
    box(fillColor("red"))
  ],
  [ box(fillColor("blue")),
    hcat([box(hshrink(0.9)),
            vcat([
              box(),
              box(fillColor("yellow"))
            ])
          ])
  ]
],std(lineWidth(6));
```

Fig. 4. Code creating Mondriaan painting, the colors relate each piece of code to the corresponding area in the layout



(a) An overlay (b) Using a screen

Fig. 5. The use of overlay and screen

and 1.0 means completely on the right side or bottom side. Shorthands such as `left()` are available. Consider the following example (Figure 5a), where the `overlay` composition is used which stacks figures:

```
overlay([
  ellipse(text("A"), left(),top(),
                fillColor("red"),shrink(0.6)),
  ellipse(text("B"), center(),
                fillColor("green"),shrink(0.6)),
  ellipse(text("C"), right(), bottom(),
                fillColor("yellow"),shrink(0.6))
])
```

Now suppose we want to display the same image as our last example, but with the labels on the left as displayed in Figure 5b. We could achieve this effect by positioning the labels in a separate overlay and horizontally composing these two overlays. However this means that the user must manually specify the alignment of the labels. To raise the level of automation in such cases we introduce the notion of a *screen*. A screen is a horizontal or vertical line on which figures can be projected. Thus to obtain the picture displayed in Figure 5b we place a screen on the left of the overlay and then project a label from each ellipse on this screen in the following way:

```
leftScreen("s",
  overlay([
    ellipse(project(text("A"),"s"), left(),top(),
                fillColor("red"), shrink(0.6)),
    ellipse(project(text("B"),"s"), center(),
                fillColor("green"), shrink(0.6)),
    ellipse(project(text("C"),"s"), right(), bottom(),
                fillColor("yellow"), shrink(0.6))
  ])
);
```

There are also composition for laying out arbitrary figures according to their relation in a graph or a tree since this common in software visualization. We argue that with these special purpose layout operators and the general purpose composition operators described above the user can describe most layouts needed in software visualization in a concise, declarative and reusable way.

2.5 Scales of Measurement

Representing software facts requires mapping measurements to scales. Mapping of software measures to visual properties was pioneered in polymetric views as described in [7]. We want to provide a similar mechanism but make the mapping from measurement to graphic representation explicitly, so that it can later be scaled or manipulated interactively. Traditionally, the following scales of measurement are distinguished [13]:

- A *nominal* scale consist of unordered data points and only equality of data points is defined. This kind of data can, for instance, be represented by text labels or color codes.
- An *ordinal* scale consists of ordered data points; this implies that a comparison between data points is possible but that differences between values are not meaningful.
- An *interval* scale consists of ordered values, with a constant scale, but no natural zero. The typical example are temperatures and dates.
- A *ratio* scale consists of ordered, constant scale, values with a natural zero. Examples are height and age.

We distinguish the following aspects of a scale (of which the last three are inspired by [9]):

- Its classification in the above four categories.
- The figures that are to be mapped to the scale. For example, the bars in a bar chart.

```
vcat([
  hcat( [ box(text(n),fillColor(convert(t,"accessType")))
        | <n,t> <- [<"equals","public">,
                    <"intersects","protected">,
                    <"toString","public">,
                    <"getPassword","private">,
                    <"union","protected">]
         ]),
  colorPallette("Access types of methods",
                "accesType",
                hshrink(0.5))]
)
```

Fig. 6. Example using `colorPallette`

– The specific property whose value is to be mapped to the scale. For example, the height of a bar in a bar chart or the coordinates of a point in a function or scatter plot.
– The figure that explains the mapping. For example, an axis or a color legend.

We introduce a figure type for each kind of scale of measurement and annotate values of figure properties with the name given to the desired scale figure to establish the mapping. All scales are figures, and can be placed in the visualization just like ordinary figures. They are visualized as an explanation of the mapping, i.e., axis or color legend. Our ambition is to support all four scales of measurement as built-in primitives, but currently we only support the nominal and the ratio scale.

Nominal scales. Assume that a list of Java method names and their access modifiers are given. We want to visualize each method as a named box with a color that represents its access modifier. The access modifiers can be represented by a nominal scale in which each access modifier is mapped to a different color.

Figure 6 shows how this can be achieved. First we create horizontally concatenated boxes with the method name as text and a fillColor that is determined by the value of each access modifier converted to a nominal scale with name `accessType`. Below these boxes, we then place the nominal scale `colorPallette` with a title and — as expected — the name (`accessType`); it will convert the nominal values that were added in the box declarations to a color on the pallette. The result is shown in Figure 7.

Interval Scales: Axes. Figure 8 gives the anatomy of a single bar in a typical bar chart: the bar has a height (a numerical value) that should be mapped to units on the vertical axis, and the bar should also be mapped to a nominal label on the horizontal axis.

To map numeric values, we introduce the notion of an *axis*. An axis takes care of the proper interval and scale of values, and of major and minor tick marks. It has a name and contains a figure with subfigures whose dimensions are mapped using the `convert` operator. Axes exist in different flavors depending on their placement relative to the list of figures.

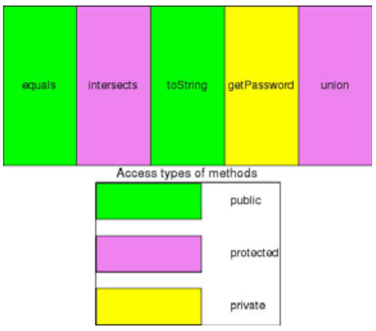


Fig. 7. A nominal scale using a color pallette

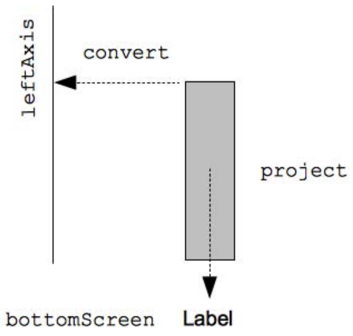


Fig. 8. Anatomy of a bar in a bar chart

```
Figure hBarChart(map[str,num] vals){
    return bottomScreen("categories",leftAxis("y",
            hcat([box(height(convert(vals[k],"y")),
                      project(text(k),"categories"),
                      fillColor("blue"))
                 | k <- vals], hgrow(1.2))
        ));
}
```

Fig. 9. `hBarChart`: a simple bar chart

For instance,

```
leftAxis("y", [ box(height(convert(10, "y"))),
                box(height(convert(15, "y")))
              ])
```

creates a vertical axis with two boxes to the right of it.

Axis and screen form the building blocks for many common chart types and we illustrate in Figure 9 how to apply them to create simple bar charts in the following function `hBarChart` that takes a map from strings to numbers and returns a bar chart that visualizes this information. See Section 3.1 for an application of this function.

2.6 Figure Interaction

A plotting package like GnuPlot, takes a description of a desired plot and delivers a static rendering of it. Given our interaction requirements (Section 2.1) we add properties and operators for interaction as listed in Tables 5, and respectively, 6. The properties allow associating interactive behavior with a specific figure, such as showing a second figure when the mouse is over the first one or handling a mouse click on a figure. The interaction operators represent separate user-interface elements like buttons, checkboxes and text fields, to which RASCAL *closures* (functions and their local context) can be

Table 4. Axes and Screens

Operator	**Description**
`leftAxis`	A vertical axis to the left of a figure, similar: `rightAxis`, `bottomAxis`, `topAxis`
`convert`	Convert a size to a datapoint on an axis
`bottomScreen`	Horizontal projection screen at the bottom of a figure, similar: `topScreen`, `leftScreen`, `rightScreen`
`project`	Project a figure to a screen

Table 5. Interaction properties

Operator	**Description**
`mouseOver`	Add a figure when mouse is over current subfigure
`onClick`	Handle mouse clicks
`onMouseOver`	Handle the mouse entering the current subfigure
`onMouseOff`	Handle the mouse leaving the current subfigure

Table 6. Interaction Figures

Operator	**Description**
`computeFigure`	Compute a new (sub)figure triggered by interaction
`button`	Button with call back
`textField`	Text entry field with call back
`combo`	Combo box with call back
`choice`	List of choices with call back
`checkbox`	A check box with call back

attached as call backs. Although RASCAL has value-based semantics, it does allow assignment to variables and this can be used to represent the state of the user-interface inside the RASCAL program.

Examples. In Figure 10 we illustrate how to define a function that returns a figure property, in this case, a `mouseOver` property that will display a yellow box with text when the mouse hovers over the figure which has this property. The box will be 1.2 times larger than the text and it is not resizable.

In Figure 11 the creation of a textfield is illustrated. Note how the `textfield` property has a closure parameter

```
void(str s){TERM = s;}
```

that acts as call back function. It assigns to the environment variable `TERM` and in this way global state can be maintained across calls to call back functions.

```
public FProperty popup(str S){
   return mouseOver(box(text(S),
                    fillColor("lightyellow"),
                    grow(1.2),
                    resizable(false)));
}
```

Fig. 10. Defining a popup

```
    str TERM = ""; // Initial search term
    searchField =
         hcat([ text("Enter search term:"),
                textfield("<TERM>", void(str s){TERM = s;})
              ]);
```

Fig. 11. Defining a text entry field

3 Examples

3.1 Bar Chart of File Name Extensions

The first task we want to solve is to extract all files from an Eclipse project, count the file name extensions (i.e., different file types) and draw a bar chart of the result. The code is shown in Figure 13 and re-uses the `hBarChart` function defined earlier in Figure 9. The auxiliary function `getExtensions` use deep pattern matching (`/`) to search for all files in the project and to count their frequencies. It returns a map from extensions to

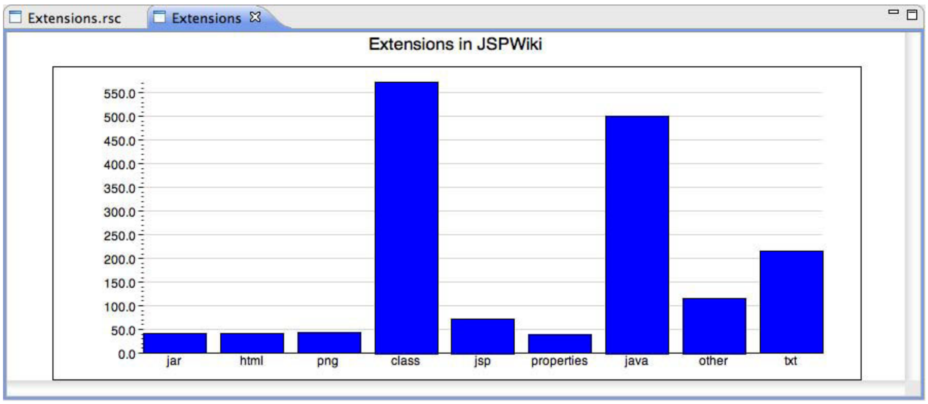


Fig. 12. Bar chart with frequencies of file name extensions in JSPWiki

frequencies.[18] In the statement `m[l.extension]?0 += 1;` the current value associated with the value of `l.extension` in table `m` is incremented. The binary *undefined operator* `?` caters for the case that that key value is not yet in the table and uses 0 instead. Application to a sample project as in `drawBarChart(|project://JSPWiki|)`, and rendering the result gives the bar chart shown in Figure 12.

```
public Figure drawBarChart(loc project) {
  e = getExtensions(getProject(project));
  return vcat([ text("Extensions in <project.host>",
                                fontSize(17)),
                box(hBarChart(e),grow(1.1))]);
}

// Extract all file extensions
public map[str, num] getExtensions(Resource r) {
  m = ();
  for (/file(loc l) := r)
      m[l.extension]?0 += 1;
  return m;
}
```

Fig. 13. Visualizing file name extensions

3.2 Search and Browse Files

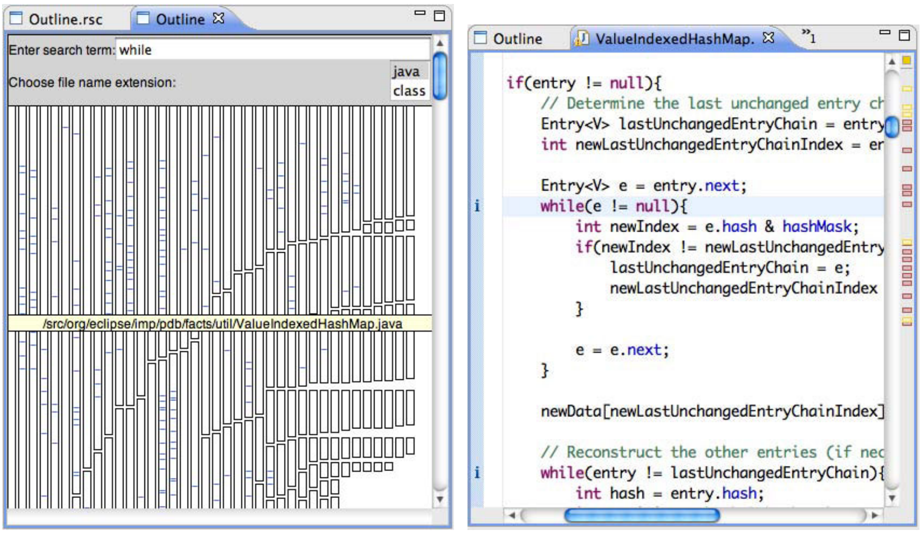
The second final task we want to solve deals with search and file exploration. It integrates analysis, visualization and user interface construction and consists of the following steps:

- Present the user with a search field to enter a search term.
- Present the user with an option to search Java files or class files.
- Present the search results in the form of an outline per file, with colored lines representing hits.
- When hovering over an outline, the corresponding file name is shown as a popup.
- When clicking on an outline, a new editor is opened for the corresponding file, with the occurrences of the search term highlighted.

Figure 14a shows an example of visualizing the results of a query for the term ”while”, and Figure 14b shows an editor that appears when clicking on one of the file outlines in Figure 14a.

The function `mkOutline` (Figure 15) takes the location of a source file, a word to search for, a message to attach to each line, and a scale factor to compute the relative position of hits in the file. It returns an outline figure, with line information associated with it. This outline has two properties that implement interaction:

[18] Extensions with a frequency below 2% are collected in the category ”other”; this is not shown in the code.



Enter search term: while

Choose file name extension: java class

/src/org/eclipse/imp/pdb/facts/util/ValueIndexedHashMap.java

```
if(entry != null){
    // Determine the last unchanged entry ch
    Entry<V> lastUnchangedEntryChain = entry
    int newLastUnchangedEntryChainIndex = er

    Entry<V> e = entry.next;
    while(e != null){
        int newIndex = e.hash & hashMask;
        if(newIndex != newLastUnchangedEntry
            lastUnchangedEntryChain = e;
            newLastUnchangedEntryChainIndex
        }

        e = e.next;
    }

    newData[newLastUnchangedEntryChainIndex]

    // Reconstruct the other entries (if nec
    while(entry != lastUnchangedEntryChain){
        int hash = entry.hash;
```

(a) Search results for ”while” (b) Clicking on outline opens editor

Fig. 14. Browsing search results

```
public Figure mkOutline(loc f, str word, str msg, int scale){
  lineInfo = [];
  lines =  readFileLines(f);
  for(i <- index(lines)){
    if(/<word>/ := lines[i])
      lineInfo += info(i + 1, msg);
  }
  return outline(lineInfo,
                 size(lines),
                 size(5, size(lines)/scale),
                 popup(f.path),
                 onClick(void () {edit(f, lineInfo);}));
}
```

Fig. 15. `mkOutline` create an outline with search results

– a `mouseOver` that displays the file name when the mouse hovers over this outline; this re-uses the function `popup` discussed earlier in Section 2.6
– an `onClick` properties that defines a parameterless function to be called when a mouse click occurs on this outline. This function calls, in turn, the library function `edit` that opens a source code editor for the given file.

The function `find` (Figure 16), searches through all files with the right file extension, applies `mkOutline` to each file and `packs` the resulting outlines as densely as possible. Finally, the function `searchGUI` (Figure 17) creates the user-interface and uses `find` as utility.

```
public Figure find(loc project, str word, str suffix){
 if(word == "")
    return
     box(text("No results", center()), fillColor("silver"));
 P = getProject(project);
 outlines = [ mkOutline(l, word, "Found: <word>", 2)
            | /file(loc l) := P, l.extension == suffix
            ];
 return pack(outlines,  top(), left(), gap(3));
}
```

Fig. 16. `find`: top level function to create the visualization

```
public void searchGUI(){
    // Create text field for search term
    str TERM = ""; // Initial search term
    searchField =
         hcat([ text("Enter search term:"),
                textfield("<TERM>", void(str s){TERM = s;})
              ]);
    // Create choice box for file extensions
    str EXT = "java"; // Initial file name extension
    extField =
         hcat([ text("Choose file name extension:"),
               choice(["java", "class"], void(str s){EXT = s;})
             ]);
    // Combine both fields in a box
    pane = box(vcat([searchField, extField]),
               fillColor("lightgrey"),
               stdVresizable(false), vgrow(1.1));
    // Create computed figure for visualization of result
    result = computeFigure(
               Figure(){ return find(project, TERM, EXT);}
             );
    // Render the pane and search result
    render("Outline", vcat([pane, result],  left()));
}
```

Fig. 17. `searchGUI`: create the user-interface

4 Conclusions

We have presented a high-level overview of the RASCAL visualization framework that we are currently developing and we hope that this overview has convinced you that a declarative, coordinate-free, visualization approach is both feasible and highly applicable. Once the framework has stabilized, we envisage to explore how interaction facilities can be further extended and how animation can be added. We are also planning to systematically describe the most popular chart types with the primitives presented here as starting point. A series of case studies will help to assess the applicability of our framework and to further improve it.

This effort acts as a domain analysis for software visualization and we expect that the concepts we have identified can form the basis for a true DSL for software visualization. You will have observed that we have presented the framework as an abstract data type with prefix constructor functions and this leaves the advanced RASCAL facilities for syntax definition and parsing completely unused. Another next step is therefore to design a concise textual syntax for the visualization primitives presented here and to integrate them even further in the RASCAL language.

Acknowledgments. We thank our colleagues from the RASCAL team for the brainstorms, suggestions and support. Jurgen Vinju commented on drafts of this paper.

References

1. Bostock, M., Heer, J.: Protovis: A Graphical Toolkit for Visualization. IEEE Transactions on Visualization and Computer Graphics 15(6), 1121–1128 (2009)
2. Coutaz, J.: A layout abstraction for user-system interface. SIGCHI Bull. 16, 18–24 (1985)
3. Diehl, S.: Software visualization: visualizing the structure, behaviour, and evolution of software. Springer (July 2007)
4. Elliott, C.: Functional images. In: The Fun of Programming. Cornerstones of Computing. Palgrave (March 2003)
5. Finne, S., Peyton Jones, S.: Pictures: A simple structured graphics model. In: Glasgow Workshop on Functional Programming (January 1995)
6. Klint, P., van der Storm, T., Vinju, J.: EASY Meta-Programming with Rascal. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) Generative and Transformational Techniques in Software Engineering III. LNCS, vol. 6491, pp. 222–289. Springer, Heidelberg (2011)
7. Lanza, M., Ducasse, S.: Polymetric views - a lightweight visual approach to reverse engineering. IEEE Transactions on Software Engineering 29(9) (September 2003)
8. Lie, H.: Cascading Style Sheets. PhD thesis, Faculty of Mathematics and Natural, Sciences University of Oslo (2005)
9. Lucas, W., Shieber, S.M.: A Simple Language for Novel Visualizations of Information. In: Filipe, J., Shishkov, B., Helfert, M., Maciaszek, L.A. (eds.) Software and Data Technologies. CCIS, vol. 22, pp. 33–45. Springer, Heidelberg (2009)
10. Meyer, M., Gîrba, T., Lungu, M.: Mondrian: an Agile Information Visualization Framework. In: Proceedings of the 2006 ACM Symposium on Software Visualization - SoftVis 2006, pp. 135–144. ACM Press, New York (2006)
11. Ousterhout, J.: Tcl and the Tk Toolkit. Addison-Wesley, Reading (1994)
12. Shneiderman, B.: The eyes have it: A task by data type taxonomy for information visualizations. In: Proceedings of IEEE Symposium on Visual Languages, pp. 336–343 (1996)
13. Stevens, S.S.: On the Theory of Scales of Measurement. Science, New Series 103(2684), 677–680 (1946)
14. Storey, M.-A.D., Wong, K.: Rigi: A Visualization Environment for Reverse Engineering. In: Proceedings of the 1997 (19th) International Conference on Software Engineering, pp. 606–607. ACM (1997)

A Dedicated Language for Context Composition and Execution of True Black-Box Model Transformations

Andreas Seibel, Regina Hebig, Stefan Neumann, and Holger Giese

Hasso Plattner Institute at the University of Potsdam
Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany
{forename.surname}@hpi.uni-potsdam.de

Abstract. Model-Driven Engineering (MDE) automates development activities by employing model transformations. Thereby, a plethora of model transformation approaches with individual capabilities have been developed. In certain cases, complex and automated MDE activities require the interaction of various, potentially heterogeneous, model transformations. This can be achieved by a loosely coupled and highly cohesive composition of model transformations implemented in different model transformation languages. However, existing approaches either do not support context composition, using other model transformations as additional context, or they violate the important black-box principle because they require adapting model transformations for context composition. In this paper, we present a dedicated model transformation composition framework (MoTCoF) that does not require the adaptation of model transformations and, thus, treats model transformations as true black-boxes. We illustrate our approach with an application example taken from an industrial case study.

Keywords: model transformation, composition, model-driven engineering, traceability.

1 Introduction

Today, Model-Driven Engineering (MDE) promises to improve software development activities by employing models and model transformations as first-class citizens.[1] As an essential part of recent MDE environments, model transformations are not applied in isolation but are rather in combination. Due to changing requirements to the software that is built, model transformations has to evolve properly because evolving meta models of design artifacts.

However, it is not always feasible to simply append new functionality to existing model transformations, since the employed model transformation languages

[1] We consider model transformations as any model operation that manipulates models or model elements; e.g., derivation, synthesis, reverse engineering, migration, optimization, adaptation, refactoring, etc. (cf. [1]).

might be insufficient for realizing the new requirements. For example, in an industrial case study in cooperation with dSPACE[2] we have developed a tool chain that transforms SysML[3] models into AUTOSAR[4] models within Eclipse[5] and further transforms these AUTOSAR models into SystemDesk[6] conform AUTOSAR models (cf. [2]). Because of the new timing extension of AUTOSAR, we now also had to transform textual SysML timing requirements into structural AUTOSAR latency timing constraints. Yet, without making modifications to the employed model transformation language (triple graph grammars (TGG) [3]), we would not be able to extend the existing model transformation. Thus, we needed to seamlessly compose existing model transformations. Since re-implementing a complete model transformation in another model transformation language is time-consuming, error-prone and expensive, this leads to two major challenges.

Firstly, it must be possible to define reusable, modifiable and extendible model transformations. This can be reached by promoting model transformations with loose coupling and high cohesion. The term coupling is *a measure of the strength of interconnection between one module and another* [4]. Thus, a model transformation should have as few interconnections as possible. The term cohesion is *the degree of functional relatedness of processing elements within a single module* [4]. Only focusing on a single concern – the actual transformation task – renders a model transformation as highly cohesive. Thus, any additional concern, i.e., navigating to the right application context or focusing on composition concerns, decreases cohesion. Secondly, the ability to compose model transformations implemented in other model transformation languages must not be restricted. This requires the consideration of model transformations as true black-boxes.

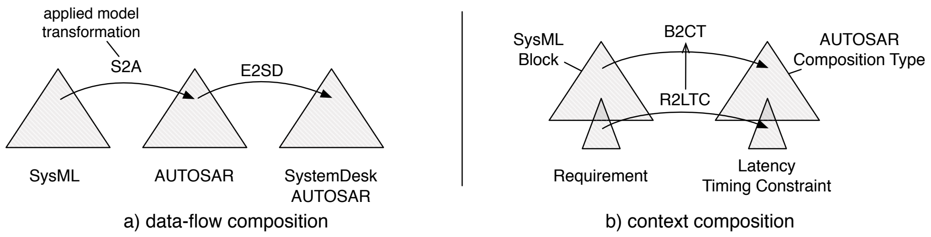


Fig. 1. Principle of data-flow and context composition

Existing approaches to the composition of black-box model transformations can be distinguished between two types of composition that we call data-flow composition and context composition. The principles of these compositions are illustrated in Fig. 1, which reflects our application example we use throughout this paper.

[2] http://www.dspace.com

[3] http://www.sysml.org/specs.htm

[4] AUTOSAR 4.0; http://autosar.org/

[5] http://www.eclipse.org

[6] SystemDesk is an environment from dSPACE to model embedded systems.

Model transformation chains (i.e., UniTI [5]) or workflows are implementations of data-flow compositions. A data-flow composition between model transformations exists whenever the output of one model transformation is the input of another model transformation. Fig 1 a) shows a SysML to AUTOSAR (S2A) model transformation and an Eclipse to SystemDesk (E2SD) model transformation. A data-flow composition exists between S2A and E2SD because S2A produces an AUTOSAR model that is consumed by E2SD to produce a SystemDesk conform AUTOSAR model.

A major benefit of data-flow composition is that it does not require an explicit composition concept but rather implicitly sharing models is sufficient to establish a data-flow composition. Thus, the data-flow composition of independent model transformations is still possible. However, a drawback of data-flow composition is that the composition of model transformations is principally only possible if the model transformations act on the same level-of-granularity (e.g., between models). In our case study, we defined the TGG model transformation (S2A) between SysML models and AUTOSAR models, but we defined the extending model transformation on a different level-of-granularity (between SysML requirements and AUTOSAR latency timing constraints). The data-flow composition of these two model transformations requires lifting the extending transformation to the model level which implies the second drawback of data-flow compositions; the extending model transformation has to implement additional navigation concerns to find the right application context (e.g., the model elements that are consumed and produced); the extending model transformation has to replicate concerns from the model transformation that it extends. Thus, the extending model transformation needs to re-produce which SysML block is transformed into which AUTOSAR composition type. Only then is it possible to correctly transform a SysML requirement that relates to a SysML block into an AUTOSAR latency timing constraint that relates to an AUTOSAR composition type. Reproducing the decisions of the extended model transformation decreases cohesion and implicitly augments coupling.

In contrast to data-flow composition, context composition basically defines that the application of a model transformation requires the application of another model transformation as additional context for its own application. Thus, an extending model transformation has access to traceability information generated by the application of another model transformation. Thus, context composition overcomes the aforementioned issues of the data-flow composition by relying on the application of another model transformation and not only on shared models or model elements. Fig. 1 b) shows the application of a model transformation Block to CompositionType (B2CT) that transforms a SysML block into an AUTOSAR composition type. Now, another model transformation Requirements to LatencyTimingConstraint (R2LTC) transforms a SysML requirement of a SysML block into an AUTOSAR latency timing constraint of a corresponding AUTOSAR composition type. Thus, R2LTC can directly use B2CT as context because B2CT provides the necessary application context – a SysML block and a corresponding AUTOSAR composition type – to start the transformation R2LTC. An extending model transformation might be applied multiple times in the context of another transformation because multiple SysML requirements might be contained by one SysML block.

In this paper we show a model transformation composition framework called MoTCoF that supports data-flow and context composition of true black-box model transformations implemented in different model transformation languages. We achieve this by not requesting model transformations to interpret traceability information for proper context composition. MoTCoF preserves the black-box property of model transformations by shifting the interpretation of the traceability information from the model transformations to the composition framework. Therefore, the composition framework applies model transformations into the required application context by unpacking the traceability information. In addition, MoTCoF uses pre-conditions (instantiation checks) to define fine-grained conditions for proper application of model transformations. These pre-conditions are defined separately from the model transformation to ensure separation of concerns. Thus, MoTCoF takes total control of creating traceability information as well as applying model transformations into the required application context. Nevertheless, in certain scenarios model transformations create proprietary traceability information that should also be used for context composition (i.e., model transformations that create traceability information as a by-product of their application). Therefore, MoTCoF 'translates' proprietary traceability information into traceability information of our composition framework. This will be illustrated by means of an application example.

In the following, we first explain our industrial case study including an application example in Section 2. In Section 3, we elucidate our dedicated language to specify sufficient interfaces for model transformations (modules) and how to define context compositions as well as instantiate data-flow compositions. In Section 4, we show how to execute model transformations that are instantiated in a data-flow as well as defined in a context composition. We delimit our approach against related work in Section 5 and, finally, conclude our paper in Section 6.

2 Case Study

In an industrial case study in cooperation with dSPACE, we have developed a tool chain that is able to transform SysML models within Eclipse into AUTOSAR conforming models within SystemDesk. In systems engineering, the engineers use SysML, which reuses a subset of UML[7], for developing embedded systems. In the automotive domain, parts of these SysML models, which are important for software engineering, are later refined in AUTOSAR for further software development. Thus, automotive companies that develop the complete system architecture or subsystems using SysML need to transform relevant parts of those models to an AUTOSAR model.

[7] UML = Unified Modeling Language; http://www.omg.org/spec/UML/

Fig. 2 illustrates our developed tool chain. It employs SysML models and AUTOSAR models in an EMF[8] compatible representation. We use TGGs to realize the model transformation between these models (S2A). Another model transformation (E2SD) bridges the technological gap between Eclipse + EMF and the tool SystemDesk from dSPACE.

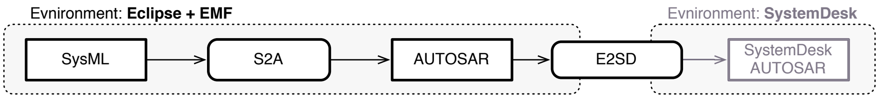


Fig. 2. Complete tool chain of our case study

In the current release of AUTOSAR, defining timing constraints becomes possible by a timing extension [6]. Thus, our goal was to extend the model transformation S2A, such that SysML requirements are transformed into AUTOSAR latency timing constraints. Since we invested much effort in the development of the model transformation S2A, we desired to reuse the model transformation for this purpose. However, the requirements in SysML are formulated textually. Thus, our existing model transformation cannot handle these requirements, because it is a structural model transformation language that does not support text parsing. For this purpose, we had to extend the existing model transformation by another technology for handling the transformation of textual requirements without adapting the existing model transformation technology. Thus, we have implemented a separate model transformation in Java, which is well-suited for text parsing. This additional model transformation only transforms a single requirement into a corresponding latency timing constraint. The composition approach should be responsible for finding all needed application contexts.

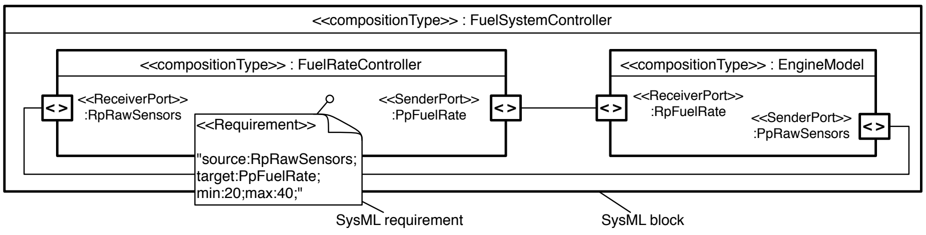

<<compositionType>> : FuelSystemController
<<compositionType>> : FuelRateController
<<compositionType>> : EngineModel
<<ReceiverPort>> :RpRawSensors
<<SenderPort>> :PpFuelRate
<<ReceiverPort>> :RpFuelRate
<<SenderPort>> :PpRawSensors
<<Requirement>>
"source:RpRawSensors; target:PpFuelRate; min:20;max:40;"
SysML requirement
SysML block

Fig. 3. Software architecture of the fuel system controller encoded in SysML

Fig. 3 shows the software architecture of a fuel system controller as a SysML model including a SysML requirement, which acts as our application example.[9] The FuelRateController computes the fuel injection rate, which depends on the current speed of the engine. The EngineModel represents the engine. Because of

[8] EMF = Eclipse Modeling Framework; http://www.eclipse.org/modeling/emf

[9] This is an example shipped with SystemDesk.

the given real-time requirements of an automotive engine, the FuelRateController is subject to specific timing requirements that are reflected in the form of the shown SysML requirement.

3 Specification and Instantiation of Modules

The methodology of MoTCoF includes two manual activities, and one automated activity. These activities interact as Fig. 4 shows.

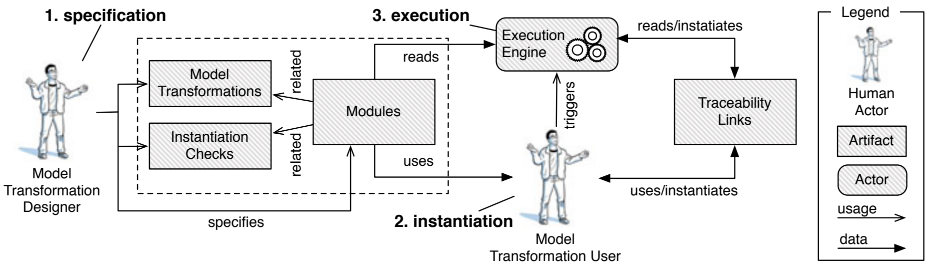


Fig. 4. Overview of the MoTCoF approach

First, a model transformation designer is specifies model transformations and instantiation checks using different model transformation technologies (languages). For each of the specified model transformation and instantiation checks – as well for reused ones – the model transformation designer specifies modules, based on the types of the consumed and produced modeling artifacts.[10] Finally, the model transformation designer specifies context compositions between modules that encapsulate the model transformations.

Second, a model transformation user can use model transformations by instantiating earlier specified modules. Instances of modules are technically traceability links, which are defined between concrete modeling artifacts that are consumed and produced by the related model transformation. MoTCoF eases the user when instantiating modules by leveraging module information (it only shows compatible modeling artifacts when instantiating modules). For the application of a data-flow composition the model transformation user simply instantiates all modules to be composed on shared source and target modeling artifacts. For the application of a context composition, it is sufficient to instantiate the most high-level module that does not extend any other module.

Third, the execution engine is executing all necessary model transformations that are in data-flow compositions and context compositions. The execution engine is explicitly triggered by the model transformation user and is explained in Section 4 in more detail.

[10] We use the term modeling artifact as a synonym for model and model element.

3.1 Specification of Modules

MoTCoF does not provide a model transformation language but rather a model transformation composition language. Thus, existing model transformation languages complement MoTCoF. The model transformation designer integrates existing model transformations or newly specified model transformations by specifying modules within MoTCoF. A module is a suitable interface of the model transformation it represents. In addition, modules contain information about context composition, which defines the required context. The execution engine needs the context composition information for proper execution of the model transformations.

A module has a set of parameter types. A parameter type is either source or source & target, or target of a module. A parameter type defines the types of modeling artifacts that are consumed and/or produced by the model transformation related to the module. A modeling artifact type is any meta model or meta model element.[11] A source parameter type defines that an associated model transformation consumes an instance of the related modeling artifact type and potentially instances of contained modeling artifact types. A source & target parameter type defines that an associated model transformation requires an instance of the related modeling artifact type and further may manipulate the instance as well as instances of modeling artifact types that are contained by the related modeling artifact type. Finally, a target parameter type defines that an associated model transformation is responsible for creating or overwriting an instance of the related modeling artifact type and potentially instances containing modeling artifact types.

A module can be explicitly associated to a model transformation for which it is the black-box representation. By using technology adapters, MoTCoF supports the application of model transformations specified in different model transformation languages.[12] However, as we will show later a module can also be specified without being associated to a model transformation, which we consider as a 'virtual' module.

Fig. 5 shows two modules (S2A and R2LTC) from our application example encoded in concrete syntax of our module language. S2A is the representation of our TGG model transformation (denoted by the (T) symbol), which transforms SysML models into AUTOSAR models. It additionally creates a Correspondence model that contains proprietary traceability information created by the TGG model transformation when applying it. R2LTC is a module that represents the model transformation, which only transforms a SysML requirement into an AUTOSAR latency timing constraint. Because a latency timing constraint needs to be contained by a composition type, the transformation needs a composition type as additional source & target. It must be a source & target because the

[11] We assume that meta models are structured like trees. Ecore models are always trees and, thus, fulfill our assumption.

[12] Because of the lack of space, we do not explain the technology adapters that are responsible for applying model transformations. We also do not show the concrete model transformations that we use throughout this paper.

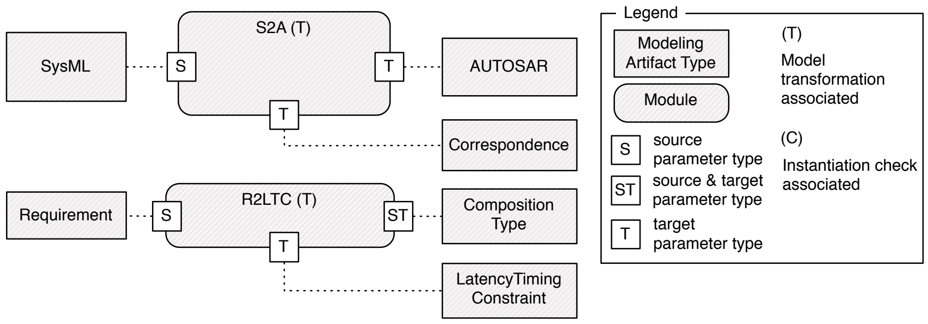


Fig. 5. Primary modules from our case study's application example

model transformation will append the created latency timing constraint to the given composition type. It has to be noted that the specification of both model transformations is completely independent.

A module can be associated with an instantiation check (denoted by the (C) symbol). The execution engine uses instantiation checks as pre-conditions for the instantiation of a module into some application context. If the instantiation check evaluates to true, the execution engine can correctly instantiate the related module into the given application context. Instantiation checks are considered as black-boxes, too. Thus, the model transformation designer can specify them in any model transformation language because we use the same technology adapters (e.g., OCL[13]). Furthermore, instantiation checks must not have any side effects.

Instantiation checks and model transformations can also be combined within a singe module. This way, we can separate between reasoning about the correctness of an application context and the operational semantic of model transformation implementation, which makes it more reusable.

3.2 Specification of Context Compositions

After the specification of modules, a model transformation designer specifies all necessary context compositions between these modules. Concerning our application example, R2LTC should be composed into the context of S2A because it is considered as an extension of S2A. Up to this point, R2LTC cannot be appropriately composed with S2A because there is a granularity gap between the module S2A which is defined too coarse-grained and R2LTC which is defined too fine-grained. To overcome this issue, we define another (virtual) module that acts as 'glue code' between S2A and R2LTC. This virtual module will ensure that R2LTC is only applied between SysML blocks and corresponding AUTOSAR composition types. Otherwise, we might transform requirements of blocks into latency timing constraint of non-corresponding composition types, which would be incorrect.

[13] OCL = Object Constraint Language; http://www.omg.org/spec/OCL/

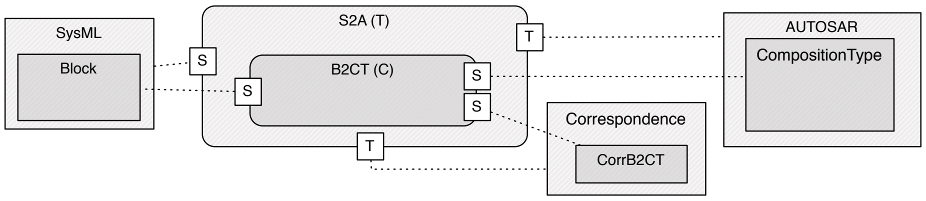


Fig. 6. S2A exports B2CT

Fig. 6 shows how S2A exports the virtual module B2CT.[14] Exporting a module is also a kind of context composition. It is called export because virtual modules only exist in the context of exactly one other module (B2CT only exists within S2A). Virtual modules can be used by other modules as context.

The exported module B2CT is not associated with a model transformation, but with an instantiation check. This instantiation check is responsible for ensuring that a block and a composition type, related to B2CT, indeed correspond. Therefore, the instantiation check could also use similarity heuristic, e.g., name equivalence. Because the TGG model transformation creates proprietary traceability information (correspondences), we added the modeling artifact type CorrB2CT as another source to B2CT. CorrB2CT already contains the information whether a block and a composition type correspond. The instantiation check uses this additional source to check whether the related block and composition type are related via a CorrB2CT. Thereby, we translate proprietary traceability information of the TGG model transformation into the concept of virtual modules.

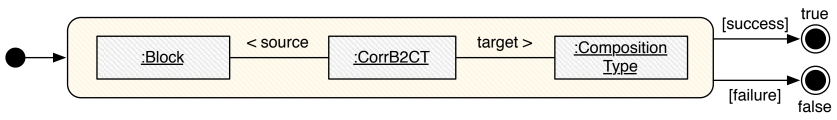


Fig. 7. Instantiation check associated to B2CT

The instantiation check of B2CT is defined as shown in Fig. 7. In our application example, we use Story Diagrams (cf. [7]) for the specification of the instantiation check. The instantiation check evaluates whether a given modeling artifact :Block and a given modeling artifact :CompositionType are connected by a given modeling artifact :CorrB2CT. If this holds, the application of the instantiation check returns true, else it returns false.

By specifying B2CT, we are now able to also explicitly define a context composition between B2CT and R2LTC and, thus, we have an indirect context composition between S2A and R2LTC. The context composition between B2CT and R2LTC is illustrated in Fig. 8. Such a context composition is explicitly defined

[14] The module is called B2CT because it connect a block and a composition type that correspond to each other.

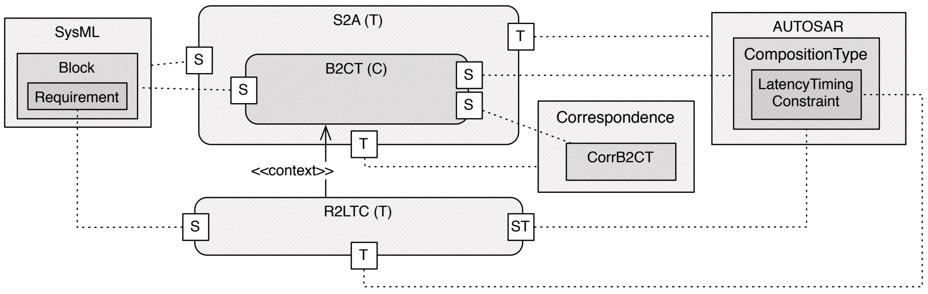


Fig. 8. R2LTC composed via context into B2CT

by a reference between modules labeled with <<context>>. A module can have multiple context references and, thus, can exist in multiple context compositions (but not at the same time which means XOR-semantics).

Summarized, the model transformations that are related to modules that use B2CT as context are completely decoupled from the model transformation related to the module S2A. We only couple their module specifications via context dependencies.

3.3 Instantiation of Modules

Up until now, we have shown how to specify modules that appropriately represent model transformations as black-boxes and how to define context composition between model transformations through modules. Now, we show how a model transformation user is able to instantiate modules and, therefore, model transformations for eventual application by employing traceability links.

The instantiation of a module is always performed into a specific application context (i.e., between specific models or model elements). This application context of a traceability link has to conform to the modeling artifact types of the instantiated module, which means that the modeling artifacts are instances of the modeling artifact types that are related to the parameter types of the module.

Fig. 9 shows a traceability link that is an instantiation of the module S2A and, therefore, an instantiation of a SysML to AUTOSAR model transformation. fuelsys:SysML is the SysML model that provides the fuel system controller of our

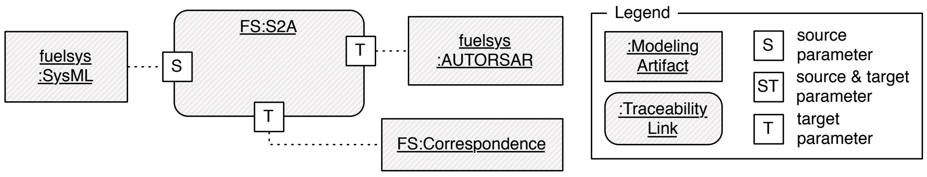


Fig. 9. Example of the concrete syntax of traceability links

application example shown in Fig. 3, FS:Correspondence is the Correspondence model that is created by the TGG model transformation, and fuelsys:AUTOSAR is the AUTOSAR model that is created by the TGG model transformation based on the given SysML model.

3.4 Specification of Data-Flow Compositions

Fig. 10 shows a sufficient configuration of the tool chain from our application example that is illustrated in Fig. 2. FS:S2A and FS:E2SD are composed by a data-flow because FS:S2A is producing fuelsys:AUTOSAR which is subsequently consumed by FS:E2SD. We have chosen the data-flow composition between FS:S2A and FS:E2SD because E2SD does not extend S2A but rather transforms a AUTOSAR model as a whole into a SystemDesk conform representation.

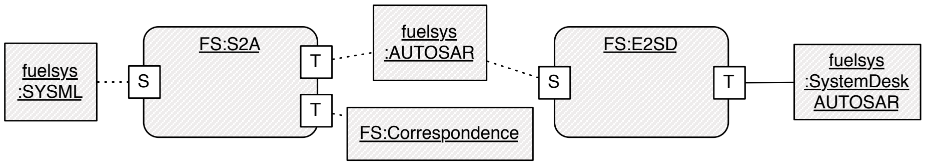


Fig. 10. Instantiation of a data-flow composition representing the tool chain

4 Execution of True Black-Box Model Transformations

Based on the instantiation shown in Fig. 10, we show how MoTCoF automatically executes the presented tool chain, transforming the SysML model of the fuel system controller – including textual timing requirements – into a SystemDesk conforming AUTOSAR model. Our implementation distinguishes between the execution of instantiated model transformations that are in a data-flow composition and model transformations that are in a context composition. Thereby, context-composed model transformations are automatically instantiated during execution. The execution sequence always follows the same schema: firstly, an already instantiated model transformation is applied and, secondly, its context compositions are completely instantiated and applied before the next already instantiated model transformation is applied. Fig. 11 illustrates the execution sequence based on our application example.

In step one, the model transformation, which is related to the module S2A, is applied into the application context of FS:S2A. This partly transforms the given fuelsys:SysML model into the related fuelsys:AUTOSAR model. In step two, the execution engine automatically creates (exports) instances of B2CT to all possible application contexts that exist within the context of FS:S2A (this are all combinations of a :Block, which are directly or indirectly contained by fuelsys:SysML, a :CompositionType, which is contained by fuelsys:AUTOSAR, and a :CorrB2CT, which is contained by FS:Correspondence). However, B2CT is only instantiated if the associated instantiation check, shown in Fig. 7, evaluates to true. Otherwise, the application context is withdrawn. This is necessary because B2CT should only exist between SysML blocks and AUTOSAR composition types that indeed correspond to each other. In our application example, three instances of B2CT are created in step two (one for each transformed SysML block generated by applying S2A).

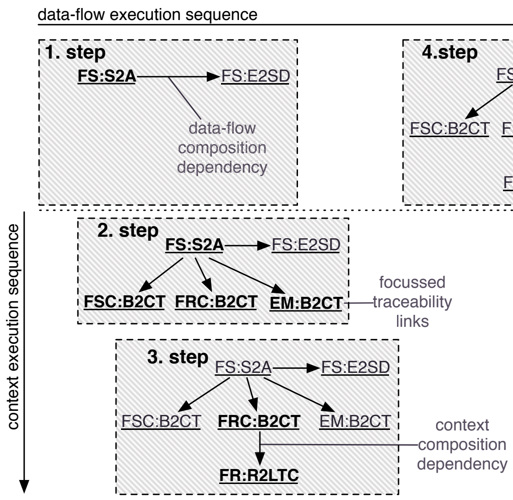


Fig. 11. Execution sequence of our application example

After all correct application contexts for B2CT were found and, thus, B2CT is instantiated completely, in step three each instance of B2CT (FSC:B2CT, FRC:B2CT and EM:B2CT) is employed as context for instantiating the module R2LTC because R2LTC is explicitly defined as composed via context into B2CT. As R2LTC has Requirement as source, a CompositionType as source & target and LatencyTimingConstraint as target, it is sufficient for an application context to consist out of a :Requirement and a :CompositionType only. Targets are automatically created or overwritten when instantiating modules. Since R2LTC is not associated to an instantiation check, the execution engine automatically creates instances of R2LTC for any application context that is found in the context of a :B2CT. In our application example, there is only one requirement specified (FR:Requirement). Thus, only one R2LTC is instantiated (FR:R2LTC). Because R2LTC is associated to a model transformation, the model transformation is subsequently applied into the application context of FR:R2LTC. Consequently, FR:LatencyTimingConstrained is created and added to FuelRateController:CompositionType so that it appropriately represents FR:Requirement.

Figure 12 shows FS:S2A after executing step one, two and three. It can be seen that FR:LatencyTimingConstraint is created within the right fuel rate controller FuelRateController:CompositionType because the FR:R2LTC was executed only in the context of a correct B2CT (which contains the requirements that is transformed). Thus, the associated transformation does not need to be costly to reproduce the information where to transform the SysML requirement into. Because no more context compositions are available, in a fourth step the already configured FS:E2SD is applied, which transforms the fuelsys:AUTOSAR model into a SystemDesk conforming representation fuelsys:SystemDeskAUTOSAR.

The execution engine is made of two execution algorithms: a data-flow execution algorithm and a context execution algorithm. The data-flow execution algorithm is shown in Listing 1.1. It is responsible for applying model transformations that are instantiated in data-flow compositions in a correct sequence. We assume that the instantiation of data-flows do not result in cyclic dependencies at all. If so, the data-flow execution algorithm cannot be applied correctly. Thus, before executing the data-flow composition a simple algorithm can detect cyclic dependencies and inform the model transformation user to resolve the cycles.

The context execution algorithm is shown in Listing 1.2. It is responsible for instantiating and applying model transformations that are specified in context compositions in a correct sequence. The context execution algorithm is subordinate to the data-flow execution algorithm and, thus, is triggered by the latter one after each step.

4.1 Data-Flow Execution Algorithm

The data-flow execution algorithm automatically executes all traceability links in a correct sequence, which is defined by means of the data-flow composition.

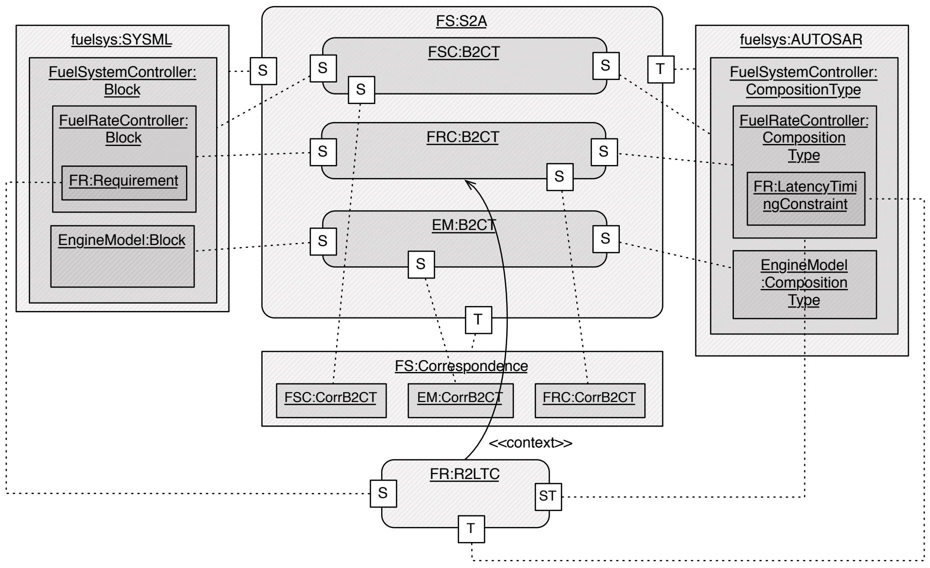


Fig. 12. Traceability links after executing steps one, two and three

It takes a set of traceability link L as input that should be executed, which are instances of modules that are not in a context composition (does not depend on the context of another model transformation, e.g., FS:S2A and FS:E2SD as shown in Fig. 10). The correct sequence for executing traceability links is obtained by topologically sorting L, which sorts them concerning produced and consumed modeling artifacts.

```
1 procedure $executeDataFlow(L) : void$ {
2     $L' := topologicallySortTraceabilityLinks(L)$;
3     forall ($l \in L'$) {
4         $module := getModuleOf(l)$;
5         $operation := getModelTransformation(module')$;
6         if ($operation \neq null$) $apply(l', operation)$;
7         $executeContext(l)$;
8     }
9 }
```

Listing 1.1. Data-flow execution algorithm

For each traceability link in the sorted set L', the associated model transformation is applied, if available, using the technology adapter interface (Line 5-6). Then, the context execution algorithm is triggered to also execute model transformations that are directly and indirectly composed via context into the module related to l (Line 7).

4.2 Context Execution Algorithm

The context execution algorithm automatically instantiates and executes all modules that are directly context-composed into the module that is related to the given traceability link l. Firstly, the algorithm triggers *instantiate Context* (see Listing 1.3) on l, which returns all correct instances of all modules directly context-composed into the given module (Line 3). Secondly, for each of the newly instantiated traceability links, the associated model transformation, if available, is applied using the technology adapter interface (Line 5-9). Thirdly, the algorithm is recursively triggered for each previously instantiated and executed traceability link in L' because the related module of each traceability link l' may be the source of context compositions, too. The recursion terminates because the context composition structure is a tree. Thus, a module with no further context compositions will be reached definitely.

The algorithm is wrapped into a fix-point (Line 2 and 4) because a module can depend the outcome of a direct or indirect neighbor module. The fix-point is reached as soon as no more traceability links can be instantiated within the context of l. A fix-point exists, if there are no model transformations that mutually create their own pre-conditions, which should usually hold.

4.3 Instantiation Algorithm

The instantiation algorithm instantiates all modules that are directly composed via context into the module related to l. Therefore, for each module that is

```
procedure $executeContext(l) : void$ {
    while (true) {
        $L' := instantiateContext(l)$;
        if ($L' == \emptyset$) break; //break if fix-point is reached
        forall ($l'$ in $L'$) {
            $module' := getModuleOf(l')$;
            $operation := getModelTransformation(module')$;
            if ($operation \neq null$) $apply(l', operation)$;
        }
        forall ($l'$ in $L'$) $executeContext(l')$;
    }
}
```

Listing 1.2. Context execution algorithm

```
1  procedure $instantiateContext(l) : L$ {
2      $L := \emptyset$;
3      $module := getModule(l)$;
4      forall ($module'$ that are exported or composed via context by
           $module$) {
5          $AC := findApplicationContexts(l, module')$;
6          forall ($applicationContext \in AC$) {
7              if ($\nexists l' \in L : getApplicationContext(l') = applicationContext \wedge$
                   $getModule(l') = module'$) {
8                  $l' := createInstance(applicationContext, module')$;
9                  $operation := getInstantiationCheck(module')$;
10                 if ($operation == null$ || $apply(l', operation)$) {
11                     $L := L \cup l'$;
12                     //add $l'$ as child to $l$
13                 }
14             }
15         }
16     }
17     return $L$;
18 }
```

Listing 1.3. Instantiation of modules in the context of a traceability link

directly composed via context, the algorithm first finds all possible application contexts (Line 5). An application context is a set of modeling artifacts with only modeling artifacts are considered that are directly related to l or that are directly or indirectly contained by modeling artifacts directly related to l. Then, for each context-composed module and each application context, an instance of that module is tried to be created into that context. However, creating an instance depends on the following conditions: no instance of the considered module has to exist in that application context (Line 7) and if the module is associated with an instantiation check, the instantiation check must evaluate to true (Line 10). The algorithm returns a set of correctly instantiated traceability links.

4.4 Technology Adapters

Technology adapters act as abstraction layer between model transformation technologies and MoTCoF. Any technology adapter has to implement an interface, such that MoTCoF can trigger the application of model transformations without reasoning about technical details and the technology behind. For each technology only one adapter has to be implemented. Currently, MoTCoF provides technology adapters for TGG, Java, Story Diagrams, ATL, EMF Compare[15]. This interface is also used to trigger the application of instantiation checks.

```
procedure $apply(l, operation) : boolean$ {
  boolean $success := false$;
  $applicationContext := getApplicationContext(l)$;
  //implementation that applies operation in applicationContext
      using the technology specific API
  ...
  return success;
}
```

Listing 1.4. Schematic implementation of a technology adapter interface

Listing 1.4 shows a schematic implementation of the technology adapter interface *apply* that has to be implemented. The parameters of the interface are a traceability link l and a model transformation/instantiation check *operation* that should be applied in the application context of l. Because we consider model transformations and instantiation checks as true black-boxes we do not expect to hand over the traceability link l but rather the application context of l, which are the concrete modeling artifacts processed by the model transformations or instantiation check. The result of *apply* is true if the application of *operation* in *applicationContext* was successful. Otherwise, it should return false.

5 Related Work

We have considered several approaches that are dedicated to the composition of primarily black-box model transformations. We compare all considered approaches by means of the following properties. Composition type declares what type of composition is applied, which are data-flow or context compositions, as explained in the introduction. True black-box is the ability to apply and compose model transformations implemented in different model transformation languages, so that an approach does not require to extend or modify model transformations for composition compatibility. Table 1 shows a direct comparison of MoTCoF with all considered related works.

5.1 Data-Flow Composition

In the following, approaches that support data-flow composition are introduced. In UniTI [5] a sophisticated approach to chain model transformations is shown. There, the authors provide a component-like language to define model transformations and their interconnections. Similarly, in [8] Oldevik composes black-box model transformations by means of UML Activity Diagrams and in [9] MWE is

[15] http://www.eclipse.org/emf/compare/

Table 1. Direct comparison of approaches

Approach	Composition Type		True Black-Box
	Data-Flow	Context	
UniTI [5]	√	×	√
J. Oldevik [8]	√	×	√
MWE [9]	√	×	√
MCC [10]	√	×	√
D. Wagelaar [11]	√	×	√
Aldazabal et al. [12]	√	×	√
Etien et al. [13]	√	×	∼
TraCo [14]	√	×	√
Cuadrado et al. [15]	×	√	×
Vanhooff et al. [16]	√	√	×
Epsilon [17]	√	√	×
QVT Relational [18]	×	√	×
MoTCoF	√	√	√

proposed to build chains of model transformations, which is part of the oAW[16] framework. In [10] a scripting language called MCC[17] is introduced. Based on this scripting language the authors show how to compose model transformations sequentially and in parallel. Both types of composition are data-flow compositions. The parallel composition can be applied if two transformations do not depend on the same models. In [11] Wagelaar defines model transformation chains by specifying a meta model that fits into a specific schema. From these kinds of meta models, Ant scripts are generated for further execution. In [12] Aldazabal et al. define the composition of model transformations by means of BPEL[18] models. Thus, the composition is basically a data-flow composition including additional logic by means of the BPEL language. Etien et al. define in [13] independent model transformations as model transformation chains. Due to the specific problem they solve, the composed transformations do not access the same models directly, but the models are somehow translated. Finally, in [14] the data-flow composition approach TraCo[19] is presented. There, model transformations that have overlapping input and output modeling artifacts are executed sequentially, while model transformations that do not share modeling artifacts are executed in parallel.

As mentioned above, all these approaches focus on data-flow composition, only. Further each of the approaches considers model transformations as true black-boxes, except Etien et al.'s approach [13], where this is not mentioned explicitly.

[16] oAW = openArchitectureWare; http://www.eclipse.org/workinggroups/oaw/
[17] MCC = MDA Control Center.
[18] BPEL = Business Process Execution Language; http://bpel.xml.org/
[19] TraCo = Transformation Composition.

5.2 Context Composition

Besides data-flow composition approaches, there are already approaches that support context composition. One of them is shown by Cuadrado et al. in [15]. In this approach potentially heterogeneous model transformations are grouped in so called phases. The composition of these phases is obtained in two different modes. First, independent model transformations are composed and second, model transformations that depend on the outcome of previously applied model transformations are composed, which the authors call refinement. They use implicitly created traceability information that can be queried by model transformation through a function they provide. Thus, their approach supports context composition. However, the authors assume that model transformations have to implement the provided traceability function to reason about the additional context provided by traceability. Since, this cannot be implemented with each transformation technology, Cuadrado et al.'s approach violates the black-box principle. Another context composition approach is presented by Vanhooff et al. in [16]. The authors use a global traceability graph, which contains traceability information, as context for the composition of model transformations. Nevertheless as in Cuadrado et al.'s approach [15], Vanhooff et al. expect that model transformations need to explicitly interpret traceability information and, thus, violate the black-box principle, too. They further do not show how to execute compositions. Finally, in [17], the Epsilon model management framework is presented. Epsilon allows black-box data-flow composition of Epsilon tasks, defined in E* languages[20] or additional languages like ATL[21]. The data-flow composition is implemented with Ant[22] scripts. In addition, Epsilon provides its own transformation language ETL. Within ETL, context compositions are possible. However, context compositions are not possible with other transformation languages as ETL and, as in the two approaches above, the traceability information of the context composition has to be interpreted by the transformation. Thus, Epsilon provides no black-box context composition.

These three approaches support context composition, but all of them violate the black-box principle, because they require that model transformations need to explicitly query the context information (traceability). In contrast to these approaches, MoTCoF does not pass the traceability information directly to the model transformation, but rather unpacks the information before. Thus, it provides the required context in the form of modeling artifacts only. Thus, our approach for data-flow and context composition is a true black-box approach.

To the best of our knowledge, there is currently no approach that supports context composition of true black-box model transformations. Nevertheless, QVT[23] Relational in combination with QVT Black-Box could be employed to provide a proper solution. If we assume to employ QVT as a language to only compose black-box model transformations, context composition could be realized by

[20] http://www.eclipse.org/gmt/epsilon/
[21] http://www.eclipse.org/atl/
[22] http://ant.apache.org/
[23] QVT = Query View Transformation; http://www.omg.org/spec/QVT/

means of *when* and *where* clauses, that could be used to directly invoke other black-box model transformations. The invocation does not pass traceability information directly, but rather parameters that are coming from the invoking model transformation, which acts as context.

However, beside the fact that currently no implementation of a combination of QVT Relational and QVT Black-Box exists, none of the clauses implement a composition semantic that is suitable for our transformation extension scenario. The implication of the where clause is that the success of the invoking model transformation (comparable to our extended transformation) relies on the success of the invoked model transformation (i.e., the extending transformation)(top-down). In our scenario, we expect that the extended model transformation can be applied successfully even if the invoked model transformation cannot be applied successfully. For example, consider the case that a SysML block does contain a not yet syntactically correct requirement. In this case, the extending model transformation would fail. However, this should imply that the extended model transformation fails. The when clause is principally the contrary (bottom-up) of the where clause. This implies that, e.g., applying R2LTC would invoke B2CT. However, an AUTOSAR model might contain a high amount of requirements. Thus, needing to apply R2LTC on each requirement manually makes the context composition useless.

6 Conclusions and Future Work

In this paper, we have introduced our dedicated model transformation composition framework MoTCoF that supports the data-flow and context composition of model transformations specified in different model transformation languages. The model transformations do not require the inclusion of any additional information that would be necessary for the composition. Thus, we can consider model transformations as true black-boxes. Therefore, we have defined a module language to represent model transformation in MoTCoF. Modules are suitable interfaces for model transformations, which enable us to contemplate model transformations as black-boxes. Nesting modules or connecting them by means of context references is a sufficient definition for context composition. The execution of model transformations first requires the manual instantiation of modules into concrete application contexts (between concrete modeling artifacts), which implies the instantiation of data-flow compositions. Second, for a set of instantiated modules, the execution engine automatically applies the represented model transformations in the required sequence determined by the data-flow composition. Furthermore, the execution engine automatically instantiates modules and applies the related model transformations into application contexts given by the previous application of model transformations. The execution engine further uses the context composition between modules to determine the necessary sequence for applying model transformations.

There are several points that we want to improve in the future. Currently, MoTCoF only supports the definition of data-flow composition between manually instantiated modules. To support the model transformation user, we think of instantiating modules automatically that are in a data-flow composition. We also suppose that the automatic instantiation of modules, which are in a data-flow composition, into the context of another module is beneficial for certain scenarios. To support model transformation designers, we believe that an automated validation of modules, which are defined in a context composition, is necessary. It is not obvious if the context composition of modules is applicable because it requires that related modeling artifacts exist in a transitive containment hierarchy. Yet, a major open challenge is to support not only the initial application of model transformations but also to reapply model transformations. For example, an incremental re-execution based on changes.

References

1. Mens, T., Van Gorp, P.: A taxonomy of model transformation. Electronic Notes in Theoretical Computer Science 152, 125–142 (2006)
2. Giese, H., Hildebrandt, S., Neumann, S.: Model Synchronization at Work: Keeping SysML and AUTOSAR Models Consistent. In: Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (eds.) Nagl Festschrift. LNCS, vol. 5765, pp. 555–579. Springer, Heidelberg (2010)
3. Giese, H., Wagner, R.: Incremental Model Synchronization with Triple Graph Grammars. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 543–557. Springer, Heidelberg (2006)
4. Yourdon, E., Constantine, L.L.: Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, 1st edn. Prentice-Hall, Inc., Upper Saddle River (1979)
5. Vanhooff, B., Ayed, D., Van Baelen, S., Joosen, W., Berbers, Y.: UniTI: A Unified Transformation Infrastructure. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 31–45. Springer, Heidelberg (2007)
6. AUTOSAR: Specification of Timing Extensions V1.1.0 R4.0 Rev 2 (2010), http://www.autosar.org/download/R4.0/AUTOSAR_TPS_TimingExtensions.pdf
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. Oldevik, J.: Transformation Composition Modelling Framework. In: Kutvonen, L., Alonistioti, N. (eds.) DAIS 2005. LNCS, vol. 3543, pp. 108–114. Springer, Heidelberg (2005)
9. openArchitectureWare: The modeling workflow engine (2011), http://www.eclipse.org/modeling/emft/?project=mwe
10. Kleppe, A.: MCC: A Model Transformation Environment. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 173–187. Springer, Heidelberg (2006)
11. Wagelaar, D.: Blackbox composition of model transformations using domain-specific modelling languages. In: Kleppe, A. (ed.) First European Workshop on Composition of Model Transformations, CMT 2006, pp. 15–19. Centre for Telematics and Information Technology, University of Twente, Enschede (2006)

12. Aldazabal, A., Baily, T., Nanclares, F., Sadovykh, A., Hein, C., Ritter, T.: Automated model driven development processes. In: Proc. of the ECMDA Workshop on Model Driven Tool and Process Integration. Fraunhofer IRB Verlag, Stuttgart (2008)
13. Etien, A., Muller, A., Legrand, T., Blanc, X.: Combining independent model transformations. In: Proceedings of the 2010 ACM Symposium on Applied Computing, SAC 2010, pp. 2237–2243. ACM, New York (2010)
14. Heidenreich, F., Kopcsek, J., Aßmann, U.: Safe Composition of Transformations. In: Tratt, L., Gogolla, M. (eds.) ICMT 2010. LNCS, vol. 6142, pp. 108–122. Springer, Heidelberg (2010)
15. Cuadrado, J., Molina, J.: Modularization of model transformations through a phasing mechanism. Software and Systems Modeling 8(3), 325–345 (2009)
16. Vanhooff, B., Van Baelen, S., Joosen, W., Berbers, Y.: Traceability as Input for Model Transformations. In: Proc. of 3rd Workshop on Traceability (ECMDA-TW), June 11-15, pp. 37–46. SINTEF, Haifa (2007)
17. Kolovos, D.S., Paige, R., Polack, F.: A Framework for Composing Modular and Interoperable Model Management Tasks. In: MDTPI Workshop, EC-MDA, Berlin, Germany (June 2008)
18. Object Management Group: MOF 2.0 QVT 1.0 Specification (2008)

An Algorithm for Layout Preservation in Refactoring Transformations

Maartje de Jonge and Eelco Visser

Dept. of Software Technology, Delft University of Technology, The Netherlands
m.dejonge@tudelft.nl, visser@acm.org

Abstract. Transformations and semantic analysis for source-to-source transformations such as refactorings are most effectively implemented using an abstract representation of the source code. An intrinsic limitation of transformation techniques based on abstract syntax trees is the loss of layout, i.e. comments and whitespace. This is especially relevant in the context of refactorings, which produce source code for human consumption. In this paper, we present an algorithm for fully automatic source code reconstruction for source-to-source transformations. The algorithm preserves the layout and comments of the unaffected parts and reconstructs the indentation of the affected parts, using a set of clearly defined heuristic rules to handle comments.

1 Introduction

The successful development of new languages is currently hindered by the high cost of tool building. Developers are accustomed to the integrated development environments (IDEs) that exist for general purpose languages, and expect the same services for new languages. For the development of Domain Specific Languages (DSLs) this requirement is a particular problem, since these languages are often developed with fewer resources than general purpose languages. Language workbenches aim at reducing that effort by facilitating efficient development of IDE support for software languages [10]. The Spoofax language workbench [12] generates a complete implementation of an editor plugin with common syntactic services based on the syntax definition of a language in SDF [23]. Services that require semantic analysis and/or transformation are implemented in the Stratego transformation language [3]. We are extending Spoofax with a framework for the implementation of refactorings.

Refactorings are transformations applied to the source code of a program. Source code has a formal *linguistic structure* [6] defined by the programming language in which it is written, which includes identifiers, keywords, and lexical tokens. Whitespace and comments form the *documentary structure* [6] of the program that is not formally part of the linguistic structure, but determines the visual appearance of the code, which is essential for readability. A fundamental problem for refactoring tools is the informal connection between linguistic and documentary structure.

Refactorings transform the formal structure of a program and are specified on the abstract syntax tree (AST) representation of the source code, also used in the compiler for the language. Compilers translate source code from a high-level programming language to a lower level language (e.g., assembly language or machine code), which is intended for consumption by machines. In the context of compilation, the layout of the output is irrelevant. Thus, compiler architectures typically abstract over layout. Comments and whitespace are discarded during parsing and are not stored in the AST.

In the case of refactoring, the result of the transformation is intended for human consumption. Contrary to computers, humans consider comments and layout important for readability. Comments explain the purpose of code fragments in natural language, while indentation visualizes the hierarchical structure of the program. Extra whitespace helps to clarify the connections between code blocks. A refactoring tool that loses all comments and changes the original appearance of the source code completely, is not useful in practice.

The loss of comments and layout is an intrinsic problem of AST-based transformation techniques when they are applied to refactorings. To address the concern of layout preservation, these techniques use layout-sensitive pretty-printing to construct the textual representation [13,14,16,18,20]. Layout is stored in the AST, either in the form of special layout nodes or in the form of tree annotations. After the transformation, the new source code is reconstructed entirely by unparsing (or pretty-printing) of the transformed AST. This approach is promising because it uses language independent techniques. However, preservation of layout is still problematic. Known limitations are imperfections in the whitespace surrounding the affected parts (indentation and inter-token layout), and the handling of comments, which may end up in the wrong locations. The cause of these limitations lies in the orthogonality of the linguistic and documentary structure; projecting documentary structure onto linguistic structure loses crucial information (Van De Vanter [6]).

In this paper, we address the limitations of existing approaches to layout preservation with an approach based on automated text patching. A text patch is an incremental modification of the original text, which can consist of a deletion, insertion or replacement of a text fragment at a given location. The patches are computed automatically by comparing the terms in the transformed tree, with their original term in the tree before the transformation. The changes in the abstract terms are translated to text patches, based on origin tracking information, which relates transformed terms to original terms, and original terms to text positions [22]. A layout adjustment strategy corrects the whitespace at the beginning and end of the changed parts, and migrates comments so that they remain associated with the linguistic structures to which they refer. The layout adjustment strategy uses explicit, separately specified layout handling rules that are language independent. Automated text patching offers more flexibility regarding layout handling compared to the pretty-print approach. At the same time, the layout handling is language generic and fully automatic, allowing the refactoring developer to abstract from layout-specific issues.

The paper provides the following contributions:

– A formal analysis of the layout preservation problem, including correctness and preservation proofs for the reconstruction algorithm;
– A set of clearly defined heuristic rules to determine the connection of layout with the linguistic structure;

– An algorithm that reconstructs the source code when the underlying AST is changed. The algorithm maximally preserves the whitespace and comments of the program text.

We start with a formalization of the problem of layout preservation. Origin tracking is introduced in Section 3. Section 4 explains the basic reconstruction algorithm, refined with layout adjustment and comment heuristics in Section 5. Finally, in Section 6 we report on experimental results.

2 Layout Preservation in Refactoring

Refactorings are behavior-preserving source-to-source transformations with the objective to 'improve the design of existing code' [9]. Although it is possible to refactor manually, tool support reduces evolution costs by automating error-prone and tedious tasks. Refactoring tools automatically apply modifications to the source code, attempting to preserve the original layout, which is not trivial to accomplish.

2.1 Example

We discuss the problems related to layout preservation using an example in WebDSL, a domain specific language for web applications [24]. Extract-entity is a refactoring implemented for WebDSL, Figure 1 shows the textual transformation. The required source code modifications are non trivial. A new entity (`Account`) is created from the selected properties, and inserted after the `User` entity. The selected properties are replaced by a new property that refers to the extracted entity. Comments remain attached to the code structures to which they refer. Thus, the comments in the selected region are moved jointly with the selected properties. Furthermore, the comment `/*Blog Info*/` still precedes the `Blog` entity. The layout of the affected parts is adjusted to conform to the style used in the rest of the file. In particular, indentation and a separating empty line are added to the inserted entity fragment.

```
entity User {
  name : String
  //account info
  pwd : String //6ch
  user : String
  expire : Date
}

/*Blog info*/
entity Blog { ... }
```

```
entity User {
  name : String
  account : Account
  expire : Date
}

entity Account {
  //account info
  pwd : String //6ch
  user : String
}

/*Blog info*/
entity Blog { ... }
```

Fig. 1. Textual transformation

Figure 2 displays the abstract syntax of the program fragment. Abstract syntax trees represent the formal structure of the program, abstracting from comments and layout. Automatic refactorings are typically defined on abstract syntax trees; the structural representation of the program is necessary to reliably perform the analyses and transformations needed for correct application. Moreover, abstracting from the arbitrary layout of the source code simplifies the specification of the refactoring.

```
[Entity(
  "User"
, [ Prop("name", "String")
  , Prop("pwd", "String")
  , Prop("user", "String")
  , Prop("expire", "Date")])
,Entity("Blog", [...])]
```

Fig. 2. Abstract syntax

2.2 Problem Analysis

The refactoring transformation applied to the AST results in a modified abstract syntax tree. The AST modifications must be propagated to the concrete source text in order to restore the consistency between the concrete and abstract representation. Figure 3 illustrates the idea. S and T denote the concrete and the abstract representation of the program, the PARSE function maps the concrete representation into the abstract representation, while TRANSF applies the transformation to the abstract syntax tree. To construct the textual representation of the transformed AST, an UNPARSE function must be implemented that maps abstract terms to strings.

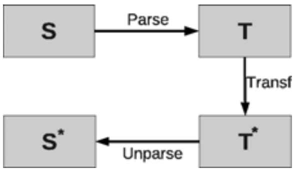


Fig. 3. Unparsing

The PARSE function is surjective, so for each well-formed abstract syntax term t, there exists at least one string that forms a textual representation of t. An UNPARSE function can be defined that constructs such a string [21]. The PARSE function is not injective; strings with the same linguistic structure but different layout are mapped to the same abstract structure, that is $\exists s : \textsc{unparse}(\textsc{parse}(s)) \neq s$. It follows that layout preservation can not be achieved by a function that only takes the abstract syntax as input, without having access to the original source text.

In the context of refactoring, it is required that the layout of the original text is preserved. A text reconstruction function that maps the abstract syntax tree to a concrete representation must take the original text into account to preserve the layout (Figure 4). We define two criteria for text reconstruction:

Correctness. $\textsc{Parse}(\textsc{ConstrText}(\textsc{transf}(\textsc{parse}(s)))) = \textsc{transf}(\textsc{parse}(s))$
Preservation. $\textsc{ConstrText}(\textsc{parse}(s)) = s$

The correctness criterion states that text reconstruction followed by parsing is the identity function on the AST after transformation. The preservation criterion states that parsing followed by text reconstruction returns the original source text. Preservation as defined above only covers the identity transformation. Section 4 gives a more precise criterion that defines preservation in the context of (non-trivial) transformations.

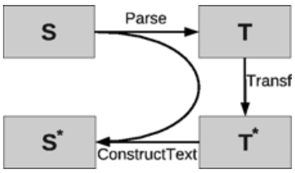


Fig. 4. Text reconstruction

The layout preservation problem falls in the wider category of view update problems. Foster et al. [8] define a semantic framework for the view update problem in the context of tree structured data. They introduce lenses, which are bi-directional tree transformations. In one direction (GET), lenses map a concrete tree into a simplified abstract tree, in the other direction (PUTBACK), they map a modified abstract view, together with the original concrete tree to a correspondingly modified concrete tree. A lens is well-behaved if and only if the GET and PUTBACK functions obey the following laws: $\text{GET}(\text{PUTBACK}(t, s)) = t$ and $\text{PUTBACK}(\text{GET}(s), s) = s$. These laws resemble our correctness and preservation criterion. Indeed, the bi-directional transformation PARSE, CONSTRUCTTEXT forms a well-behaved lens.

3 Origin Tracking

Text reconstruction implements an unparsing strategy by applying patches to the original source code. The technique requires a mechanism to relate nodes in the transformed tree to fragments in the source code. This section describes an infrastructure for preserving origin information. Figure 5 illustrates the internal representation of the source code. The program structure is represented by an abstract syntax tree (AST). Each node in the AST keeps a reference to its leftmost and rightmost token in the token stream, which in turn keep a reference to their start and end offset in the character stream. Epsilon productions are represented by a token for which the start- and end- offset are equal. This architecture makes it possible to locate AST-nodes in the source text and retrieve the corresponding text fragment. The layout structure surrounding the text fragment is accessible via the token stream, which contains layout and comment tokens.

When the AST is transformed during refactoring, location information is automatically preserved through *origin tracking* (Figure 6, dashed line arrows). Origin tracking is a general technique which relates subterms in the resulting tree back to their originating term in the original tree. The rewrite engine takes care of propagating origin information, such that nodes in the new tree point to the node from which they originate. Origin tracking is introduced by Van Deursen et al. in [22], and implemented in Spoofax [12]. We implemented a library for retrieving origin information. The library

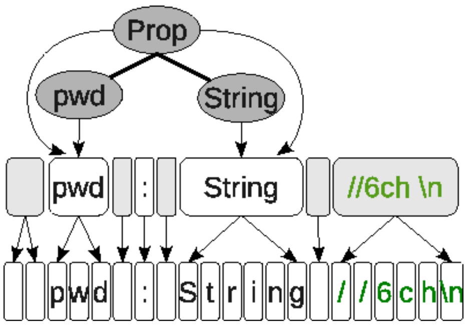


Fig. 5. Internal representation

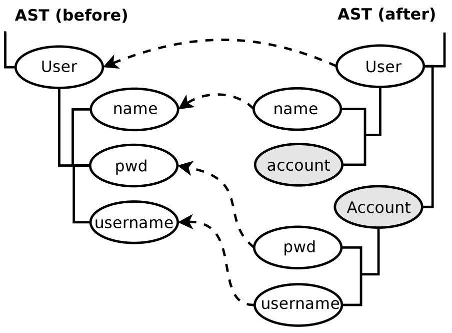


Fig. 6. Origin tracking

exposes the original node, its associated source code fragment, and details about surrounding layout such as indentation, separating whitespace and surrounding comments.

4 Layout Preservation for Transformed AST

In this section we describe the basic reconstruction algorithm and prove correctness and preservation.

4.1 Formalization

We introduce a formal notation for terms in concrete and abstract syntax, which stresses the correspondence between both representations. Given a grammar G, let S_G be the set of strings that represent a concrete syntax (sub)tree, and let T_G be the set of well-formed abstract syntax (sub)trees. We use the following notation for tree structures: $(t, [t_0...t_k]) \in T_G$ denotes a term $t \in T_G$ with subterms $[t_0...t_k] \in T_G$ (t_i denotes the subterm at the ith position). Equivalently, $(s, [s_0...s_k]) \in S_G$ means a string $s \in S_G$ with substrings $[s_0...s_k] \in S_G$, so that each s_i represents an abstract term $t_i \in T_G$, and s represents a term $(t, [t_0...t_k]) \in T_G$. Terms are characterized by their signature, consisting of the constructor name and the number of subterms and their sorts. When the constructor of a term is important, it is added in superscript ($(x^N, [x_0, ...x_k])$). Finally, for list terms the notation $[x_0, ...x_k]$ is used as short notation for $(x^{[]}, [x_0, ...x_k])$, leaving out the list constructor node.

We define the following operations on S_G and T_G, using the subscripts $_S$ and $_T$ to specify on which term representation the operation applies. Given a term $(x, [x_0...x_k])$ with subterm x_i, then $\mathrm{R}(x_i, x_{new})(x)$ replaces the subterm at position i with a new term x_{new} in term x. In case x is a list, additional operations are defined for deletion and insertion. $\mathrm{D}(x_i)(x)$ defines the deletion of the subterm at position i in x, while $\mathrm{IB}(x_i, x_{new})(x)$ and $\mathrm{IA}(x_i, x_{new})(x)$ define the insertion of x_{new} before (IB) or after (IA) the i^{th} element.

Assumption 1. *Let* PRS : $S_G \rightarrow T_G$ *the parse function that maps concrete terms onto their corresponding abstract terms.* PRS *is a homomorphism on tree structures.*

The text reconstruction algorithm translates the transformation in the abstract representation to the corresponding transformation in the concrete representation. This translation essentially exploits the homomorphic relationship between abstract and concrete terms. The applicability of the homomorphism assumption and techniques to overcome exceptional cases are discussed later in this section.

Lemma. *Let* PRS : $S_G \rightarrow T_G$ *the parse function, and assume* PRS : $S_G \rightarrow T_G$ *is a homomorphism on tree structures. Then the following equations hold:*

L 1. $\mathrm{PRS} \circ \mathrm{R}_S(s_i', s_i)(s) = \mathrm{R}_T(\mathrm{PRS}(s_i'), \mathrm{PRS}(s_i)) \circ \mathrm{PRS}(s)$

L 2. $\mathrm{PRS} \circ \mathrm{D}_S(s_i')(s) = \mathrm{D}_T(\mathrm{PRS}(s_i')) \circ \mathrm{PRS}(s)$

L 3. $\mathrm{PRS} \circ \mathrm{IB}_S(s_i', s_i)(s) = \mathrm{IB}_T(\mathrm{PRS}(s_i'), \mathrm{PRS}(s_i)) \circ \mathrm{PRS}(s)$

L 4. $\text{PRS} \circ \text{IA}_S(s'_i, s_i)(s) = \text{IA}_T(\text{PRS}(s'_i), \text{PRS}(s_i)) \circ \text{PRS}(s)$

Proof. This follows from the assumption that PRS is a homomorphism on tree structures. □

Definition. *Given the functions* PRS $: S_G \rightarrow T_G$*,* PP $: T_G \rightarrow S_G$*,* ORTRM $: T_G \rightarrow T_G$*,* ORTXT $: T_G \rightarrow S_G$*, with* PP *a pretty-print function and* ORTRM *and* ORTXT *functions that return the origin term respectively the origin source fragment of a term. The following properties hold:*

D 1. $\text{ORTRM}(\text{PRS}(s)) = \text{PRS}(s)$

D 2. $\text{ORTXT}(\text{PRS}(s)) = s$

D 3. $\text{PRS}(\text{ORTXT}(\text{ORTRM}(t))) = \text{ORTRM}(t)$

D 4. $\text{PRS}(\text{PP}(t)) = t$

D 5. $\text{PP}(s) = s\ for\ all\ string\ terms\ s$

4.2 Algorithm

We define an algorithm that reconstructs the source code after the refactoring transformation (Figure 8). CONSTRUCTTEXT$(node)$ takes an abstract syntax term as input and constructs a string representation for this term. Three cases are distinguished; reconstruction for nodes (l. 1-5), reconstruction for lists (l. 6-11), and pretty printing in case the origin term is missing, i.e. when a term is newly created in the transformation (l. 12-14). We discuss those cases.

If an origin term with the same signature exists (l. 2-3), the text fragment is reconstructed from the original text fragment, corrected for possible changes in the subterms. The function $\text{R}_S(t'_i, ti) : String \rightarrow String$ subsequently replaces the substrings that represent original subterms with substrings for the new subterms constructed by a recursive call to CONSTRUCTTEXT (l. 5). The (relative) offset is used to locate the text fragment associated to the original subterm ($\text{ORTXT}(t'_i)$), this detail is left out of the pseudo code.

Text reconstruction for list terms (line 6-11) implements the same idea, except that the changes in the subterms may include insertions and deletions. The textual modifications are calculated by a differencing function (DIFF) and subsequently applied to the original list fragment (line 11). The DIFF function matches elements of the new list with their origin term in the original list; the matched elements are returned as replacements (line 25), the unmatched elements of the old list form the deletions (lines 21, 29), while the insertions consist of the unmatched elements in the new list (lines 23, 30). It is crucial that the elements of the new list are correctly matched with related elements from the old list, since they automatically adopt the surrounding layout at the position of the old term, which may contain explanatory comments.

New terms are reconstructed by pretty-printing. To preserve the layout of subterms associated with an origin fragment, the pretty print function is applied after replacing the subterms with their textual representation, constructed recursively (line 14).

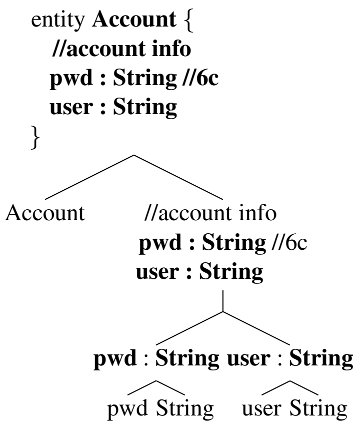


Fig. 7. Reconstruction example

The reconstruction algorithm implements a postorder traversal of the transformed abstract syntax tree, constructing the text fragment of the visited term from the text fragments of its subterms that were already constructed in the traversal. Figure 7 illustrates the reconstruction of the account entity. The substrings printed in bold are constructed by traversing the subterms, while the surrounding characters are either retrieved from the origin fragment, or constructed by pretty printing.

4.3 Correctness

We prove correctness of CONSTRUCTTEXT : $T_G \rightarrow S_G$ (Figure 8, abbreviated as CT), assuming that PARSE : $S_G \rightarrow T_G$ is a homomorphism on tree structures.

Theorem (Correctness). $\forall t \in T_G$ $\text{PARSE}(\text{CT}(t)) = t$

The proof is by induction on tree structures. We distinguish two cases for the leaf nodes, dependent on whether an origin term exists with the same signature.

Base case (a). Let $t = (t^N, [])$ a leaf node with origin term $(t^N, [])$.
$\text{PRS}(\text{CT}(t)) =^{line\ 5} \text{PRS}(\text{ORTXT}(\text{ORTRM}(t))) =^{D\ 3} \text{ORTRM}(t) = (t^N, [])$ □

Base case (b). Let $(t, [])$ a leaf node for which no origin term exists.
$\text{PRS}(\text{CT}(t)) =^{line 13-14} \text{PRS}(\text{PP}(t)) =^{D\ 4} t$ □

IH. $\text{PARSE}(\text{CT}(t_i)) = t_i$ *holds for all subterms* t_0 *to* t_k *of a term* $(t, [t_0, ..., t_k])$.

We now proof the induction step $\text{PARSE}(\text{CT}(t)) = t$.

Induction step (a). Assuming the induction hypothesis, we first prove a property of text modification operations as applied in lines 5, 11.

```
CONSTRUCTTEXT($term$) ▷ Abbreviated as CT
 1  if
 2      $(t^N, [t_0, ...t_k]) \leftarrow term$
 3      $(t'^N, [t'_0, ...t'_k]) \leftarrow$ ORTRM($term$)
 4  then  ▷ Term with origin info
 5      return $R_S$(ORTXT($t'_0$), CT($t_0$)) $\circ \cdots \circ$ $R_S$(ORTXT($t'_k$), CT($t_k$))
            $\circ$ ORTXT(ORTRM($term$))
 6  else if
 7      $[t_0, ...t_k] \leftarrow term$
 8      $[t'_0, ...t'_j] \leftarrow$ ORTRM($term$)
 9  then  ▷ List term
10      [MOD$_0$, ...MOD$_z$] $\leftarrow$ DIFF(ORTRM($term$), $term$)
11      return MOD$_0 \circ \cdots \circ$ MOD$_z$(ORTXT(ORTRM($term$)))
12  else  ▷ New constructed term
13      $(t^N, [t_0, ...t_k]) \leftarrow term$
14      return PP $\circ$ R$_T$($t_0$, CT($t_0$)) $\circ \cdots \circ$ R$_T$($t_k$, CT($t_k$))($t^N$)

DIFF($originLst, newLst$)
15  $diffs, unmatched \leftarrow []$
16  for each $el$ in $newLst$ do
17      if ORTRM($el$) $\in originLst$ then
18          $el' \leftarrow$ ORTRM($el$)
19          if PREFIX($el'$, $originLst$) $\neq []$ then
20              $deletedElems \leftarrow$ PREFIX($el'$, $originLst$)
21              $diffs \leftarrow D_S$(ORTXT($deletedElems$)) :: $diffs$
22          if $unmatched \neq []$ then
23              $diffs \leftarrow IB_S$(ORTXT($el'$), CT($unmatched$)) :: $diffs$
24              $unmatched \leftarrow []$
25          $diffs \leftarrow R_S$(ORTXT($el'$), CT($el$)) :: $diffs$
26          $originLst \leftarrow$ SUFFIX($el'$, $originLst$)
27      else
28          $unmatched \leftarrow unmatched$ ::: $[el]$
29  $diffs \leftarrow D_S$(ORTXT($originLst$)) :: $diffs$
30  $diffs \leftarrow IA_S$(ORTXT($originLst$), CT($unmatched$)) :: $diffs$
31  return REVERSE($diffs$)
```

Fig. 8. Pseudo code reconstruction algorithm

p 1. *Given a concrete syntax term* $(s, [...\textsc{OrTxt}(t'_i)...])$*. The following holds for modification operations* MOD $\in R,\ IB,\ IA,\ D$.

$\text{PRS} \circ \textsc{Mod}_S(\textsc{OrTxt}(t'_i), \text{CT}(t_i))(s) =^{L\ 1, L\ 2, L\ 3, L\ 4}$

$\textsc{Mod}_T(\text{PRS} \circ \textsc{OrTxt}(t'_i), \text{PRS} \circ \text{CT}(t_i)) \circ \text{PRS}(s) =^{D\ 3, IH}$

$\textsc{Mod}_T(t'_i, t_i) \circ \text{PRS}(s)$

We prove the induction step for constructor terms (t^N) below, the proof for list terms follows the same logic. Let $t = (t^N, [t_0...t_k])$ a term with origin term $t' = (t'^N, [t'_0...t'_k])$.

$\text{PRS} \circ \text{CT}(t) =^{line\ 5-6}$

$\textsc{prs} \circ \mathrm{R}_S(\textsc{OrTxt}(t'_0), \textsc{ct}(t_0))... \circ \mathrm{R}_S(\textsc{OrTxt}(t'_k), \textsc{ct}(t_k)) \circ \textsc{OrTxt}(t') =^p \boxed{1}$
$\mathrm{R}_T(t'_0, t_0) \circ ... \circ \mathrm{R}_T(t'_k, t_k) \circ \textsc{prs} \circ \textsc{OrTxt}(t') = D\,\boxed{3}$
$\mathrm{R}_T(t'_0, t_0) \circ ... \circ \mathrm{R}_T(t'_k, t_k)(t'^N, [t'_0...t'_k]) = (t^N, [t_0...t_k]) = t$ $\square$

Induction step (b). First, we prove a property for pretty printing.

p 2. $\textsc{prs} \circ \textsc{pp} \circ \mathrm{R}_T(t'_i, t_i)(t) =^D \boxed{4}$
$\mathrm{R}_T(t'_i, t_i)(t) =^D \boxed{4}$
$\mathrm{R}_T(\textsc{prs} \circ \textsc{pp}(t'_i), \textsc{prs} \circ \textsc{pp}(t_i)) \circ \textsc{prs} \circ \textsc{pp}(t) =^L \boxed{1}$
$\textsc{prs} \circ \mathrm{R}_S(\textsc{pp}(t'_i), \textsc{pp}(t_i)) \circ \textsc{pp}(t)$

Let $(t, [t_0...t_k])$ a node for which no origin term exists.
$\textsc{prs} \circ \textsc{ct}(t) =^{line\ 14}$
$\textsc{prs} \circ \textsc{pp} \circ \mathrm{R}_T(t_0, \mathrm{CT}(t_0)) \circ ... \circ \mathrm{R}_T(t_k, \mathrm{CT}(t_k))(t) =^p \boxed{2}$
$\textsc{prs} \circ \mathrm{R}_S(\textsc{pp}(t_0), \textsc{pp} \circ \mathrm{CT}(t_0)) \circ ... \circ \mathrm{R}_S(\textsc{pp}(t_k), \textsc{pp} \circ \mathrm{CT}(t_k)) \circ \textsc{pp}(t) =^{L\,\boxed{1}, D\,\boxed{5}}$
$\mathrm{R}_T(\textsc{prs}{\circ}\textsc{pp}(t_0), \textsc{prs}{\circ}\mathrm{CT}(t_0)){\circ}...{\circ}\mathrm{R}_T(\textsc{prs}{\circ}\textsc{pp}(t_k), \textsc{prs}{\circ}\mathrm{CT}(t_k)){\circ}\textsc{prs}{\circ}\textsc{pp}(t) =^{D\,\boxed{4}, IH}$
$\mathrm{R}_T(t_0, t_0) \circ ... \circ \mathrm{R}_T(t_k, t_k)(t) =^{D\,\boxed{4}} t$ $\square$

Applicability. The correctness proof depends on the assumption that parsing is a homomorphism on tree structures, we discuss two common exceptions. Tree structures in the concrete syntax representation can be ambiguous, in which case the parse result is determined by disambiguation rules. Syntactic ambiguities invalidate the homomorphic nature of the parse function. For instance, "2∗4+5", is parsed as $(t^{Plus}, [(t^{Mult}, [2, 4]), 5])$, while the alternate parse $((t^{Mult}, [2, (t^{Plus}, [4, 5])]))$ is rejected. Thus, bottom up text reconstruction fails to produce the correct code fragment for $(t^{Mult}, [2, (t^{Plus}, [4, 5])])$ in case $(t^{Plus}, [4, 5])$ is reconstructed as "$4 + 5$" instead of "$(4 + 5)$". To guarantee correctness, a preprocessor step is required that adds parentheses at the necessary places in the tree, where text reconstruction does not yield an expression between parentheses. The rules for parentheses insertion can be derived from the syntax definition [21]. This approach is taken in GPP [4], the generic pretty printer that is used in Spoofax. Another exception with respect to the homomorphism property concerns separation between list elements. When a list element is inserted (or deleted), it must be inserted (deleted) inclusive a possible separator, which is determined by the parent node. The separation is retrieved from the original source text in case the origin list has two or more elements, otherwise its looked up in the pretty-print table, based on te signature of the parent term.

4.4 Layout Preservation

Abstract syntax terms in general have multiple textual representations. These representations differ in the use of layout between the linguistic elements. In addition, small differences may occur in the linguistic elements; typically the use of braces is optional in some cases. We introduce the notion of formatting that covers these differences. Then we prove that the text reconstruction algorithm preserves formatting for terms that are not changed in the transformation, although they may have changes in their subterms.

Definition. *Given $(s, [s_0, ...s_k]) \in S_G$. The formatting of s is defined as the list consisting of the substring preceding s_0, the substrings that appear between the subterms $s_0, ...s_k$, plus the substring succeeding s_k*

Theorem (Maximal Layout Preservation). *Let $t \in T_G$ with origin term* $\textsc{OrTrm}(t) \in T_G$*. If t and* $\textsc{OrTrm}(t)$ *have the same signature, then* $\textsc{ct}(t)$ *and* $\textsc{OrTxt}(\textsc{OrTrm}(t))$ *have the same formatting.*

Proof. Let $(t^N, [t_0...t_k])$ a term with origin term $(t'^N, [t'_0...t'_k])$, then $\textsc{ct}(t) = \mathrm{R}_S(\textsc{OrTxt}(t'_0), \textsc{ct}(t_0)) \circ ... \circ \mathrm{R}_S(\textsc{OrTxt}(t'_k), \textsc{ct}(t_k)) \circ \textsc{OrTxt}(\textsc{OrTrm}(t))$. Since R_S only affects the substrings that represent the child nodes, the formatting of the parent string is left intact. For list terms: Let $t = [t_0...t_k]$ a list with origin term $\textsc{OrTrm}(t) = [t_0...t_l]$, then $\textsc{ct}(t) = \textsc{Mod}_{t'_0} \circ ... \circ \textsc{Mod}_{t'_l} \circ \textsc{OrTxt}(\textsc{OrTrm}(t))$ $\textsc{Mod}_{t'_i} \in \{\mathrm{R}_S, \mathrm{D}_S, \mathrm{IB}_S, \mathrm{IA}_S\}$. By definition, the modification functions affect the substrings representing the child nodes, or insert a new substring. In both cases the formatting of the parent string is preserved. □

5 Whitespace Adjustment and Comment Migration

The algorithm of Figure 8 preserves the layout of the unaffected regions, but fails to manage spacing and comments at the frontier between the changed parts and the unchanged parts. Figure 9 shows the result of applying the algorithm to the refactoring described in section 2 (Figure 1). Comments end up at the wrong location (`//account info`, `/*Blog info*/`), the whitespace separation around the `account` property and `Account` entity is not in accordance with the separation in the original text, and the indentation of the `Account` entity is disorderly.

The algorithm in Figure 8 translates AST-changes to modifications on code structures, but ignores the layout that surrounds these structures. To overcome this shortcoming, we refine the implementation of the algorithm so that whitespace and

```
entity User {
  name : String
  //account info

  account : Account expire : Date
}

/*Blog info*/
entity Account {
password : String //6 chars
  username : String
}entity Blog { ... }
```

Fig. 9. Layout deviation

```
IBADJUSTED($t_{old}, t_{new}$)
1  $text \leftarrow \textsc{ct}(t_{new})$
2  $text \leftarrow \textsc{RemoveIndent}(text)$
3  $text \leftarrow \textsc{AddIndent}($
       $text,$
       $\textsc{OriginIndent}(t_{old}))$
4  $text \leftarrow \textsc{ConcatStrings}([$
       $text,$
       $\textsc{OrSeparation}(t_{old})\ ])$
5  $offset \leftarrow \textsc{OffsetWithLO}(t_{old})$
6  return $\mathrm{IB}_S(offset, text)$
```

Fig. 10. Layout adjustment function

```
/**
 * Processes income data and displays statistics #1
 */
public static void displayStatistics(Scanner input) {
   //Initialize variables #2a
   int    count = 0;   // Number of values #3a
   double total = 0;   // Sum of all incomes #3b

   //Process input values until EOF #2b
   System.out.println("Enter income values");
   while (input.hasNextDouble()) {
      double income = input.nextDouble();
      //System.out.println("processing: " + income); #4
      if(income>=0){
         count++;             // Keep track of count
         total += income;     // and total income #5
      }
   }

   //Display statistics #2c
   double average = calcAverage(count, /*sum*/ total); #6
   System.out.println("Number of values = " + count);
   System.out.println("Average = "  + average);
}
```

Fig. 11. Comment styles

comments are migrated together with their associated code structures. This is implemented by using the layout-sensitive versions of the origin tracking functions to access origin fragments and locate textual changes. Language generic layout adjustment functions are implemented that correct the whitespace of reconstructed fragments, so that the spacing of the surrounding code is adopted. In particular, an inserted fragment is indented and separated according to the layout of the adjacent nodes. Figure 10 shows the layout adjustment steps for IB$_S$. First, the text is reconstructed with its associated comments. Then, the existing separation and (start)indentation is removed, leaving the nesting indentation intact. Subsequently, the start indentation at the insert location is retrieved from the adjacent term (t_{old}) and appended to all lines. Finally, separation is added (retrieved by inspecting the layout surrounding t_{old}) to separate the node from its successor.

5.1 Comment Heuristics

Comment migration requires a proper interpretation of how comments attach to the linguistic structure, which is problematic because of the informal nature of comments. The use of comments differs, depending on style conventions for a particular language and the personal preference of the programmer. Van De Vanter [6] gives a detailed analysis.

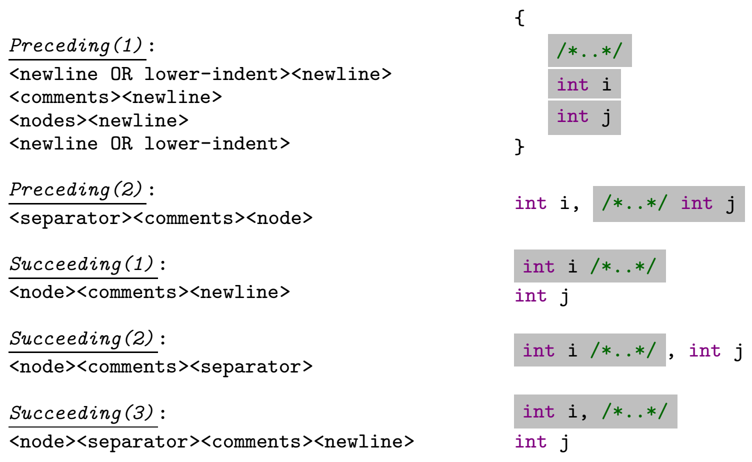
Figure 11 illustrates the use of comments with different style conventions used in combination. Fragment #1 is a block comment that explains the purpose of the accompanying method. The comment resides in front of its structural referent. This is also the case for the comments in #2a,b,c. However, these comments do not attach to a single structure element, but instead relate to a group of statements. The blank lines that surround these grouped statements are essential in understanding the scope of the comments. Contrary to the previous examples, the line comment in #3 points backwards to the preceding statement. #6 provides an example of a comment in the context of list elements separated by a comma. In this case, the location of the comma determines whether the comment points forward or backward. The commented-out `println` statement in #4 does not have a structural referent. It can best be seen as lying between the surrounding code elements. Finally, #5 illustrates a single comment that is spread over two lines. A human reader will recognize it as a single comment, although it is structurally split in two separate parts. In this case, the vertical alignment hints at the fact that both parts belong together.

Figure 11 makes clear why attaching comments to AST nodes is problematic. The connection of comments with AST-nodes only becomes clear when taking into account the full documentary structure, including newlines, indentation and separator tokens. Comments can point forward, as well as backward and, purely based on analysis of the tree structure, it is impossible to decide which one is the case. Even more problematic are #2 and #4; both comment lines lack an explicit referent in terms of a single AST node. The former refers to a sublist, while the latter falls between the surrounding nodes.

Text reconstruction allows for a more flexible approach towards the interpretation of comments. Instead of a fixed mapping between comments and AST nodes, heuristic rules are defined that interpret the documentary structure around the moved AST-part. Comment heuristics are defined as layout patterns using newlines, indentation, and separators as building blocks (Figure 12). If a pattern applies to a given node (or group of nodes), the node is considered as the structural referent of the comment(s) that take part in the pattern. The binding heuristics have the following effect on the textual transformation; if a node / group of nodes is (re)moved, all adjacent comments that bind to the node(s) are (re)moved as well. Adjacent comments that do not bind, stay at their original position in the source code. Comments that lie inside the region of the migrated node(s) automatically migrate jointly.

The patterns in Figure 12 handle the majority of comment styles correctly. The comment styles in Figure 11 are recognized by the patterns, with the exception of vertical alignment (#5), which is not detected. Preceding(1) binds #1 to the `displayStatistics` method, and #2a,b,c to the statement groups they refer to. #3 is interpreted by Succeeding(1). None of the patterns applies to #4, which indeed neither binds to the preceding nor to the succeeding node. The comment in #6 is associated with the succeeding node by application of Preceding(2). Finally, #5 is associated to its preceding statement, but not recognized as a single comment spread over two lines.

Heuristic rules will never handle all cases correctly; ultimately, it requires understanding of the natural language to decide the meaning of the comment and how it relates to the program structure. While our experience so far suggests that the heuristics are adequate, further experience with other languages, other refactorings, and other code bases is needed to determine whether these rules are sufficient.



```
Preceding(1):                                      {
<newline OR lower-indent><newline>                     /*..*/
<comments><newline>                                    int i
<nodes><newline>                                       int j
<newline OR lower-indent>                          }

Preceding(2):
<separator><comments><node>                        int i, /*..*/ int j

Succeeding(1):                                     int i /*..*/
<node><comments><newline>                          int j

Succeeding(2):
<node><comments><separator>                        int i /*..*/ , int j

Succeeding(3):                                     int i, /*..*/
<node><separator><comments><newline>               int j
```

Fig. 12. Comment patterns

6 Evaluation

We implemented the layout preservation algorithm in Spoofax [12], the sources of the library are available on-line [2]. We successfully applied the algorithm to renaming, extraction and inlining refactorings defined in WebDSL [24], MoBL [11] and Stratego [3]. In addition, we applied the algorithm to the Java refactorings mentioned in this section. For future work we will implement more refactorings and we will experiment with different languages and layout conventions.

Van De Vanter [6] points out the importance of the documentary structure for the comprehensibility and maintainability of source code. The paper gives a detailed analysis of the documentary structure consisting of indentation, line breaks, extra spaces and comments. The paper sketches the prerequisites for a better layout handling by transformation tools. We use the examples and requirements pointed out by Van De Vanter to provide a qualitative evaluation of our approach.

It is impossible for automatic tools to handle all layout correctly. After all, textual comments are written for human beings. Ultimately, comments can only be related to the code by understanding natural language. Therefore, instead of trying to prove that our tool handles layout correctly, we show that our approach meets practical standards for refactoring tools. We compare the layout handling of our technique with the refactoring support in Eclipse Java Development Tools (JDT), which is widely used in practice. We use a test set consisting of Java fragments with different layout styles. This set includes test cases for indentation and separating whitespace, as well as test cases for different comment styles, covering all comment styles discussed by Van De Vanter [6] and illustrated in Figure 12.

The results are summarized in Table 1; + means that the layout is accurately handled, -/+ indicates some minor issues, while - is used in case more serious defects were found. A minor issue is reported when the layout is acceptable but doen not precisely

Table 1. Layout Preservation Results

	Cat.	Description	E	CT
1	P1	Inline on method preceded by block comment	+	+
2		Inline on method preceded by a commented-out method	-	+
3		Move method preceded by multiple comments	+	+
4		Convert-to-field on the first statement of a group preceded by a comment	-	+
5		Convert-to-field on statement below commented-out line	-	+
6	P2	Change method signature	+	+
7	S1	Extract method, last stm ends with line comments	+	+
8		Extract method, preceding stm ends with line comments	+	+
9		Convert-to-field, decl with succeeding line comments	-	+
10	S2	Change method signature	+	+
11	S3	Change method signature	-/+	-/+
12	Inside	Extract method with comments in body	+	+
13		Inline method with comments in body	+	+
14	Selection	Extract method, preceding comments in selection	+	+
15		Extract method, preceding comments outside selection	+	+
16	Indent	Extract method, code style follows standards	+	+
17		Extract method, code style deviates from standards	-	-/+
18	Sep. ws	Extract method, code style follows standards	+	+
19		Extract method, code style deviates from standards	-/+	+
20	Format	Extract method, standard code style	+	+
21		Extract method, code style deviates from standard	-/+	-/+
22	V. align	Renaming so that v. alignment of "=" is spoiled	-	-
23		Renaming so that v. alignment of comments is spoiled	-	-

E : Eclipse Helios (3.6.2)
CT: Text Construction

follows the style used in the rest of the code, a serious defect is reported in case the layout is untidy or when comments are lost. The results show that our approach handles layout adequately in most cases. Different comment styles are supported (1-15), and the adjustment of whitespace gives acceptable results (16-19). 17, 19, and 23 show that variations in code style only led to some minor issues. For example in 17, the indent of the new inserted method correctly follows the indentation of the adjacent methods, but the indentation in the body follows the style defined in the pretty-print definition. Vertical alignment (22, 23) is not restored. A possible improvement is to restore vertical alignment in a separate phase, using a post processor.

Eclipse does not implement the same refined heuristic patterns as our technique, which explains the deviating results in 2, 4, and 5. In those three cases, the comments were incorrectly associated with the moved code structures and, consequently, did not remain at their original location. In all three cases the comment did not show up in the modified source code. In 9, the comment was not migrated to the new inserted field, although it was (correctly) associated to the selected variable declaration. The reason is that the relation between the inserted field and the deleted local variable is not set. In our implementation, the origin tracking mechanism keeps track of this relation. Eclipse uses editor settings to adjust the whitespace surrounding new inserted fragments, which works well under the condition that the file being edit adopts these settings.

We implemented a general solution for layout preservation with the objective to support the implementation of refactorings for new (domain specific) languages. Using our approach, the layout preservation is not a concern for the refactoring programmer but it is automatically provided by the reconstruction algorithm. The evaluation indicates that our generic approach produces results of comparable and in some cases even better quality then refactorings implemented in current IDEs.

7 Related Work

We implemented an algorithm for layout preservation in refactoring transformations. Instead of trying to construct the entire source code from the AST, the algorithm uses the original source text to construct the text for the transformed AST. Origin tracking is used to relate terms in the AST with their original code fragments, while internal changes are propagated and applied as text patches. As a result, the original layout is preserved for the unaffected parts of the program. The main challenge is the treatment of spacing and comments on the frontier between the changed and the unchanged code. Layout adjustment functions correct the whitespace of reconstructed fragments, so that the spacing of the surrounding code is adopted. Comments are migrated according to their intent. We define heuristic patterns for comment binding, that interpret the documentary structure near the node. The comment patterns are flexible in the sense that they do not assume a one-to-one relation between comments and AST nodes. The heuristic rules are language generic and cover the layout styles commonly seen in practice.

7.1 AST Approaches

Various attempts have been made to address the concern of appearance preservation by adding layout information to the AST. For a complete reconstruction, all characters that do not take part in the linguistic structure should be stored. This includes whitespace, comments and (redundant) parentheses. The modified source code is reconstructed from the transformed AST by layout-aware pretty printing [5].

Van den Brand and Vinju [20] use full parse trees in combination with rewrite rules in concrete syntax. The rewrite engine is adapted to deal with the extra layout branches, by using the assumption that any two layout nodes always match. The approach described in [18] also relies on extra layout branches. Instead of adapting the rewrite engine, the authors propose an automated migration of the transformation rules to take care of the layout branches. Layout annotations are used in [14] (Kort, Lämmel), while the RefactorErl tool [13] stores the layout information in a semantic graph.

All approaches based on extended ASTs succeed, to a certain extent, in preserving the original layout. In most approaches, layout is preserved for the unaffected parts, but the reconstruction of the affected parts has limitations. The implicit assumption is that the documentary structure can be mapped satisfactorily onto abstract syntax trees. However, the mapping of layout elements to AST nodes has intrinsic limitations. Attaching comments to preceding (or succeeding) AST nodes is a simplification that fails in cases when a comment is not associated with a single AST node, as is shown in examples provided by Van De Vanter [6]. Another shortcoming is related to indentation and whitespace separation at the beginning and end of changed parts. Migrating whitespace is not sufficient since the indentation at the new position may differ from the indentation at the old position, due to a different nesting level. Furthermore, newly constructed structures should be inserted with indentation and separating whitespace.

7.2 HaRe

HaRe [15,17] is a refactoring tool for Haskell that preserves layout. The program is internally represented by the Abstract Syntax Tree and the token stream, which are linked by source location information. Layout preservation is performed explicitly in the transformation steps, which process the token stream and the AST in parallel. After the transformation, the source code is extracted from the modified token stream.

Haskell programs can be written in layout-sensitive style for which the meaning of a syntax phrase may depend on its layout. For this reason, it is essential for the refactoring tool not to violate the layout rules when transforming the program. HaRe implements a layout adjustment algorithm to keep the layout correct. The algorithm ensures that the meaning of the code fragments is not changed, which does not necessarily mean that the code is as much as possible like the original one in appearance. HaRe uses heuristic rules to move/remove comments together with the associated program structures. These heuristics include rules for comments that precede a program structure and end-of-line comments that follow after a structure.

Similar to our approach, HaRe uses the token stream to apply layout analysis and to extract source code fragments. The main difference is that HaRe modifies the token stream during the transformation, while we reconstruct the source code afterwards, using origin-tracking to access the original source. The requirement to change the AST and token stream in parallel makes it harder to implement new transformations and requires an extension of the rewrite machinery specific for source-to-source transformations. We clearly separate layout handling from rewriting, which enables us to use the existing compiler infrastructure for refactoring transformations.

7.3 Eclipse

The Java Developer Toolkit (JDT) used in Eclipse offers an infrastructure for implementing refactorings [1]. Refactoring transformations are specified with replace, insert and remove operations on AST nodes, which are used afterwards to calculate the corresponding textual changes. Common to our approach, the replace, insert and remove operations on AST nodes are translated to textual modifications of the source code. However, instead of being restricted to the replace, delete and insert operations on AST nodes, we compute the primitive AST modifications by applying a tree differencing algorithm to the transformed abstract syntax tree. As a result, the transformation and text reconstruction are clearly separated. Thanks to this separation of concerns, we can specify refactorings in a specialized transformation language (Stratego).

7.4 Text Patching

The LS/2000 system [7,19] is a design-recovery and transformation system, implemented in TXL. LS/2000 is successfully applied for ”year 2000” remediation of legacy COBOL, PL/I, and RPG applications. The system implements an approach based on automated text patching. The differences between the original code and the transformed code are calculated with a standard differencing algorithm, operating on the token stream. The deviating text regions are merged back into the original text.

The token based differencing successfully captured changes that were relatively small. For millennium bug renovations, typical changes were the local insertion of a few lines of code. When the changes are large, or involve code movement, standard differencing algorithms do not work well [19]. We implemented a tree differencing algorithm that reconstructs moved code fragments by using origin tracking, furthermore, fragments with nested changes are reconstructed by recursion on subtrees.

7.5 Lenses

Foster et al. [8] implement a generic framework for synchronizing tree-structured data. Their approach to the view update problem is based on compoundable bi-directional transformations, called lenses. In the GET direction, the abstract view is created from the concrete view, projecting away some information; in the PUTBACK direction, the modified abstract view is mapped to a concrete representation, restoring the projected elements from the original concrete representation. The lens laws, which resemble our preservation and correctness criteria, impose some constraints on the behavior of the lens. Given a certain GET function, in general, many different PUTBACK functions can be defined. The real problem is to define a PUTBACK function that does what is required for a given situation. We define CONSTRUCTTEXT as a PUTBACK function for parsing, and prove that it fulfills the correctness and (maximal) layout preservation criteria.

Our approach is based on origin tracking as a mechanism to relate abstract terms with their corresponding concrete representation. Origin tracking makes it possible to locate moved subtrees in the original text. Furthermore, lists are compared using the origin relation to match corresponding elements. In contrast, lenses use the concrete representation as an input parameter to the PUTBACK function. As a consequence, details are lost about how subterms relate to text fragments. This seems especially problematic in case terms have nested changes, or when they are moved to another location in the tree. We defined heuristic rules for comment binding and layout adjustment functions to correct the spacing surrounding the changed parts. Layout adjustment and comment migration might be hard to express in the lenses framework. Foster et al. [8] mention the expressiveness of their approach as an open question. Layout preservation seems a challenging problem in this respect.

8 Conclusion

Refactorings are source-to-source transformations that help programmers to improve the structure of their code. With the popularity and ubiquity of IDEs for mainstream general purpose languages, software developers come to expect rich editor support including refactorings also for domain-specific software languages. Since the effort that can be spent on implementations of DSLs is often significantly smaller than the effort that is spent on (IDEs for) languages such as Java, this requires tool support for the high-level definition of refactorings for new (domain-specific) software languages.

An important requirement for the acceptability of refactorings for daily use is their faithful preservation of the layout of programs. Precisely this aspect, as trivial as it often seems compared to the actual refactoring transformation, has confounded meta-tool developers. The result is typically that the definitions of refactorings are contaminated with code for layout preservation. The lack of a generic solution for layout preservation has held back widespread development of refactoring tools for general purpose and domain-specific languages.

In this paper, we have presented an approach to layout preservation that separates layout preservation from the structural definition of refactorings, allowing the refactoring developer to concentrate on the structural transformation, leaving layout reconstruction to a generic library. The library computes text patches based on the differences between the old and the new abstract syntax tree, relying on origin tracking to identify the origins of subtrees. The approach applies layout conventions for indentation and vertical layout (blank lines) from the old code to newly created pieces of code; heuristic rules are defined for comment migration.

The separation of layout preservation from transformation enables the implementation of refactorings by the common meta-programmer. With this framework in place we expect to develop a further library of generic refactorings that will further simplify the development of refactorings for a wide range of software languages.

References

1. Eclipse documentation: Astrewrite Eclipse, JDT 3.6 (2010), `http://help.eclipse.org/helios/index.jsp?topic=/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/rewrite/ASTRewrite.html`
2. The Spoofax language workbench (2010), `http://strategoxt.org/Spoofax`
3. 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)
4. de Jonge, M.: A pretty-printer for every occasion. In: The International Symposium on Constructing Software Engineering Tools (CoSET 2000), pp. 68–77. University of Wollongong, Australia (2000)
5. de Jonge, M.: Pretty-printing for software reengineering. In: ICSM 2002: Proceedings of the International Conference on Software Maintenance (ICSM 2002), p. 550. IEEE Computer Society, Washington, DC (2002)
6. Van de Vanter, M.L.: Preserving the documentary structure of source code in language-based transformation tools. In: 1st IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001), November 10, pp. 133–143. IEEE Computer Society, Florence (2001)
7. Dean, T.R., Cordy, J.R., Schneider, K.A., Malton, A.J.: Using design recovery techniques to transform legacy systems. In: ICSM 2001: Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2001), p. 622. IEEE Computer Society, Washington, DC (2001)

8. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. 29(3) (2007)
9. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison Wesley (1999)
10. Fowler, M.: Language workbenches: The killer-app for domain specific languages? (2005)
11. Hemel, Z., Visser, E.: Programming the Mobile Web with Mobl. Technical Report 2011-01, Delft University of Technology (January 2011)
12. Kats, L.C.L., Visser, E.: The Spoofax language workbench. Rules for declarative specification of languages and ides. In: Rinard, M. (ed.) Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, Reno, NV, USA, October 17-21 (2010)
13. Kitlei, R., Lóvei, L., Nagy, T., Horváth, Z., Kozsik, T.: Layout preserving parser for refactoring in Erlang. Acta Electrotechnica et Informatica 9(3), 54–63 (2009)
14. Kort, J., Lämmel, R.: Parse-tree annotations meet re-engineering concerns. In: Proceedings of Third IEEE International Workshop on Source Code Analysis and Manipulation (September 2003)
15. Li, H., Thompson, S.: A comparative study of refactoring Haskell and Erlang programs. In: Penta, M.D., Moonen, L. (eds.) Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006), pp. 197–206. IEEE (September 2006)
16. Li, H., Thompson, S., Orosz, G., Toth, M.: Refactoring with Wrangler, updated: Data and process refactorings, and integration with Eclipse. In: Horvath, Z., Teoh, T. (eds.) Proceedings of the Seventh ACM SIGPLAN Erlang Workshop, p. 12. ACM Press (September 2008)
17. Li, H., Thompson, S., Reinke, C.: The Haskell Refactorer: HaRe, and its API. In: Boyland, J., Hedin, G. (eds.) Proceedings of the 5th Workshop on Language Descriptions, Tools and Applications (LDTA 2005) (April 2005)
18. Lohmann, W., Riedewald, G.: Towards automatical migration of transformation rules after grammar extension. In: CSMR 2003: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering, p. 30. IEEE Computer Society, Washington, DC (2003)
19. Malton, A., Schneider, K.A., Cordy, J.R., Dean, T.R., Cousineau, D., Reynolds, J.: Processing software source text in automated design recovery and transformation. In: Proc. International Workshop on Program Comprehension (IWPC 2001), pp. 127–134. IEEE Press (2001)
20. van den Brand, M., Vinju, J.: Rewriting with layout. In: Kirchner, C., Dershowitz, N. (eds.) Proceedings of RULE (2000)
21. van den Brand, M., Visser, E.: Generation of formatters for context-free languages. ACM Transactions on Software Engineering Methodology 5(1), 1–41 (1996)
22. van Deursen, A., Klint, P., Tip, F.: Origin tracking. J. Symb. Comput. 15(5-6), 523–545 (1993)
23. Visser, E.: Syntax Definition for Language Prototyping. PhD thesis, University of Amsterdam (September 1997)
24. Visser, E.: WebDSL: A Case Study in Domain-Specific Language Engineering. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007. LNCS, vol. 5235, pp. 291–373. Springer, Heidelberg (2008)

Cloning in DSLs: Experiments with OCL

Robert Tairas and Jordi Cabot

AtlanMod, INRIA & École des Mines de Nantes – France
{robert.tairas,jordi.cabot}@inria.fr

Abstract. Code cloning (i.e., similar code fragments) in general purpose languages has been a major focus of the research community. For domain specific languages (DSLs), cloning related to domain-specific graphical languages has also been considered. This paper focuses on domain-specific textual languages in an effort to evaluate cloning in these DSLs where instances of such DSLs allow for less code to express domain-specific features, but potentially more frequently used code constructs. We suggest potential application scenarios of using clone detection for the maintenance of DSL code. We introduce a clone detection mechanism using a Model Driven Engineering (MDE) based approach to evaluate the extent of cloning in an initial DSL (i.e., the Object Constraint Language (OCL)). The evaluation reveals the existence of cloning in OCL, which suggests the relevance and potential applications of clone detection and analysis in DSLs.

Keywords: Code clones, clone detection, domain-specific languages, ATL Transformation Language, Object Constraint Language.

1 Introduction

Code clones represent similar fragments of source code, where the similarity of the clones can vary, ranging from clones that are exactly the same syntactically to clones that are similar because they represent the same semantics. Research related to code clones has received much attention in the past decade. However, most efforts are geared toward clones found in source code written in general purpose language (GPLs). The evaluation of cloning in domain-specific languages (DSLs), specifically textual DSLs, has not received as much attention. These languages, which in many cases are used as modeling languages, are typically smaller in size compared to programs written in GPLs [5]. However, the language constructs in DSLs are more specific to a domain and hence could be used more often in the code, which could potentially introduce duplication resulting in clones. Our overall goal is to suggest scenarios for the use of clone detection in the maintenance of DSL code. Therefore, we are also interested in determining the relevance of cloning as it relates to DSLs to support the utility of clone detection-based tools for DSLs.

This paper focuses on the evaluation of cloning in artifacts containing code associated to the Object Constraint Language (OCL). We consider OCL artifacts as an initial step of a broader understanding of cloning in DSLs. We perform clone detection on these artifacts using a Model-Driven Engineering (MDE) approach in which models consisting of OCL expressions are transformed, thus producing a new model containing information about the clones of the expressions grouped together based on their similarities. The contribution of this paper is the study of cloning within expressions (of OCL) based on the observation of various OCL-related artifacts. The term *OCL expressions* and *expressions* will be used interchangeably throughout the remainder of this paper.

The paper is structured as follows: Section 2 outlines current state-of-the-art and related work. Section 3 offers scenarios where clone detection could be utilized in a DSL environment. Section 4 outlines the process used to detect clones and Section 5 details the detection process in OCL. Section 6 provides an evaluation of the detected clones in several artifacts related to OCL. Section 7 summarizes threats to validity of the approach described in this paper and Section 8 concludes the paper and summarizes future work.

2 Related Work

The research topic of code clones has received much attention in the past decade. The initial research activity was the introduction of automated techniques to detect clones in code. Early efforts include a technique proposed by Johnson [11] with more recent approaches given by Kim et al. [15]. Current state-of-the-art clone detection tools are able to detect clones in GPLs such as C and Java in reasonable speed for code bases that can reach up to several million lines of code, as can be seen in Table 1. A detailed listing of publications related to code clones is available at http://www.cis.uab.edu/tairasr/clones/literature. From the listing of publications at this web site, it can also be seen that research related to code clones is not solely focused on their detection, but also on the analysis of the detection results for various purposes including, for example, how clones evolve [16].

Table 1. Clone coverage percentage in programs

Artifact	LOC	% of Clones
Linux kernel [19]	4,365K	15%
Java Development Kit (JDK) 1.4.2 [10]	2,418K	8%
Process-Control System [1]	400K	12%

As seen in the previous paragraph, much of the research concerning code clones has focused on cloning in GPLs. In terms of DSLs, it should be noted that clone detection and analysis has been considered for domain-specific graphical languages [20][4]. However, textual-based DSLs has received less attention. The collection of domain-specific textual languages is an in-between case, as they can conform to a metamodel, but also are textual in nature. Because they conform to a metamodel, textual DSLs could use detection techniques from graphical DSLs. However, the detection techniques of graphical DSLs utilize graph-based algorithms to identify clones and hence place emphasis on the graphical layout of the models. Contextual information within the models is either not considered or abstracted thus removing information that could be included in the detection process. Our detection process attains as much information about the textual language as possible. Abstractions or parameterizations are only used to determine clones that consist of differing terminal values, but match syntactically.

Lämmel and Pek [17] observed cloning of P3P, a non-executable DSL related to Web-based privacy policies, as part of an overall analysis of the language. The DSL is an XML-based language and hence used for storing data, compared to the declarative type language of OCL, which we evaluated. The detection process for P3P is simpler as identifiers are non-existent, hence Type II clones (described Section 5.2) are not considered. The study of P3P and our study of OCL can serve as a repository of evaluations related to different families of DSLs. Li and Thompson [18] have considered clone detection on the Erlang/OTP functional language. As primarily a declarative language, OCL is somewhat related to functional languages. However, in [18] clone detection is performed with the main goal of finding refactoring opportunities to abstract certain parts of the source code. Our evaluation of OCL includes possible uses of OCL other than modularizing the code.

3 Application Scenarios

In this section, we describe potential application scenarios involving clone detection in DSLs as compared to the use in GPLs. Specifically, these scenarios relate to the maintenance of the DSL code that is associated with the clones.

3.1 In-place Maintenance

An alternative maintenance mechanism other than modularizing clones that has been proposed in GPLs is to keep track of where the clones are located and notify the user when one of the clones is edited to allow the user to consider editing the remaining clone(s). This mechanism is useful if the user does not want to modularize the clones right away, because they may have the property of being, for example, short-lived. Because the clones remain where they are, the maintenance can be considered to be *in-place*. Such a technique for Java has been proposed by Duala-Ekoko and Robillard [6]. The technique proposes *clone region descriptors* that will allow the tracking of associated clones even when the surrounding code has changed. This technique can not identify new clones that may be associated to any tracked clones, because clone detection is only executed at the first instance when the tool is initialized. This is due to the size of programs written in Java and most GPLs, which can result in each clone detection run to take a considerable amount of time.

Programs or code written in DSLs are mainly smaller compared to GPLs, because of their more expressive nature. This characteristic could potentially allow the execution of clone detection to be performed more frequently. If clone detection can be performed more often, then newer code fragments associated to any existing clone group can be identified more frequently. This will produce a more up-to-date and accurate tracking of clones and hence allow for a more complete in-place maintenance option. It should be noted that the option of in-place maintenance should not rule out any opportunities to modularize certain clone groups.

3.2 Pattern Detection and Suggestion

A separate scenario of utilizing clone detection is to determine commonly used patterns of language constructs. The results of clone detection can be used by an expert to determine commonly used patterns as an alternative to performing this step manually. The patterns can be stored in a repository and then used in several different ways that are described in the following paragraph.

The repository of common patterns can be used to assist users to complete a language construct when, for example, the user types in the first part of a construct. This repository could also be used to notify users when the code that was typed may be erroneous, similar to the mechanism proposed by Gruska et al. [9]. If what the user typed corresponds to a pattern in the repository, then the pattern can be displayed to the user to determine if user's code is correct. It should be noted that the common patterns represent multiple instances of use that were previously identified as clones. A robust clone detection technique can provide clones in varying levels of granularities compared to the function-level focus in [9]. A third scenario is the suggestion of a more optimal language construct to perform a specific functionality. If the clone detection process can identify as clones more varying degrees of language constructs that represent the same functionality, a DSL expert can determine the most optimal version among these clones. This optimal version can then be suggested to the user who in many cases does not have a programming background and hence is less experienced in determining optimal code options. Specifically related to OCL, properly parameterized common patterns can optionally be placed into a generic library such as that proposed by Chimiak-Opoka [3]. This library in turn can be used by multiple OCL artifacts.

4 Clone Detection Process

The scenarios outlined in Section 3 are meaningless if the relevance of cloning in DSLs is not strong. We devote the remainder of this paper to evaluate cloning in DSLs, with an initial consideration of OCL-related artifacts. Again, we consider OCL artifacts as an initial step of a broader understanding of cloning in DSLs.

Our overall process of finding clones in a DSL follows an MDE approach. We plan to apply the same detection process on other DSLs and consider MDE to allow for a more generic focus on the overall process, which can then be adapted to specific cases of DSLs. MDE proposes generating models of systems and performing transformations on these models to achieve a specific goal [22]. In our case, the models represent DSL code and the goal of the transformations of these models is to determine the duplication in the code. The top part of Figure 1 outlines the process that we propose to find clones in a DSL. A file or files containing the DSL code is parsed and a model or models representing the code are generated (Step 1). A model transformation step is performed that actually performs the detection of duplicate language constructs (Step 2). The resulting model or target model consists of information about the grouping of the detected clones. This information consists of the location of the clones in the original file of the DSL code (Step 3). This information is used to generate statistical data about the clones. It is also used to generate an HTML report of the clone groups, which is manually evaluated.

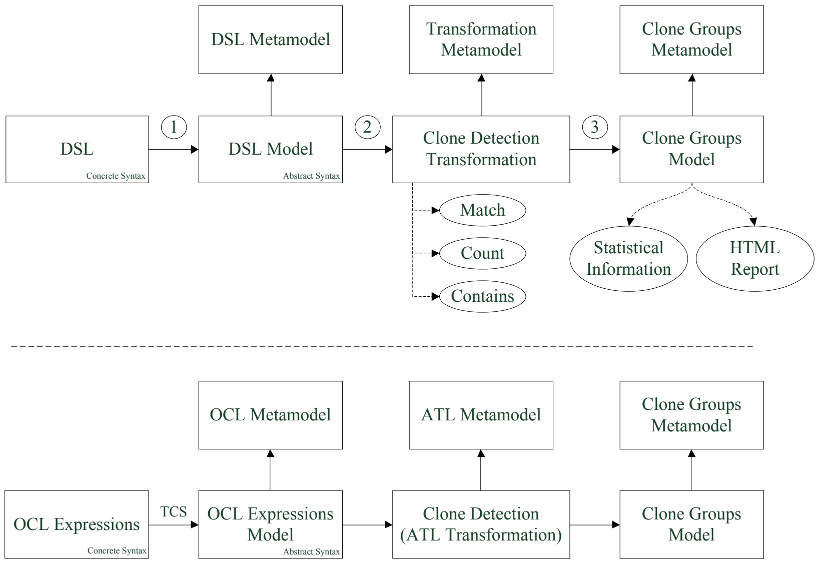


Fig. 1. Clone detection process (in general and for OCL)

The model transformation in Step 2 can be conceptually separated into three main sub-steps. `Match` performs the task of determining whether two language constructs match each other based on similarity rules that are pre-defined. `Count` is used to determine the size of a language construct and is used to filter out elements that are less than a specified size. `Contains` is used to evaluate two language constructs to determine if one is contained or is a sub-construct of another. This is used to filter the detection results to avoid reporting clone groups that can be subsumed by other clone groups. These sub-steps will be described further in the next section as it relates to OCL.

For a different DSL to utilize this process, the three sub-steps would need to be modified to "fit" the metamodel of the DSL. Recent progress in higher order transformations (HOTs) [23] provide a promising mechanism to automate the construction of transformations for different metamodels. In our case, HOTs could potentially be used to generate the three sub-steps associated with the clone detection transformation automatically based on the metamodel of the DSL in question. All three sub-steps perform their tasks by traversing the models that conform to the metamodel. We envision HOTs to provide transformations based on the metamodel for each of the sub-steps, which in turn will generate the necessary functionalities to perform the detection process on the new DSL.

5 Clone Detection Process in OCL

In this section, we provide details of the detection process described in the previous section as it relates to detecting clones among OCL expressions. The bottom part of Figure 1 displays the clone detection process instantiated for OCL.

5.1 Obtaining a Model or Models of OCL Expressions

In order for the detection process using the model transformation mechanism to work, the OCL expressions must first be converted into or *injected as* a model. We use the Textual Concrete Syntax (TCS) [13] DSL, which allows for the textual specification of models and in turn the parsing of these textual specifications into models (i.e., text-to-model). The result of the injection is a model or models of the OCL expressions. This model conforms to the OCL metamodel, which consists of the important elements that are required to model the expressions (i.e., the abstract syntax of the expressions).

5.2 Clone Detection through Model Transformation

Code clones can be categorized into several levels of similarities. Bellon et al. [2] identify exactly identical fragments as Type I clones. Type II clones are fragments that are syntactically the same, but are parameterized through the renaming of properties such as identifier names in the fragments. Near exact matches (i.e., the addition or deletion of a few lines of code) are represented by Type III clones. Roy et al. [21] further propose Type IV clones, which are semantically similar fragments that can differ syntactically. Our approach to clone detection using model transformation performs an evaluation of the OCL expressions and groups expressions that are determined to be duplicates based on clones of Types I and II. The information about the location of the clones within these groups is exactly the elements of the resulting model or target model, which will be described in the next sub-section. We use the ATL Transformation Language (ATL) [12] to generate the transformation.

It should be noted that an OCL expression can consist of a collection of smaller OCL expressions. Our detection process performs duplication evaluation on all OCL expressions. In order to reduce the number of expressions that are evaluated for duplication, we perform some filtering of the expressions and results. We set a minimum "size" for an OCL expression. This size is based on the number of nodes that represents an OCL expression in the model. For example, Figure 2 provides the model for the following OCL expression.

```
ATL!Helper.allInstances()->asSequence()
```

This expression consists of three nodes. Setting the filter to a minimum of four nodes will not include this expression in the detection process. The `Count` transformation helper (i.e., in Figure 1) assists in determining the size of each expression whose information is used to filter out expressions that are less than the pre-defined minimum size.

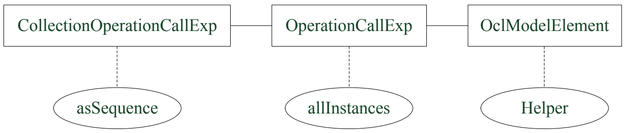


Fig. 2. OCL expression model

Listing 1 outlines in a more imperative style compared to the original ATL transformation of how we detect and group OCL expression clones. All instances of OCL expressions are extracted from the model. This collection is passed through the sub-step of removing expressions that do not meet the minimum number of nodes representing them (i.e., lines 2-6 in Listing 1 where the minimum is set to eight). After this sub-step, each remaining expression is evaluated for duplication. The first iteration creates a clone group for the first expression that is evaluated. The second iteration evaluates the first two expressions. If they do not match, then the second expression is placed in a separate group. The third expression is then evaluated against the two original groups (if the two groups did not match). If the third expression matches one of these groups, then it is included in the group. If the third expression does not match the two groups, it is placed into its own group. The fourth expression is then evaluated against the existing groups and is placed into one of the groups if they match or is put into a new group if none match, and so on.

The comparisons between two expressions (i.e., function `MATCH` in line 11 of Listing 1) is performed by recursively traversing the model. For two expressions to be considered as exact duplicates (i.e., Type I clones), their representations in the model based on the metamodel has to be the same. For example in Figure 2, the first comparison is whether both expressions contain a `CollectionOperationCallExp` element. If so, then the comparison looks for matching `OperationCallExp` elements. And finally, the comparisons look for matching `OCLModelElement` elements. Because the representations in a clone group are structurally identical, the `MATCH` function only compares the first element in the group, because all clones already in the clone group will have the same representation. This detection technique is adopted from the way the Eclipse Java Development Tools (JDT)[1] performs detection within its refactoring framework. The technique uses a Visitor pattern [8] to recursively traverse the nodes in the abstract syntax tree to determine similarity.

Listing 1. Clone grouping process

```
$filteredExps \leftarrow \emptyset$
for all $exp$ in $OCLexpressions$ do
    if COUNT($exp$) $\geq 8$ then
        $filteredExps \leftarrow filteredExps \cup \{exp\}$
    end if
end for
$groups \leftarrow \emptyset$
for all $exp$ in $filteredExps$ do
    $matched \leftarrow false$
    for all $group$ in $groups$ do
        if MATCH($exp$, $group[0]$) then
            $group \leftarrow group \cup \{exp\}$
            $matched \leftarrow true$
        end if
    end for
    if !$matched$ then
        $newGroup \leftarrow exp$
        $groups \leftarrow groups \cup \{newGroup\}$
    end if
end for
$filteredGroups \leftarrow \emptyset$
for all $group$ in $groups$ do
    if !ISSUBGROUP($group$, $filteredGroups$) then
        $filteredGroups \leftarrow filteredGroups \cup \{group\}$
    end if
end for
```

The detection technique in this paper is not flexible enough for detecting Type III-like clones, but is sufficient to find Type I and II clones. For Type II clones, certain elements that are being compared are allowed to differ but still considered as matching (i.e., having parameterized differences). In our case, the elements that are allowed to differ include typical parameterized differences, such

[1] Eclipse JDT, http://www.eclipse.org/jdt

as identifiers names and boolean, integer, real, and string values. More specific to OCL, values of “OCLType” are also allowed to differ. This is similar to a class type in Java.

The *groups* variable in Listing 1 contains the initial collection of clone groups. This collection is passed through the third sub-step, which removes redundant groups whose clones are completely covered by clones in another group (i.e., lines 22-26 in Listing 1). Inside function `ISSUBGROUP`, the `Contains` transformation helper (i.e., in Figure 1) assists in the evaluation of whether one clone group can be subsumed by another clone group and hence be removed from the final results. For example, let two detected groups be $G_1 = (c_1, c_2, c_3)$ and $G_2 = (c_4, c_5, c_6)$. If $c_4 \subseteq c_1$, $c_5 \subseteq c_2$, and $c_6 \subseteq c_3$, then G_2 is not included in the final results.

5.3 OCL Expressions Clones Information

The result of the model transformation is an output model that has grouped together expressions that are either Type I or Type II clones. The information regarding each clone consists of the name of the original file containing the clone and the location of the clone in the file (i.e., the starting and ending lines of the expression represented by the clone). The offsets within each line are also given as part of the location information due to the fact that OCL expressions are not necessarily separated in different lines as is evident with statements in GPLs. During clone detection in GPLs, the statement level is typically the smallest element that is evaluated for duplication, which are usually separated in different lines. For OCL expressions in our case, sub-expressions within larger expressions are also considered during the detection process. These sub-expressions may have been written in the same line as the larger expression, but may not include the entire line. An example can be seen in the expression below in its original layout, where only the sub-expression between “<<” and “>>” is part of the detected clone. The first part of the first line and last part of the last line are not part of the clone. This clone was found in the XML2DSL ATL transformation model.

```
XML!Element.allInstances() -> select(e | << if e.name = 'model'
           then if e.parent.name = 'dmd'
               then e.getAttrVal('name') <> 'Core'
               else false endif
           else false
       endif >> )-> first()
```

The information about the location of the clones is used to calculate statistical information, which is summarized in the following section. In addition, the clone information is also used to generate an HTML report consisting of the actual OCL expressions related to the clones. This report is used for further manual evaluation of the characteristics of the clones.

6 Evaluation Results

We evaluated the duplication of expressions in several artifacts containing OCL-like and generic OCL expressions. The first collection of artifacts were ATL transformations obtained from the ATL transformation zoo,[2] because ATL includes OCL-like expressions as a very important part of its language. The remaining collection of artifacts comes from UML projects in which OCL expressions are used in the specification of UML models. Three of these projects come from case studies of the use of UML in software development: DBLP,[3] EURent [7], and OSCommerce.[4] DBLP is a web-based application for displaying computer science-related bibliography. EURent is a well-known case study that represents a fictitious rental car company. OSCommerce represents an open source e-commerce application. The remaining two artifacts are the UML specification itself, which contains OCL expressions within the specification[5] and test cases containing OCL expressions obtained from the Dresden Toolkit.[6] This toolkit supports the parsing and evaluation of OCL constraints.

Table 2 summarizes the amount of cloning detected within the various OCL artifacts. We do not show cloning in terms of number of lines of code, because as stated in Section 5.3, several OCL expressions can potentially be written in the same line. Instead, Table 2 shows the number of expressions that were part of the detected clones compared to the number of total expressions that were evaluated for expressions with eight or more nodes. This is different from the lines of code measurement as seen in Table 1. It should be noted that this total includes sub-expressions of OCL that are found within larger expressions, which are also counted.

Table 2. Cloned expressions (minimum of eight nodes)

Artifact	Expressions	Cloned expressions
ATL Transformations	6659	3923 (58%)
DBLP Case Study	34	21 (61%)
Dresden Toolkit Test Cases	5803	4421 (76%)
EURent Case Study	625	199 (31%)
OSCommerce Case Study	574	330 (57%)
UML Specification	376	107 (28%)

It can be seen in Table 2 that cloned expressions account for 28% or more of the overall expressions that were evaluated. When we reduced the minimum size of the expressions to four nodes, the percentage of cloned expression were higher. For example, DBLP had 65 out of 82 (79%) expressions that were part of clones. EURent had 605 out of 1205 (50%) expressions that were part of clones.

[2] ATL Transformations, http://www.eclipse.org/m2m/atl/atlTransformations
[3] The DBLP Case Study, http://guifre.lsi.upc.edu/DBLP.pdf
[4] OSCommerce Conceptual Schema, http://guifre.lsi.upc.edu/OSCommerce.pdf
[5] UML Specification, http://www.omg.org/spec/UML/2.3/Superstructure
[6] Dresden OCL, http://www.dresden-ocl.org/index.php/DresdenOCL

OSCommerce had 1090 out 1372 (79%) expressions and the UML specification had 362 out of 796 (45%) expressions that were included in the detected clone groups. Based on these numbers, we make an observation that cloning occurs frequently in OCL code. A related question of whether within these collections of clones interesting clones can be found is considered in the remainder of this section.

We were interested in the extent of clones found within several different models (i.e., inter-duplication). From the collection of artifacts, only OCL expressions in ATL transformations artifact were contained in multiple models. A typical ATL transformation set up transforms a source model that conforms to a metamodel, to a target model conforming to the same or different metamodel. Related to inter-duplication, we were interested to know whether clones existed in transformations in general that involved different metamodels or whether clones are more associated to transformations in which the metamodel is the same.

Table 3 summarizes the amount of inter-duplication among ATL models based on clone detection that is filtered for three different minimum expression sizes. The table classifies clone groups based on the number of models they reside in. For example, during the detection of expressions with nodes greater than or equal to eight, 125 clone groups had clones residing in two models. It can be seen in the table that most of the clone groups consist of clones residing in the same ATL model. A closer look at the clones found in multiple ATL models reveals that most duplication is related to transformations where either the source or target conforms to the XML metamodel. In other words, the same OCL expressions used to transform to or from an XML model is used over and over again in other XML-related transformation. For example, the following expression pattern represented by 14 clones in one group was scattered in eight ATL models (i.e., XMLHelpers, XML2MySQL, XML2Make, XML2GeoTrans, XML2DXF, XML2Ant, XML2Maven, and XML2Book). As the names of the models suggest, the metamodel of the source model was XML. The expression itself traverses through the child elements of a node and finds the value of an attribute with the name equal to `c.name`.

```
self.children
  ->select(c | c.oclIsKindOf(XML!Attribute) and c.name = name)
```

A smaller clone group containing three clones in three models (i.e., XML2DSL, XML2ATOM, and XML2RSS) represent expressions, one of which is given following this paragraph, that extract the text from an XML element. Again, this is a typical function in transformations related to XML. These expressions are actually part of helper definitions, which suggests that the functionality has been modularized. However, the modularization is only local and thus each model contains identical helper definitions. The helper definition could instead be placed in a more general helper library, which can then be included in the transformation models.

Table 3. Clone groups classified based on the number of models they reside in

	Number of models										
Expression size	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10-19*	$\geq$ *20*
$\geq$ 10 nodes	298	89	28	11	3	3	1	1	0	0	0
$\geq$ eight nodes	402	125	42	16	4	3	3	4	0	2	0
$\geq$ six nodes	486	171	60	42	10	8	3	7	1	10	1

```
if e.isEmpty() then
  ''
else
  let r : Sequence(XML!Element) = e->select(f | f.name = name)
  in
    if r.isEmpty() then
      ''
    else
      r->collect(d | d.children)->flatten()
       ->select(f | f.oclIsTypeOf(XML!Text))
       ->select(g | g.name = '#text')->first().value
    endif
endif
```

An example of inter-duplication that is not XML-specific is found in a clone group whose clones reside in 14 models (i.e., JavaSource2Table, SD2flatSTMD, BibTeX2DocBook, MySQL2KM3, PathExp2PetriNet, Mantis2XML, SpreadsheetMLSimplified2XML, SoftwareQualityControl2Mantis, Monitor2Semaphore, Bugzilla2XML, SoftwareQualityControl2Bugzilla, SSL2SDLTransformation, TextualPathExp2PathExp, and SpreadsheetMLSimplified2SoftwareQualityControl). These clones represent a commonly used expression in ATL called `resolveTemp`, which allows for the referencing of target model elements. An example is given in the following expression.

```
b.attachment->collect(e | thisModule.resolveTemp(e, 'a'))
```

From the observations of inter-duplication in ATL models, we can note that the majority of cloning in multiple models is related to transformation involving the XML metamodel. Unfortunately, the other OCL artifacts were not separated into different models. We were unable to consider the other artifacts for the evaluation of inter-duplication and hence the evaluation of ATL models alone does not provide a strong argument of whether inter-duplication within OCL is prevalent or not.

Table 4 lists the sizes of the detected clone groups. One observation is that in each of the results, approximately half of the groups contain only a pair of clones. For example, in the clone groups from the ATL transformations, 314 out of 601 clone groups contain only pair-wise clones. However, larger sized clone groups are also evident some of which are considerable in size. For example, we observed a clone group of 43 clones all residing in the same ATL model called UMLDI2SVG that consisted of an expression, which selects elements of `typeInfo` of either "CompartmentSeparator" or "NameCompartment." Although these clones reside in a single model, identifying both parameterized cases of the expression can not be done by a simple textual search function. Clone detection provides a more robust searching mechanism that can ignore terminal values. The following is an example of the cloned expression.

```
n.contained
  ->select(e | e.semanticModel.typeInfo = 'CompartmentSeparator')
```

Table 4. Clone group sizes

	Number of clones											
Artifact	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10-19*	*20-29*	*30-39*	$\geq$ *40*
ATL Transformations	314	99	64	21	23	19	8	10	27	6	7	3
DBLP Case Study	2	3	0	0	0	0	0	0	0	0	0	0
Dresden Toolkit Test Cases	185	54	31	15	13	3	1	2	19	24	0	2
EURent Case Study	27	3	2	2	1	0	0	2	2	0	0	0
OSCommerce Case Study	22	8	4	0	3	2	0	1	7	2	0	0
UML Specification	19	2	2	1	0	0	0	0	0	0	0	0

In a separate case, a group of clones can be found in two clone groups in the Dresden Toolkit Test Cases, an example of which can be seen in the following expression. This is an interesting group of clones, because it represents a typical constraint in OCL that asserts the names of two customers cannot be equal. In this case, modularizing the clones would not be the main goal. Instead this pattern could be included in the repository of patterns that was described in Section 3 and could be used to assist users during the coding of such constraints.

```
self.participants
  ->forAll(c1 : Customer, c2:Customer |
      c1 <> c2 implies c1.name <> c2.name)
```

6.1 Summary of Evaluation Results

Table 2 shows the varying degrees of cloning among the OCL-related artifacts that we evaluated. We can not make a direct comparison with Table 1, because the measurement units are different. However, we can see that the percentage of cloned expressions compared to the total number of expressions evaluated is considerable. Table 4 revealed that small clone group sizes comprised around half of clone groups reported from the detection process. However, larger clone groups are still evident and clearly relevant. Furthermore, we have provided examples of interesting clone groups described earlier in this section that were found from the manual evaluation of the clone groups. We conclude that the occurrences of cloning in OCL is evident and interesting clones can be found, thus warranting future efforts to provide maintenance assistance in OCL as it relates to cloning and clone detection.

Regarding the specific examples of interesting clones in our evaluation, observing the OCL-like expressions in ATL models revealed that much of the cloning among multiple models (i.e., inter-duplication) occurred in transformations involving the XML metamodel. Modularizing some of these clones was actually done, but only within individual transformation models. We suggest writing a general library of helper definitions consisting of common tasks performed during transformations that are associated with the XML metamodel, information of which could be obtained from the clone detection results.

ATL provides a construct to "import" helper libraries using the `uses` command. However, OCL currently does not provide such a mechanism in which a general library of helper definitions can be shared among OCL instances. Chimiak-Opoka [3] has proposed OCLLib, which offers such a feature. We suggest evaluating the clone detection results with the OCL community where the results provide empirical information of commonly used expressions. This information can be used to support the development of OCLLib or even re-engineering the OCL language to include patterns that are used much more frequently.

Clone detection results provide general cloning information. In order for the identification of interesting clone groups (i.e., groups representing common patterns as described in Section 3) to be more effective, the detection results must be filtered to reduce the number of uninteresting clone groups. For example, filtering could remove clone groups containing expressions such as expressions that are too simple, but keep expressions such as common constraints used in UML models. Filtering of detection results is considered part of the clone analysis phase (i.e., post-detection activity). We seek to apply one or more of analysis techniques used on GPLs to assist in filtering the detection results of OCL expressions, but also consider newer approaches in order to determine common patterns of expressions.

7 Threats to Validity

The clone detection process described in this paper is able to detect exact matching expressions and expressions that are syntactically the same, but may consist of differing terminal values. This restricts the types of clones that can be detected, because clones with slightly differing syntax are not detected and hence are not included in the overall evaluation.

Current tools that provide automated detection of clones typically contain several adjustable configuration settings that when changed can return different results of cloning. Popular tools such as CCFinder [14] and CloneDR [1] consist of several adjustable settings. The detection technique described in this paper also allows changing certain properties of the detection process, e.g., number of minimum nodes allowable for an expression to be included in the duplication evaluation. This single setting can return different numbers of clone groups as seen in Table 3 when the number of nodes varies. Settings that were selected could potentially influence the conclusions of the clone evaluation.

The OCL artifacts evaluated in the paper represent a varying collection of artifacts (i.e., both generic and OCL-like expressions). However, the artifacts have limitations, such as the small number of expressions in the DBLP Case Study. In addition, all artifacts except for the Dresden Toolkit Test Cases represent OCL expressions in actual ATL transformations or UML model specifications. The expressions from the Dresden Toolkit are for testing purposes, but still consist of expressions that would be used in the specification of a model. We are limited in the amount of publicly available OCL artifacts that can be used for evaluation and hence we advocate the establishment of repositories of DSL artifacts.

Because of the specific nature of DSLs, many different DSLs are in use today. Further validation of cloning in DSLs requires more studies of other DSLs. We have provided an evaluation of only one DSL, which limits the support of our conclusions as it relates to DSLs in general. However, we consider this evaluation to be an initial evaluation that will be complemented by evaluations of cloning in other DSLs.

8 Conclusion and Future Work

In this paper, we have provided an evaluation of cloning in OCL expressions as an initial step toward a broader understanding of cloning in DSLs. We offered scenarios of the utilization of clone detection in DSL code maintenance. However, we emphasized the need to determine the relevance of the topic of cloning in DSLs. In order to determine this, we first described a MDE-based clone detection technique that was used to detect clones in several OCL artifacts. Based on our evaluation of clone detection results, there is a considerable amount of cloning among the expressions, and we conclude that cloning in OCL specifically is evident. In addition, interesting clones can be found among the reported clone groups. These initial observations through OCL support future efforts of utilizing clone detection with OCL code, including the scenarios that were previously proposed.

Future work related to the clone detection technique described in this paper includes developing a more robust clone detection process, which includes being more “flexible” in terms of allowing more types of clones to be detectable. In addition, further studies of more artifacts (both OCL and other DSLs) will provide more representative characteristics of cloning within DSLs. The evaluation of other types of DSLs would provide a broader understanding of cloning in DSLs. However, with the nature of DSLs being “specific,” a general understanding encompassing all DSLs may not be possible, in which case evaluation will be focused separately among the DSLs. Finally, based on the results of the evaluation of artifacts in this paper, we are currently working on the application scenarios of clone detection as described in Section 3 to determine their feasibility and usability.

References

1. Baxter, I., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: International Conference on Software Maintenance, pp. 368–377 (1998)
2. Bellon, S., Koschke, R., Antoniol, G., Krinke, J., Merlo, E.: Comparison and evaluation of clone detection tools. IEEE Transactions on Software Engineering 33(9), 577–591 (2007)
3. Chimiak-Opoka, J.: OCLLib, OCLUnit, OCLDoc: Pragmatic Extensions for the Object Constraint Language. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 665–669. Springer, Heidelberg (2009)
4. Deissenboeck, F., Hummel, B., Juergens, E., Pfaehler, M., Schaetz, B.: Model clone detection in practice. In: International Workshop on Software Clones, pp. 57–64 (2010)
5. van Deursen, A., Klint, P.: Little languages: Little maintenance. Journal of Software Maintenance: Research and Practice 10(2), 75–92 (1998)
6. Duala-Ekoko, E., Robillard, M.: Clone region descriptors: Representing and tracking duplication in source code. ACM Transactions of Software Engineering and Methodology 20(1), 1–31 (2010)
7. Frias, L., Queralt, A., Olivé, A.: Eu-rent car rentals specification. Tech. Rep. LSI-03-59-R, Technical University of Catalonia - Departament de Llenguatges i Sistemes Informatics (2003)
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley, Boston (1995)
9. Gruska, N., Wasylkowski, A., Zeller, A.: Learning from 6,000 projects: Lightweight cross-project anomaly detection. In: International Symposium on Software Testing and Analysis, pp. 119–130 (2010)
10. Jiang, L., Misherghi, G., Su, Z., Glondu, S.: DECKARD: Scalable and accurate tree-based detection of code clones. In: International Conference on Software Engineering, pp. 96–105 (2007)
11. Johnson, J.H.: Substring matching for clone detection and change tracking. In: International Conference on Software Maintenance, pp. 120–126 (1994)
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. Jouault, F., Bézivin, J., Kurtev, I.: TCS: A DSL for the specification of textual concrete syntaxes in model engineering. In: Jarzabek, S., Schmidt, D.C., Veldhuizen, T.L. (eds.) International Conference on Generative Programming and Component Engineering, pp. 249–254 (2006)
14. Kamiya, T., Kusumoto, S., Inoue, K.: CCFinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering 28(7), 654–670 (2002)
15. Kim, H., Jung, Y., Kim, S., Yi, K.: MeCC: Memory comparison-based clone detector. In: Taylor, R.N., Gall, H., Medvidovic, N. (eds.) International Conference on Software Engineering, pp. 301–310 (2011)
16. Kim, M., Sazawal, V., Notkin, D., Murphy, G.: An empirical study of code clone genealogies. In: European Software Engineering Conference held Jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 187–196 (2005)
17. Lämmel, R., Pek, E.: Vivisection of a non-executable, domain-specific language - understanding (the usage of) the P3P language. In: IEEE International Conference on Program Comprehension, pp. 104–113 (2010)

18. Li, H., Thompson, S.: Clone detection and removal for Erlang/OTP within a refactoring environment. In: Workshop on Partial Evaluation and Program Manipulation, pp. 169–178 (2009)
19. Li, Z., Lu, S., Myagmar, S., Zhou, Y.: CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In: Symposium on Operating Systems Design and Implementation, pp. 289–302 (2004)
20. Pham, N.H., Nguyen, H.A., Nguyen, T.T., Al-Kofahi, J.M., Nguyen, T.N.: Complete and accurate clone detection in graph-based models. In: International Conference on Software Engineering, pp. 276–286 (2009)
21. Roy, C., Cordy, J., Koschke, R.: Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Science of Computer Programming 74(7), 470–495 (2009)
22. Schmidt, D.: Guest editor's introduction: Model-driven engineering. IEEE Computer 39(2), 25–31 (2006)
23. 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)

Uniform Modularization of Workflow Concerns Using Unify

Niels Joncheere* and Ragnhild Van Der Straeten

Vrije Universiteit Brussel, Software Languages Lab
Pleinlaan 2, 1050 Brussels, Belgium
{njonchee,rvdstrae}@vub.ac.be

Abstract. Most state-of-the-art workflow languages offer a limited set of modularization mechanisms. This typically results in monolithic workflow specifications, in which different concerns are scattered across the workflow and tangled with one another. This hinders the design, the evolution, and the reusability of workflows expressed in these languages. We address this problem by introducing the Unify framework, which supports uniform modularization of workflows by allowing all workflow concerns — including crosscutting ones — to be specified in isolation of each other. These independently specified workflow concerns can then be connected to each other using a number of workflow-specific connectors. We discuss the interaction of the most invasive connector with the workflows' control flow and data perspectives. We instantiate the framework towards two state-of-the-art workflow languages, i.e., WS-BPEL and BPMN.

1 Introduction

Workflow management systems have become a popular technique for automating processes in many domains, ranging from high-level business process management to low-level web service orchestration. A workflow is created by dividing a process into different activities, and by specifying the ordering in which these activities need to be performed. This ordering is called the control flow perspective.

Separation of concerns [1] is a general software engineering principle that refers to the ability to identify, encapsulate, and manipulate only those parts of software that are relevant to a particular concept, goal, or purpose. These parts, called *concerns*, are the primary motivation for organizing and decomposing software into manageable and comprehensible modules.

Realistic workflows consist of several concerns, which are connected in order to achieve the desired behavior. However, if all of these concerns need to be specified in a single, monolithic workflow specification, it will be hard to add, maintain, remove or reuse these concerns. Although most workflow languages

* Funded by the Belgian State – Belgian Science Policy through the Interuniversity Attraction Poles program.

allow decomposing workflows into sub-workflows, this mechanism is typically aimed at grouping activities instead of facilitating the independent evolution and reuse of concerns. Moreover, a workflow can only be decomposed according to one dimension with this construct, and concerns that do not align with this decomposition end up scattered across the workflow and tangled with one another. Such concerns are called *crosscutting concerns* [2]. These problems have been discussed in related work by ourselves [3,4] and others [5,6,7], where they are mainly tackled using *aspect-oriented programming* for workflows. Nevertheless, the problems are not yet fully addressed by the proposed solutions.

The goal of our current solution is to facilitate independent evolution and reuse of *all* workflow concerns, i.e., not merely crosscutting concerns. This can be accomplished by improving the modularization mechanisms offered by the workflow language. We propose an approach called Unify which provides a set of workflow-specific modularization mechanisms that can be readily employed by a wide range of existing workflow languages. Unify facilitates specifying workflow concerns as separate modules. These modules are then composed using versatile connectors, which specify how the concerns are connected. The main contributions of Unify are the following:

1. Existing research on modularization of workflow concerns is aimed at only modularizing crosscutting concerns [6,7,3], or at only modularizing one particular kind of concern, such as monitoring [8]. Unify, on the other hand, aims to provide a uniform approach for modularizing all workflow concerns.
2. Existing aspect-oriented approaches for workflows are fairly straightforward applications of general aspect-oriented principles, and are insufficiently focused on the concrete context of workflows. Unify improves on this by allowing workflow concerns to connect to each other in workflow-specific ways, i.e., the connector mechanism supports a number of dedicated concern connection patterns that are not supported by other approaches.
3. Unify is designed to be applicable to a wide range of concrete workflow languages. This is accomplished by defining its connector mechanism in terms of a general, extensible base language meta-model.
4. Unify defines a clear semantics for its modularization mechanism. This facilitates the application of existing workflow verification techniques.
5. The Unify implementation can either be used as a separate workflow engine, or as a pre-processor that is compatible with existing workflow engines.

The structure of this paper is as follows. Section 2 specifies the motivation for Unify, and introduces a running example. Section 3 provides an initial description of our approach by showing how the running example could be developed from scratch when using Unify. Sections 4 and 5 introduce the meta-model for our base language and connector mechanism, respectively. Section 6 discusses the interaction of our connector mechanism with the control flow and data perspectives, and discusses the semantics of our connector mechanism. Section 7 describes our implementation, Section 8 gives an overview of related work, and Section 9 states our conclusions and outlines future work.

2 Motivation

Consider the workflow in Figure 1, which is a simplified version of an automated order handling process for an online book store. The workflow is visualized using the Business Process Model and Notation (BPMN) [9]; a brief overview of this notation is given in the legend at the bottom right of the figure. The workflow starts at the start event at the top of the figure. It first performs the *Login* and *SelectBooks* activities in parallel. The workflow then proceeds with the *SpecifyOptions* activity, after which the control flow is split again. A first branch contains the *Pay* and *SendInvoice* activities, while a second branch contains the *ProcessOrder* and *Ship* activities. The *VerifyBankAccount* activity synchronizes both branches. The last activity to be executed is the *ProcessReturns* activity, after which the workflow ends at the end event. Please note that only the contents of the *SelectBooks*, *Pay*, and *Ship* activities are shown, whereas the contents of other activities are omitted in the interest of brevity.

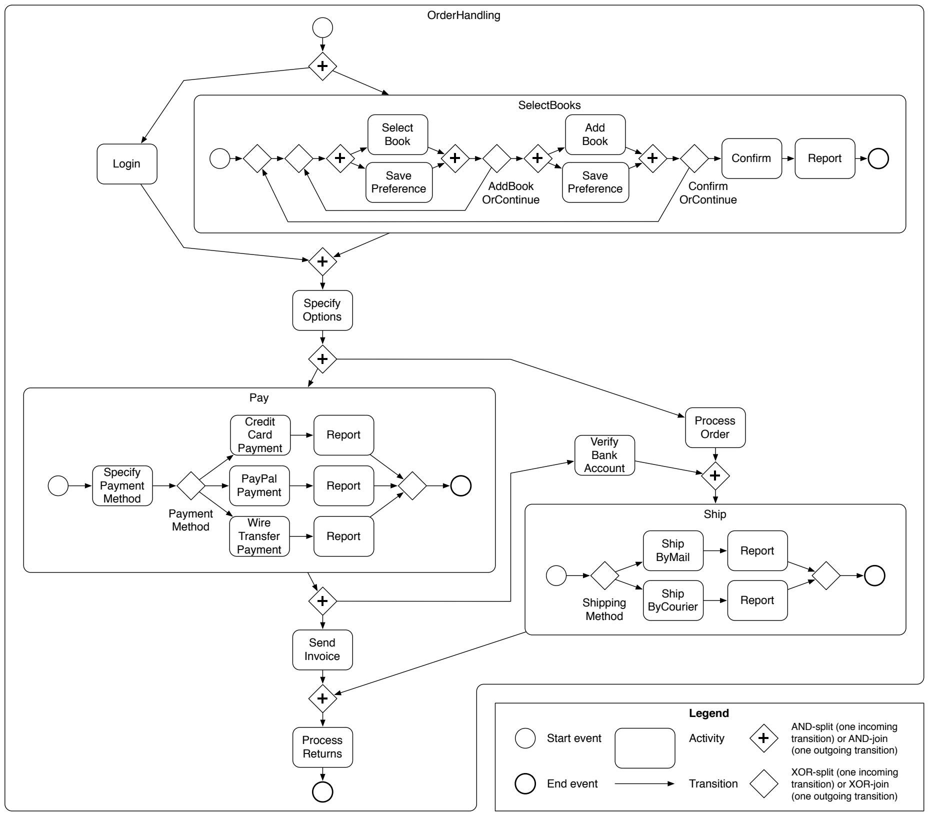


Fig. 1. Example order handling workflow, expressed using BPMN

Like any realistic software application, the workflow in Figure 1 consists of several *concerns* — parts that are relevant to a particular concept, goal, or purpose — which are connected in order to achieve the workflow's desired behavior. The main concern is obviously *order handling*. This concern has already been hierarchically decomposed into sub-concerns — such as *book selection*, *payment* and *shipping* — using the *composite activity* construct. Other concerns are *preference saving*, *reporting* and *bank account verification*, which occur at various places across the workflow. The general software engineering principle of *separation of concerns* argues that applications should be decomposed into different modules in such a way that each concern can be manipulated in isolation of other concerns. However, many current workflow languages do not allow decomposing workflows into different modules. For example, a workflow expressed using WS-BPEL [10] (the de facto standard in workflow languages) is a single, monolithic XML file that cannot be straightforwardly divided into sub-workflows. This lack of modularization mechanisms makes it hard to add, maintain, remove, or reuse concerns. In order to improve separation of concerns in workflows, workflow languages should allow concerns to be specified in isolation of each other.

However, allowing concerns to be specified in isolation of each other is not sufficient: in order to obtain the desired workflow behavior, workflow languages should also provide a means of specifying how a workflow's concerns are connected to each other.

In existing workflow languages, the only kind of connection that is supported is typically the classic sub-workflow pattern: a main workflow explicitly specifies that a sub-workflow should be executed. The choice of which sub-workflow is to be executed is made at design time, and it is hard to make a different choice afterwards. By delaying the choice of which sub-workflow is to be executed, the coupling between main workflow and sub-workflow is lowered, and separation of concerns is improved. In the workflow in Figure 1, one could for example vary the behavior of the workflow by deploying a different *Pay* sub-workflow in different situations.

A second kind of connection between concerns is useful when concerns *crosscut* a workflow: some concerns cannot be modularized cleanly using the sub-workflow decomposition mechanism, because they are applicable at several locations in the workflow. The reporting concern, for example, is present at several locations in the workflow in Figure 1. The sub-workflow construct does not solve this problem, since sub-workflows are called explicitly from within the main workflow. This makes it hard to add, maintain, remove or reuse such crosscutting concerns. This problem has been observed in general aspect-oriented research [2]. Aspect-oriented extensions to WS-BPEL, such as AO4BPEL [6] and Padus [3], allow specifying crosscutting concerns in separate aspects. An aspect allows specifying that a certain workflow fragment, called an *advice*, should be executed *before*, *after*, or *around* a certain set of activities in the base workflow. In the workflow in Figure 1, one could for example specify that the *Report* activity needs to be performed after the *Confirm* activity and after each of the three *Payment* and two *Ship* activities, without explicitly invoking the *Report* activity at each of those places. However, these aspect-oriented extensions use a new language construct for specifying crosscutting concerns, i.e., aspects. This means that concerns which are specified using the aspect construct can only be reused as an aspect, and not as a sub-workflow. On the other hand, concerns which are specified using the sub-workflow construct can only be reused as a sub-workflow, and not as an aspect.

Moreover, the aspect-oriented extensions mentioned above only support the basic concern connection patterns (*before*, *after*, or *around*) that were identified in general aspect-oriented research, and do not sufficiently consider the specifics of the workflow context. They lack support for other patterns such as parallelism and choice. For example, the before, after or around patterns do not provide an elegant way of specifying that the *SavePreference* activity should be performed *in parallel with* the *SelectBook* and *AddBook* activities. Furthermore, it is completely impossible to specify more advanced connections between concerns, e.g., specifying that the *VerifyBankAccount* activity should be executed after the *Pay* activity has been executed and before the *Ship* activity is executed, which would thus synchronize the two parallel branches by introducing a new AND-split and -join in the order handling workflow.

Finally, the aspect-oriented extensions mentioned above are all targeted at WS-BPEL, and cannot be applied easily to other languages. Each of these approaches also favors a specific implementation technique; for example, AO4BPEL can only be executed using a modified WS-BPEL engine, and Padus can only be used as a pre-processor. More variability in terms of the applicable languages and possible implementation techniques would make a modularization approach more widely applicable.

3 Developing a Workflow Using Unify

There are two main scenarios for applying Unify to workflow development. In the first scenario, Unify is used to improve a workflow that has already been developed using an existing workflow language, but without any regard for separation of concerns. Unify could then be used to decompose the existing workflow into a number of different modules, which each correspond to a concern, and which are connected to each other in order to achieve the original behavior. We will not consider this first scenario in this paper. The second scenario assumes that a developer is creating a new workflow from scratch, perhaps with a library of previously implemented concerns at his disposal. In this section, we will introduce Unify using this second scenario.

The first step in developing a workflow using Unify is to identify its concerns. In the example from Figure 1, these are, among others, *order handling*, *book selection*, *payment*, *shipping*, *preference saving*, and *reporting*. Unify promotes implementing a workflow's concerns as separate modules.[1] This can be achieved using the *composite activity* construct. Figure 2 shows how the concerns that

[1] Deciding which concerns should be modularized is partly a matter of personal preference, and is not the focus of our research.

were mentioned in the previous section could be specified separately. Note that each of the composite activities in Figure 2 contains less activities than the corresponding composite activity in Figure 1. The Unify base language, which is discussed in Section 4, defines the abstract syntax of our workflow concerns.

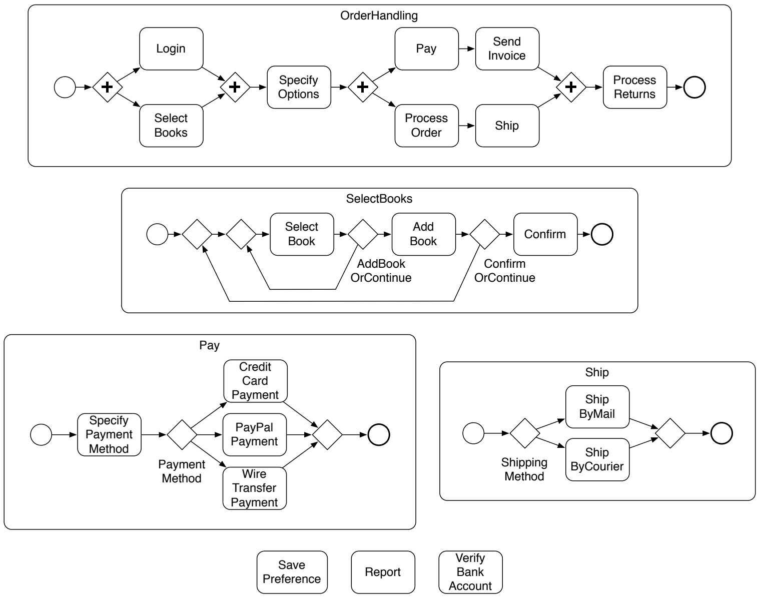


Fig. 2. Independently specified workflow concerns

The advantage of specifying workflow concerns as separate composite activities is better separation of concerns: the different parts of a concern are no longer scattered across the workflow(s), or tangled with one another. After the concerns have been identified and implemented (or retrieved from a library of previously implemented concerns), the connections between the concerns should be specified. We identify two main categories of connections between concerns:

– **Anticipated concern connections** are concern connections that are explicitly anticipated by one of the concerns: this concern is aware, at design time, of the fact that it will connect to another concern at a certain point in its execution.
– **Unanticipated concern connections** are concern connections that are *not* explicitly anticipated by the concerns: the concerns are not aware of the fact that they will connect to each other at a certain point in their execution.

An example of the former is apparent in the *OrderHandling* concern in Figure 2: this concern contains, among others, the *SelectBooks*, *Pay* and *Ship* activities, which will need to be realized by connecting them to the *SelectBooks*, *Pay* and *Ship* concerns that are shown in the middle of the figure.

An example of the latter is present in Figure 2 as well: neither the *SelectBooks*, *Pay* nor *Ship* concerns contain any reference to the *Report* concern, whereas an unanticipated concern connection can be made between the *Report* concern and those three concerns.

Unify allows specifying both anticipated and unanticipated connections using its *connector* construct. In our example, the following connectors can, among others, be used to connect the different concerns:

1. An **activity connector** can be used to specify that the *SelectBooks* activity in the *OrderHandling* concern should be executed by executing the *SelectBooks* concern (and likewise for the *Pay* and *Ship* activities and concerns). Thus, activity connectors allow hierarchically decomposing workflows into different concerns.
2. An **after connector** can be used to specify that the *Report* concern should be executed after the *Confirm* activity in the *SelectBooks* concern, after the three *Payment* activities in the *Pay* concern, and after the two *Ship* activities in the *Ship* concern. Thus, after connectors allow expressing the *after* pattern that is currently offered by aspect-oriented approaches.
3. A **parallel connector** can be used to specify that the *SavePreference* concern should be executed in parallel with the *SelectBook* and *AddBook* activities in the *SelectBooks* concern. Thus, parallel connectors allow expressing a pattern that is not currently offered by aspect-oriented approaches.
4. A **free connector** can be used to specify that the *VerifyBankAccount* concern should be executed after the *OrderHandling* concern's *Pay* activity has been executed and before its *Ship* activity is executed. Thus, free connectors allow invasively changing a concern's control flow by introducing additional splits and joins, e.g., in order to synchronize two parallel branches.

Activity connectors express anticipated concern connections, while the other connectors express unanticipated concern connections. The Unify connector mechanism, which offers other connectors in addition to the ones mentioned above, and which is discussed in Section 5, defines the abstract syntax of our connectors.

We have defined a textual concrete syntax for our connectors, which is available in Backus–Naur form at [11]. Listing 1 shows how the above activity, after, and free connectors can be expressed using this syntax. If one would apply all the above connectors to the concerns of Figure 2, one would obtain the workflow of Figure 1.

4 The Unify Base Language

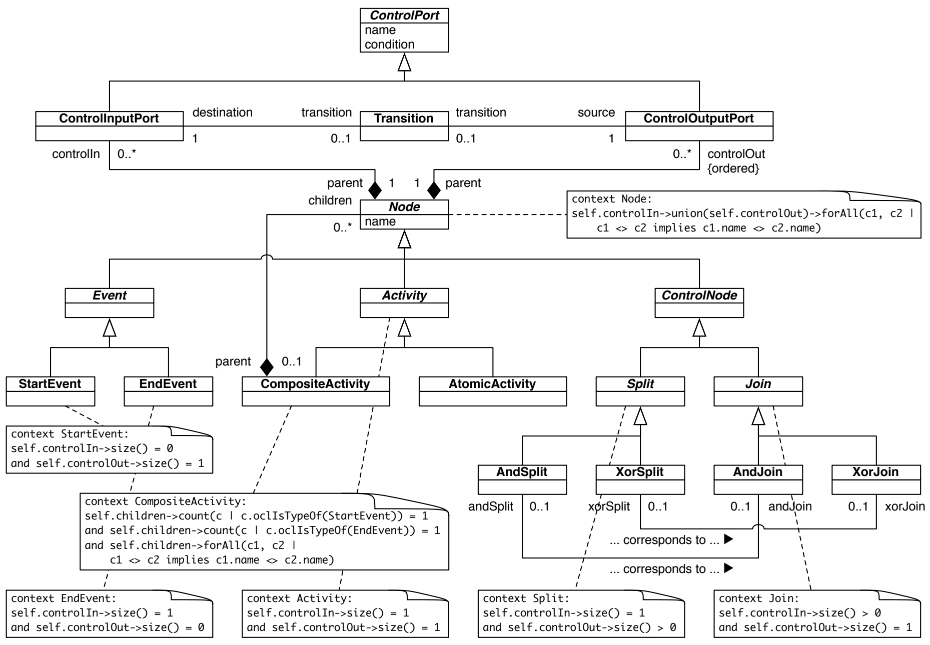
Unify is designed to be applicable to a range of concrete workflow languages, as long as they conform to a number of basic assumptions. These assumptions are expressed as a meta-model for our workflow concerns. We do not restrict ourselves to any particular concrete workflow language as long as it can be defined as an extension to this meta-model. The meta-model allows expressing *arbitrary workflows* [12], i.e., workflows whose control flow is not restricted to

```
SelectBooksConnector:
CONNECT OrderHandling.SelectBooks TO SelectBooks

ReportConnector:
CONNECT Report AFTER activity("SelectBooks\.Confirm|Pay\..*Payment|Ship\.ShipBy.*")

VerifyBankAccountConnector:
CONNECT VerifyBankAccount
AND-SPLITTING AT controlport(OrderHandling.Pay.ControlOut)
JOINING AT controlport(OrderHandling.Ship.ControlIn)
```

Listing 1. Example activity, after, and free connectors

a predefined set of control flow patterns, and is therefore also compatible with more restricted workflows such as *structured workflows*.

Figure 3 provides the meta-model for our workflow concerns. This meta-model does not contain the Unify connector mechanism, which is given in Section 5. The complete Unify meta-model is the union of these two meta-models. The meta-models are expressed using UML, with well-formedness constraints specified in OCL.



```
context Node:
self.controlIn->union(self.controlOut)->forAll(c1, c2 |
    c1 <> c2 implies c1.name <> c2.name)
```

```
context StartEvent:
self.controlIn->size() = 0
and self.controlOut->size() = 1
```

```
context CompositeActivity:
self.children->count(c | c.oclIsTypeOf(StartEvent)) = 1
and self.children->count(c | c.oclIsTypeOf(EndEvent)) = 1
and self.children->forAll(c1, c2 |
    c1 <> c2 implies c1.name <> c2.name)
```

```
context EndEvent:
self.controlIn->size() = 1
and self.controlOut->size() = 0
```

```
context Activity:
self.controlIn->size() = 1
and self.controlOut->size() = 1
```

```
context Split:
self.controlIn->size() = 1
and self.controlOut->size() > 0
```

```
context Join:
self.controlIn->size() > 0
and self.controlOut->size() = 1
```

Fig. 3. The Unify base language meta-model

A workflow concern is modeled as a *CompositeActivity*. Each *CompositeActivity* has the following children: (1) A *StartEvent*, which represents the point where the *CompositeActivity*'s execution starts. (2) An *EndEvent*, which represents the point where the *CompositeActivity*'s execution ends. (3) Any number of *Activities*, which are the units of work that are performed by the *CompositeActivity*. (4) Any number of *ControlNodes*, which are used to route the *CompositeActivity*'s control flow. (5) One or more *Transitions*, which connect the *StartEvent*, the *EndEvent*, the *Activities* and the *ControlNodes* to each other.

An *Activity* is either a *CompositeActivity* or an *AtomicActivity*. Nested *CompositeActivities* can be used to hierarchically decompose a concern, similar to the classic sub-workflow decomposition pattern. Each *Activity* has a name that is unique among its siblings in the composition hierarchy, and has one *ControlInputPort* and one *ControlOutputPort*. A *ControlInputPort* represents the point where control enters an *Activity*, while a *ControlOutputPort* represents the point where control exits an *Activity*. Each *ControlPort* has a name that is unique among its siblings. Within a *CompositeActivity*, the *StartEvent* is used to specify where the *CompositeActivity*'s execution should start when its *ControlInputPort* is triggered. The *EndEvent* is used to specify where the *CompositeActivity*'s execution should finish, and will cause the *CompositeActivity*'s *ControlOutputPort* to be triggered. Thus, a *StartEvent* only has a *ControlOutputPort*, and an *EndEvent* only has a *ControlInputPort*.

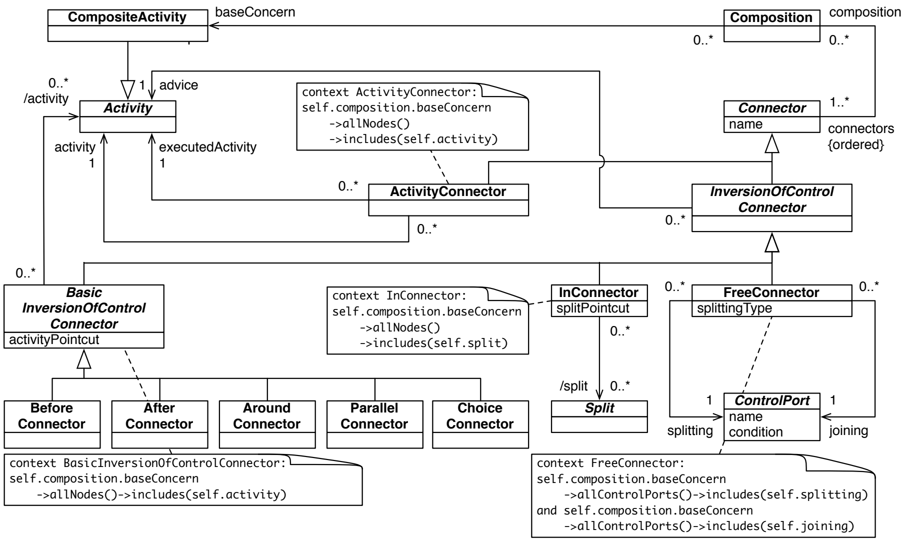
Transitions define how control flows through a *CompositeActivity*. This is done by connecting the *ControlOutputPorts* of the *CompositeActivity*'s *Nodes* to *ControlInputPorts*. *ControlNodes* can be used to route the flow of control, and are either *AndSplits*, *XorSplits*, *AndJoins* or *XorJoins*. A *Split* may have a corresponding *Join*. Together, *Transitions* and *ControlNodes* define a *CompositeActivity*'s control flow perspective.

The Unify base language meta-model does not aim to support every possible control flow pattern that has been identified in existing literature, as our research focuses on the expressiveness of the modularization mechanism rather than on the expressiveness of the individual modules. The meta-model supports the *basic control flow patterns* [13], which are sufficient for expressing most workflows. It does not aim to support more advanced patterns such as cancellation and multiple instances. Due to the generic nature of the Unify base language meta-model, the cores of most workflow languages are compatible with it. We have extended the meta-model towards the cores of the WS-BPEL and BPMN workflow languages.

5 The Unify Connector Mechanism

The Unify connector mechanism is based on aspect-oriented principles [2]. It allows adding the functionality defined by a certain workflow concern (which is modeled as a *CompositeActivity*) at certain locations in another workflow concern. In aspect-oriented terminology, the former concern is the *advice*, while the latter is the base concern. The locations where the advice is added are called *joinpoints*, and are either the base concern's activities, splits, or control ports. The process of adding the functionality to the base concern is called *weaving*.

Unify promotes separation of concerns by allowing workflow concerns to be specified in isolation of each other, as separate *CompositeActivities*. These can be executed separately, or can be connected to other concerns using connectors. Figure 4 shows the meta-model for the Unify connector mechanism. In the interest of brevity, the definition of the *CompositeActivity*'s *allNodes* and *allControlPorts* queries are omitted. These queries return the set of nodes and control ports, respectively, obtained by the transitive closure of the *children* relation.



```
context ActivityConnector:
self.composition.baseConcern
    ->allNodes()
    ->includes(self.activity)
```

```
context InConnector:
self.composition.baseConcern
    ->allNodes()
    ->includes(self.split)
```

```
context BasicInversionOfControlConnector:
self.composition.baseConcern
    ->allNodes()->includes(self.activity)
```

```
context FreeConnector:
self.composition.baseConcern
    ->allControlPorts()->includes(self.splitting)
and self.composition.baseConcern
    ->allControlPorts()->includes(self.joining)
```

Fig. 4. The Unify connector language meta-model

A *Composition* specifies which *CompositeActivity* is its base concern, and which *Connectors* are to be applied to it. The set of *Connectors* is ordered, and the connectors will be applied according to this ordering.

Connectors can be used to add functionality at certain points in a concern. They can be divided into two categories: *ActivityConnectors* and *InversionOfControlConnectors*. In a traditional workflow language, a workflow can be divided into several levels of granularity through the use of sub-workflows. Control passes from the main workflow into sub-workflows and back, with the main workflow specifying when the sub-workflow should be executed. An *ActivityConnector* allows expressing that a certain *Activity* inside a certain concern should be implemented by executing another *Activity*, which thus acts as a sub-concern. By specifying this link in a separate connector instead of inside the concern, we reduce coupling between the concern and the sub-concern, thus promoting reuse.

InversionOfControlConnectors invert the traditional passing of control from main workflow into sub-workflows: they specify that a certain concern should be adapted, while this concern is not aware of this adaptation. In this way, such connectors can be used to add concerns that were not anticipated when the concern to which they are applied was created.

Joinpoints are well-defined points within the specification of a concern where extra functionality — the *advice* — can be inserted using an *InversionOfControlConnector*. Joinpoints in existing aspect-oriented approaches for workflows are either every XML element of the workflow definition [6] or every workflow activity [3]. As is shown in Table 1, our approach supports three kinds of joinpoints: *Activities*, *Splits*, and *ControlPorts*. The joinpoint model is static, which has the advantage of allowing us to define a clear weaving semantics (see Section 6.3).

Table 1. Advice types and joinpoints

Advice type	**Joinpoint**
before	*Activity*
after	*Activity*
around	*Activity*
parallel	*Activity*
choice	*Activity*
in	*Split*
free	*ControlPort*

Table 2. Pointcut predicates

Activity pointcuts
activity(identifierpattern)
compositeactivity(identifierpattern)
atomicactivity(identifierpattern)
Split pointcuts
split(identifierpattern)
andsplit(identifierpattern)
xorsplit(identifierpattern)
Control port pointcuts
controlport(identifier)
controlinputport(identifier)
controloutputport(identifier)

Pointcuts are expressions that resolve to a set of joinpoints, and are used to specify where in the base concern the connector should add its functionality. Because all *Activities*, *Splits* and *ControlPorts* have names that are unique among their siblings, every joinpoint can be uniquely identified by prepending the name of the *Activity*, *Split* or *ControlPort* with the names of their parents. This allows specifying sets of joinpoints as identifier patterns. Pointcuts can be expressed using the predicates in Table 2.

There are seven kinds of *InversionOfControlConnectors*, one for each of the advice types listed in Table 1.

BeforeConnectors, *AfterConnectors*, and *AroundConnectors* allow inserting a certain *Activity* before, after, or around each member of a set of *Activities* in another concern. These correspond to the classic *before*, *after*, and *around* advice types that are common in aspect-oriented research.

ParallelConnectors and *ChoiceConnectors* allow adding a parallel or alternative *Activity* to each member of a set of *Activities* in another concern. These are novel advice types that have not yet been considered in aspect-oriented research.

InConnectors allow adding an *Activity* as an extra branch to an *existing Split*. These are similar to Padus's *in* advice type [3].

FreeConnectors allow (AND- or XOR-) splitting a concern's control flow into another *Activity* at a certain control port, and joining the concern at another control port. These control ports are specified using two pointcuts: the *splitting* pointcut and the *joining* pointcut, respectively. The splitting pointcut specifies where the concern's control flow will be split into the advice activity, and the joining pointcut specifies where the concern will be joined. *FreeConnectors* are more general than *Parallel-*, *Choice-*, and *InConnectors*: *Parallel-* and *Choice-Connectors* allow adding a parallel or alternative *Activity* to an existing *Activity* and *InConnectors* allow adding an *Activity* as an extra branch to an existing *Split*, whereas *FreeConnectors* allow more freedom in where the control flow of the base concern is split into the advice concern, and where the advice concern joins the control flow of the base concern.

In order to widen the applicability of our approach, our connector mechanism is defined in terms of our base language meta-model, which may be extended towards different concrete workflow languages. Thus, we assume that there are no prohibitive semantic differences between the way in which these languages' concepts are mapped to our meta-model. As there is a consensus on the semantics of the basic control flow patterns within the workflow community, this seems to be a safe assumption. The way in which semantic details may be added to our approach is an interesting avenue for future work, and will become more relevant when more advanced control flow patterns are added to the base language meta-model.

6 Discussion

6.1 Interaction with the Control Flow Perspective

The connector mechanism described above allows invasively changing a base concern by connecting other concerns to it. In this subsection, we focus on the effects of the connector mechanism on a base concern's control flow. Our goal here is to prevent that connecting a well-behaved concern to a well-behaved base concern results in a composition that is not well-behaved. A concern is well-behaved if it can never deadlock nor result in multiple active instances of the same activity [12].

Before-, *After-*, *Around-*, *Parallel-* or *ChoiceConnectors* cannot negatively influence a base concern's control flow because they merely result in the execution of some extra behavior around the joinpoint activity. Thus, they cannot influence any part of the base concern's control flow other than the execution of the joinpoint activity itself. *InConnectors* cannot negatively influence a base concern's control flow because they merely result in the addition of an extra branch to an existing split. Thus, they cannot influence any part of the base concern's control flow other than the execution of the split itself. Therefore, we will only discuss the effects of the *FreeConnector*.

Existing research [12] considers three kinds of workflows with respect to the structure of their control flow: *arbitrary workflows*, *structured workflows*, and *restricted loop workflows*. In general, the Unify base language allows expressing arbitrary workflows. However, a specific extension of the Unify meta-model may be more restrictive. Therefore, we restrict the *FreeConnector* depending on the kind of workflows that is supported by the current extension.

Arbitrary Workflows. Intuitively, arbitrary workflows are workflows in which a split does not need to have a corresponding join. There is no guarantee that every arbitrary workflow is well-behaved; deciding whether a given arbitrary workflow is well-behaved requires the use of verification techniques [12].

Using a free connector to connect a well-behaved arbitrary workflow concern to a well-behaved arbitrary base workflow concern may give rise to a composition that is not well-behaved, but it is not possible to prevent this without introducing structure into the arbitrary workflow concerns. Therefore, we do not introduce any restrictions on the free connector when it is used to connect arbitrary workflow concerns. However, the semantics of our connector mechanism (see Section 6.3) allows applying the same verification techniques that are used to decide whether the connected workflow concerns are well-behaved to the composition of these workflow concerns, and the connector mechanism thus does not introduce any new challenges in this regard.

Structured Workflows. Intuitively, structured workflows are workflows in which every AND-split has a corresponding AND-join, and every XOR-split has a corresponding XOR-join. Thus, each split and its corresponding join constitute a block structure, and control can only enter or exit this block structure through the join or split. Control can also not cross the different branches of the block structure. These restrictions guarantee that a structured workflow is well-behaved [12].

Because free connectors insert a new split and and a new join into a base workflow concern in order to execute another workflow concern as an additional branch, this new branch may make a structured base workflow concern unstructured. In order to prevent this, the splitting and joining control ports of a free connector must be part of the same branch of the same block structure in case of structured workflows. Note that this restriction precludes the use of the free connector that was introduced in Section 2 in order to synchronize two parallel branches: as the splitting and joining joinpoints of this free connector are located in *different* branches of the same block structure, this connector is disallowed when the current extension only supports structured workflows.

Restricted Loop Workflows. Intuitively, restricted loop workflows are workflows in which only loops need to be structured (i.e., a split must only have a corresponding join if it introduces a loop in the workflow). Thus, only loops constitute block structures. Restricted loop workflows are less expressive than arbitrary workflows and more expressive than structured workflows. Depending on the implementation of the underlying workflow engine, it can be guaranteed that restricted loop workflows are well-behaved [12]. Similar to our approach for structured workflows, we restrict the free connector in case of restricted loop workflows: the splitting and joining control ports of a free connector must be part of the same branch of the same block structure.

The above restrictions aim to *prevent* undesirable effects of using the Unify connector mechanism. One can also envision approaches that *detect* undesirable effects. For example, in previous work [14] we have designed and implemented a means of expressing and statically verifying control flow policies for Unify workflows.

6.2 Interaction with the Data Perspective

In addition to effects on the control flow perspective of the connected workflow concerns, the connector mechanism has effects on the data perspective of the connected workflow concerns. For example, a connected workflow concern may reference a variable that is not defined at the place where it is woven. The effects of the connector mechanism on the data perspective depend on the approach that is used by the specific extension to the Unify meta-model to pass data from one activity to another. Existing research [15] has identified the following approaches: *integrated control and data channels*, *distinct control and data channels*, and *no data passing*. We assume that both of the connected workflow concerns use the same approach.

Integrated Control and Data Channels. In this approach, control flow and data are passed simultaneously between activities, and transitions are annotated with which data elements must be passed. Activities can only access data that has been passed to them by an incoming transition. Given two workflow concerns that use this approach, we must make sure that the weaving of the two workflow concerns results in a correct composition with regard to the data elements. Therefore, a connector must specify which data elements should be passed from the base workflow concern to the other workflow concern and back. These data elements must be accessible at the joinpoint. The weaving process will generate transitions from the base workflow concern to the other workflow concern and back, and annotate these transitions with the data elements to be passed, resulting in a correct composition.

Distinct Control and Data Channels. In this approach, data is passed between activities via explicit data links that are *distinct* from control flow links (i.e., transitions). Therefore, a connector must specify which data elements should be passed from which activities in the base workflow concern to the other workflow concern and back. The weaving process will generate transitions between the workflow concerns as usual, but will also generate distinct data links from the specified activities in the base workflow concern to the other workflow concern and back, resulting in a correct composition.

No Data Passing. In this approach, activities share the same data elements, typically via access to some common scope. Thus, no explicit data passing is required. In order to implement our connector mechanism, we could merely weave a workflow concern into the base workflow conern without any regard to the data perspective. The activities of the former workflow concern could then access all the data elements that are accessible at the joinpoint. However, this would be undesirable as it would amount to dynamic scoping: a woven workflow concern might execute correctly at one joinpoint, and not at another, depending on which variables are accessible. Therefore, a connector must specify the data elements that are expected from the base workflow concern, and how these map to the data elements used in the other workflow concern. The weaving process can then verify whether all data is correctly mapped, and copy the data elements from the base workflow concen to the other workflow concern according to this mapping.

The solutions for each of these three approaches can be defined as extensions to the Unify meta-model. In the context of our extension towards WS-BPEL, we have already defined the extension for the *no data passing* approach. This extension encompassed associating every *CompositeActivity* with a *Scope*, which defines any number of *Variables*. This information can then be used by the weaving process.

6.3 Semantics

Because inversion-of-control connectors can invasively change the behavior of a concern by connecting another concern to it at an unanticipated location, it is important that the semantics of these connectors is clearly defined. Therefore, we have defined the semantics of the connector weaving using the *graph transformation* formalism [16]. In the interest of brevity, this section briefly discusses this semantics. We refer the reader to [17] for a complete description of our connector weaving semantics.

The semantics of a connector, which connects an workflow concern to a base workflow concern, is given by constructing a new concern that composes the base concern and the other concern according to the connector type and the pointcut specification. This is accomplished using *graph transformation rules* that work on the abstract syntax of the Unify base language.

A *graph* consists of a set of nodes and a set of edges. A *typed graph* is a graph in which each node and edge belong to a type defined in a *type graph*. An *attributed graph* is a graph in which each node and edge may contain attributes where each attribute is a $(value, type)$ pair giving the value of the attribute and its type. Types can be structured by an inheritance relation.

A *graph transformation rule* is a rule used to modify a host graph, G, and is defined by two graphs (L, R). L is the left-hand side of the rule representing the pre-conditions of the rule and R is the right-hand side representing the post-conditions of the rule. The process of applying the rule to a graph G involves finding a graph monomorphism, h, from L to G and replacing $h(L)$ in G with $h(R)$ (more details can be found in [16]).

In our approach, the type graph represents the meta-model shown in Figure 3. The translation of this meta-model to a type graph is straightforward: each meta-class corresponds to a typed node and each meta-association corresponds to a typed edge. Attributes in the meta-model are translated to corresponding node attributes. The well-formedness constraints can be formalized by graph constraints. Graph constraints allow the expression of properties over graphs (more details can be found in [18]).

For each possible combination of connector type (cf. Figure 4) and pointcut predicate (cf. Table 2), we specify a *composition rule*. Due to space restrictions we only explain the rules for the *FreeConnector*. The complete set of composition rules can be found in [17].

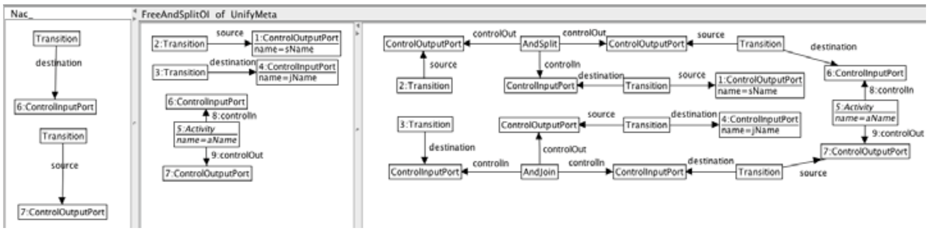


Fig. 5. The *FreeAndSplittingOI* graph transformation rule in AGG. The leftmost pane represents a NAC, which should be seen as a forbidden structure. The next pane represents the positive part of the rule's left-hand side. The rightmost pane represents the right-hand side of the rule.

Figure 5 shows the rule that corresponds to an AND-splitting *FreeConnector* expressed in the general-purpose graph transformation tool *AGG*.[2] The left-hand side of a graph transformation rule is composed of a positive condition, i.e., the presence of certain combinations of nodes and edges, and optionally, a set of negative application conditions (NACs), i.e., absence of certain combinations of nodes and edges. On the right-hand side of the transformation rule the result of weaving the workflow concern in the base workflow concern is shown. Remark that eight rules are specified for the *FreeConnector*: four rules for the AND-splitting *FreeConnector*, and four for the XOR-splitting *FreeConnector*. Each of these has four rules because of the possible combinations of control input and output ports.

The *FreeAndSplittingOI(sName : String, jName : String, aName : String)* rule adds a split at a certain control output port, adds a join at a certain control input port, and inserts an activity between the new split and join. The rule is parametrized with the name of the splitting control output port, the name of the joining control input port, and the name of the activity that is to be inserted. The left-hand side of the rule specifies the splitting *ControlOutputPort* and its outgoing *Transition*, the joining *ControlInputPort* and its incoming *Transition*, and the *Activity* that is to be inserted with its *ControlInputPort* and *ControlOutputPort*. The right-hand side specifies the graph after inserting the activity. The splitting *ControlOutputPort* is now connected to a new *AndSplit* through a new *Transition*. The *AndSplit* has two outgoing *Transitions*, the first is the splitting *ControlOutputPort*'s original outgoing *Transition*, and the second is a new *Transition* that is connected to the *ControlInputPort* of the inserted *Activity*. The *ControlOutputPort* of the inserted *Activity* is connected to a new *AndJoin*

[2] See `http://tfs.cs.tu-berlin.de/agg/`

through a new *Transition*. The other incoming *Transition* of the *AndJoin* is the joining *ControlInputPort*'s original incoming *Transition*. Finally, the *AndJoin* is connected to the joining *ControlInputPort* through a new *Transition*.

7 Implementation

We have created a proof-of-concept implementation for Unify, which is available for download at [11]. The architecture of this implementation is shown in Figure 6. At the heart of the architecture lies a Java implementation of the Unify base language (cf. Figure 3) and connector mechanism (cf. Figure 4). The Unify API allows constructing and manipulating workflow concerns in-memory, while extensions to the Unify framework provide parsers and serializers for existing concrete workflow languages. A composition specifies which concerns should be loaded and which connectors should be applied to them.

One by one, the Unify connector weaver applies the connectors to the base concern in the order specified by the composition. For each connector, the base concern is modified accordingly. The final modified base concern is transformed into a Petri nets execution model if one wants to use Unify's built-in workflow engine, or is exported back to the workflow language in which the original concerns were specified. In this latter case, the composition is serialized into a single workflow in which all concerns are woven together. An Eclipse plug-in that facilitates interaction with the Unify tool chain is currently under development.

The instantiation process is straightforward for common workflow concepts, i.e., activities and basic control flow concepts. However, three limitations may arise when instantiating Unify: (1) The base language is graph-based, which means that block-structured constructs such as those encountered in WS-BPEL should be correctly mapped to Unify's graph-based constructs, which is feasible. (2) The base language only provides the basic control flow patterns identified in existing research, which means that it is cumbersome to implement advanced control flow patterns such as those encountered in YAWL [19]. (3) The base

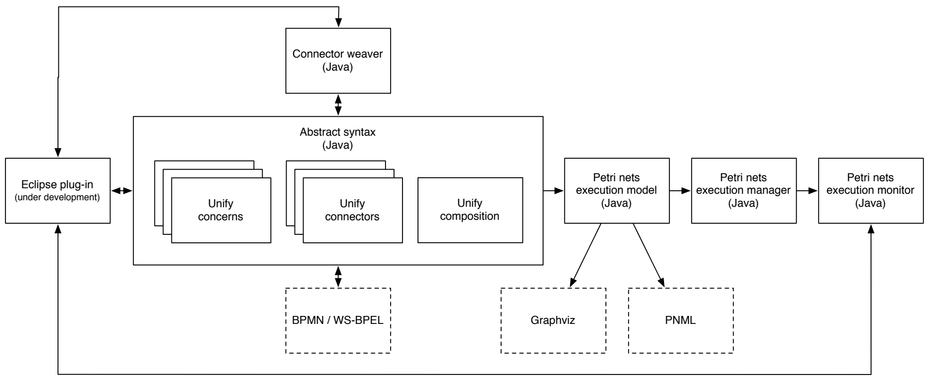

Connector weaver (Java)
Abstract syntax (Java)
Eclipse plug-in (under development)
Unify concerns
Unify connectors
Unify composition
Petri nets execution model (Java)
Petri nets execution manager (Java)
Petri nets execution monitor (Java)
BPMN / WS-BPEL
Graphviz
PNML

Fig. 6. Architecture of the Unify implementation

language focuses on the control flow perspective, which means that it provides no support for other perspectives, such as the exception handling perspective. These limitations are the result of the deliberate choice to focus on the expressiveness of the modularization mechanism rather than on the expressiveness of the individual modules, and could be addressed by iterating over the base language meta-model. We believe that the current meta-model is sufficient for demonstrating our contributions to the modularization of workflow concerns.

8 Related Work

Broadly speaking, the related work we consider can be divided into four domains:

Component Based Systems. Component based software development (CBSD) aims to promote separation of (non-crosscutting) concerns by allowing the composition of independent software components. Although some component frameworks allow modularizing specific crosscutting concerns using constructs like deployment descriptors, a general modularization mechanism for crosscutting concerns is typically unavailable. In the context of CBSD, connectors are often used to specify the roles that different software components fulfill in a composition [20]. Unify connectors are inspired by such component-based connectors, and are similarly used to specify, at deployment time, how different concerns should be composed. However, unlike component-based connectors, with respect to the connected concerns, Unify connectors describe both anticipated and non-anticipated (e.g., aspect-oriented) connections.

Traditional Workflow Languages. The most well-known current workflow languages are WS-BPEL [10] and YAWL [19]. WS-BPEL has notoriously poor support for separation of concerns (which has led to a number of aspect-oriented approaches that aim to remedy this; see below): a WS-BPEL process is a monolithic XML file that cannot be straightforwardly divided into sub-processes. YAWL improves on this in that it allows workflows to be divided into reusable sub-workflows.

Aspect-Oriented Programming for Workflows. The lack of modularization mechanisms in traditional workflow languages, most notably WS-BPEL, has led to the development of a number of aspect-oriented approaches for worfklows. AO4BPEL [6], the approach by Courbis & Finkelstein [7], and Padus [3] are the most well-known. They all allow modularizing crosscutting concerns in WS-BPEL workflows using separate aspects. Unify improves on them by allowing the modularization of *all* workflow concerns, i.e., not only crosscutting ones, and by introducing workflow-specific advice types in addition to the classic before, after, and around advice types. Moreover, Unify is not restricted to any concrete workflow language such as WS-BPEL.

Dynamic Workflow Systems. Existing research has produced a taxonomy of workflow flexibility [21]. With regard to this taxonomy, Unify mainly aims to improve workflow flexibility *by design* by providing a more expressive modularization mechanism than those offered by current workflow languages. The use of the standard Unify connector weaver precludes other forms of flexibility, i.e., flexibility by deviation, by underspecification, or by change (which describe different kinds of runtime adaptation of workflows). Extending our execution model with support for runtime enabling and disabling of connectors would remove this restriction, but is currently beyond the scope of our research, as runtime adaptation does not improve the design or reuse of workflow concerns. This constitutes an important difference in focus with regard to workflow systems that allow dynamically changing workflows.

9 Conclusions and Future Work

Existing workflow languages have insufficient support for separation of concerns. This can make workflows hard to comprehend, maintain, and reuse. We address this problem by introducing Unify, a framework that allows specifying both regular and crosscutting workflow concerns in isolation of each other. The Unify connector mechanism allows connecting these independently specified concerns using a number of workflow-specific connectors.

Activity connectors allow expressing that an existing activity in one concern should be implemented by executing another concern, in a way that minimizes dependencies between these concerns and thus facilitates their independent evolution and reuse. Additionally, inversion-of-control connectors allow augmenting a concern with other concerns that were not considered when it was designed, and again facilitates independent evolution and reuse of these concerns.

At the heart of Unify lies a meta-model that allows expressing arbitrary workflows, and which can be mapped to several concrete workflow languages and notations. We also provide a meta-model for the connector mechanism, and discuss its interaction with the control flow and data perspectives. We provide a semantics for the weaving of the connectors using the *graph transformation* formalism.

We have identified the following directions of future work: (1) Unify connectors are currently specified using a textual syntax (cf. Section 3). We are investigating how we can support workflow developers in specifying connectors in a visual way. (2) Although a proof-of-concept implementation of Unify has been developed, tool support should be extended in order to facilitate adoption of the approach. Therefore, we are developing an Eclipse plug-in that facilitates interaction with the Unify tool chain. (3) We are working on a more extensive validation of our approach, based on the refactoring of a real-life workflow application using Unify.

References

1. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Communications of the ACM 15(12), 1053–1058 (1972)
2. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Auletta, V. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
3. Braem, M., Verlaenen, K., Joncheere, N., Vanderperren, W., Van Der Straeten, R., Truyen, E., Joosen, W., Jonckers, V.: Isolating Process-Level Concerns Using Padus. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) BPM 2006. LNCS, vol. 4102, pp. 113–128. Springer, Heidelberg (2006)

4. Joncheere, N., Deridder, D., Van Der Straeten, R., Jonckers, V.: A Framework for Advanced Modularization and Data Flow in Workflow Systems. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 592–598. Springer, Heidelberg (2008)
5. Arsanjani, A., Hailpern, B., Martin, J., Tarr, P.: Web services: Promises and compromises. ACM Queue 1(1), 48–58 (2003)
6. Charfi, A., Awasthi, P.: Aspect-Oriented Web Service Composition with AO4BPEL. In: Zhang, L.-J., Jeckle, M. (eds.) ECOWS 2004. LNCS, vol. 3250, pp. 168–182. Springer, Heidelberg (2004)
7. Courbis, C., Finkelstein, A.: Towards aspect weaving applications. In: Proceedings of the 27th International Conference on Software Engineering (ICSE 2005), pp. 69–77. ACM Press, St. Louis (2005)
8. González, O., Casallas, R., Deridder, D.: MMC-BPM: A Domain-Specific Language for Business Processes Analysis. In: Abramowicz, W. (ed.) Business Information Systems. Lecture Notes in Business Information Processing, vol. 21, pp. 157–168. Springer, Heidelberg (2009)
9. Object Management Group: Business Process Model and Notation, version 2.0 (2011), http://www.omg.org/spec/BPMN/2.0/
10. Jordan, D., Evdemon, J., et al.: Web Services Business Process Execution Language, version 2.0 (2007), http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
11. Joncheere, N., et al.: The Unify framework (2009), http://soft.vub.ac.be/~njonchee/artifacts/unify/
12. Kiepuszewski, B., ter Hofstede, A.H.M., Bussler, C.J.: On Structured Workflow Modelling. In: Wangler, B., Bergman, L.D. (eds.) CAiSE 2000. LNCS, vol. 1789, pp. 431–445. Springer, Heidelberg (2000)
13. Russell, N., ter Hofstede, A.H.M., van der Aalst, W.M.P., Mulyar, N.: Workflow control -flow patterns: A revised view. BPM Center Report BPM-06-22, BPM Center (2006)
14. De Fraine, B., Joncheere, N., Noguera, C.: Detection and resolution of aspect interactions in workflows. Technical Report SOFT-TR-2011.06.20, Vrije Universiteit Brussel, Software Languages Lab, Brussels, Belgium (2011), http://soft.vub.ac.be/~njonchee/publications/TR20110620.pdf
15. Russell, N., ter Hofstede, A.H.M., Edmond, D., van der Aalst, W.M.P.: Workflow data patterns. QUT Technical Report FIT-TR-2004-01, Queensland University of Technology (2004)
16. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation, Foundations, vol. 1. World Scientific, River Edge (1997)
17. Joncheere, N., Van Der Straeten, R.: Semantics of the Unify composition mechanism. Technical Report SOFT-TR-2011.04.15, Vrije Universiteit Brussel, Software Languages Lab, Brussels, Belgium (2011), http://soft.vub.ac.be/~njonchee/publications/TR20110415.pdf
18. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of algebraic graph transformation. Monographs in Theoretical Computer Science. Springer (2006)
19. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: Yet Another Workflow Language. Information Systems 30(4), 245–275 (2005)
20. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, Upper Saddle River (1996)
21. Schonenberg, H., Mans, R., Russell, N., Mulyar, N., van der Aalst, W.M.P.: Process flexibility: A survey of contemporary approaches. LNBIP, vol. 10, pp. 16–30 (2008)

Design of Concept Libraries for C++

Andrew Sutton and Bjarne Stroustrup

Texas A&M University
Department of Computer Science and Engineering
{asutton,bs}@cse.tamu.edu

Abstract. We present a set of concepts (requirements on template arguments) for a large subset of the ISO C++ standard library. The goal of our work is twofold: to identify a minimal and useful set of concepts required to constrain the library's generic algorithms and data structures and to gain insights into how best to support such concepts within C++. We start with the design of concepts rather than the design of supporting language features; the language design must be made to fit the concepts, rather than the other way around. A direct result of the experiment is the realization that to simply and elegantly support generic programming we need two kinds of abstractions: *constraints* are predicates on static properties of a type, and *concepts* are abstract specifications of an algorithm's syntactic and semantic requirements. Constraints are necessary building blocks of concepts. Semantic properties are represented as axioms. We summarize our approach: $concepts = constraints + axioms$. This insight is leveraged to develop a library containing only 14 concepts that encompassing the functional, iterator, and algorithm components of the C++ Standard Library (the STL). The concepts are implemented as constraint classes and evaluated using Clang's and GCC's Standard Library test suites.

Keywords: Generic programming, concepts, constraints, axioms, C++.

1 Introduction

Concepts (requirements on template arguments) are the central feature of C++ generic library design; they define the terms in which a library's generic data structures and algorithms are specified. Every working generic library is based on concepts. These concepts may be represented using specifically designed language features (e.g. [15, 20]), in requirements tables (e.g., [2,22]), as comments in the code, in design documents (e.g., [21]), or simply in the heads of programmers. However, without concepts (formal or informal), no generic code could work.

For example, a Matrix library that allows a user to supply element types must have a concept of what operations can be used on its elements. In other words, the Matrix library has a concept of a "number," which has a de facto definition in terms of operations used in the library code. Interpretations of "number" can vary dramatically from library to library. Is a polynomial function a number'? Are numbers supposed to support division? Are operations on numbers supposed to be associative? Regardless, every generic Matrix library has a "number" concept.

In this work, we experiment with designs for concepts by trying to minimize the number of concepts required to constrain the generic algorithms and data structures of a library. To do such experiments, we need ways to represent concepts in C++ code. Our approach is to use constraint classes, which allows us to experiment with different sets of concepts for a library without pre-judging the needs for specific language support features. Once we have experimented with the design of concepts for several libraries, we expect our criteria for concept design and use to have matured to the point where we can confidently select among the reasonably well-understood language design alternatives. Language support should be determined by ideal usage patterns rather than the other way around.

Previous work in the development of concept libraries has resulted in the definition of large numbers of concepts to constrain comparatively few generic components. For example, the last version of the C++ Draft Standard to include concepts [5] defined 130 concepts to constrain approximately 250 data structures and algorithms. Nearly 2/3rds of that library's exported abstractions were concept definitions. Out of the 130 concepts, 108 relate are used to (directly or indirectly) express the functional, iterator, and algorithms parts of the standard library that we address here. In comparison, the Elements of Programming (EoP) book [34] covers similar material with 52 concepts for about 180 different algorithms (just under 1/3rd).

Obviously, not all of the concepts in either publication are "equally abstract". That C++ Draft Standard includes about 30 concepts enumerating requirements on overloadable operator's alone and several metaprogramming-like concepts such as **Rvalue_of**. The EoP book lists many concepts that explore variations on a theme; invariants are strengthened or weakened, operators are dropped, and definition spaces are restricted. From these and similar designs of concepts, it has been concluded that writing a generic library requires exhaustive enumeration of overloadable operations, conceptual support for template metaprogramming, and massive direct language support for concepts.

Fortunately, such conclusions are false. Furthermore, they are obviously false. The overwhelming majority of generic libraries, including the original version of STL, were written without language support for concepts and used without enforcement mechanisms. The authors of these libraries did not consider some hundred different concepts during their development. Rather, the generic components of these libraries were written to a small set of idealized abstractions.

We argue that generic libraries are defined in terms of small sets of abstract and intuitive concepts, and that an effective specification of concepts is the product of an iterative process that minimizes the number of concepts while maintaining expressive and effective constraints. Ideal concepts are fundamental to their application domain (e.g., *String*, *Number*, and *Iterator*), and consequently it is a rare achievement to find a genuine new one. These abstract and intuitive concepts must not be lost in the details needed to express them in code. Neither should the ease of learning and ease of use offered by these concepts be compromised by an effort to minimize the constraints of every algorithm.

To explore these ideas concretely, we develop a minimal design of the concepts for the STL that encompasses its functional, iterator, and algorithm libraries. The resulting concept library defines only 14 concepts, 94 fewer than originally proposed for the corresponding parts of the C++ Standard Library. We leverage the results of the experiment and the experience gained to derive a novel perspective on concepts for the C++ programming language. Our approach reveals a distinct difference between requirements that represent domain abstractions and those that support their specification: *concepts* and *constraints*, respectively. Concepts based on this distinction results in a conceptually simpler and more modular design.

We validate the design of the concepts by implementing them within the framework of a concept emulation library for the Origin C++ Libraries [38]. The libraries are implemented using the 2011 ISO C++ standard facilities. The concepts are then integrated into the STL subset of Clang's libc++ and GCC's libstdc++ and checked against their test suites for validation.

The results of this work yield several contributions to our knowledge of generic programming. First, we demonstrate that it is both feasible and desirable to experiment to seek a minimal conceptual specification for a generic library. Second, we demonstrate that distinguishing between concepts (as abstractions) and constraints (as static requirements) is an effective way to achieve this goal. Third, we identify semantic properties, axioms, as the key to practical discrimination of concepts from constraints. Finally, we provide a simpler and easier to use specification of key parts of the ISO C++ standard library.

2 Related Work

Generic programming is rooted in the ability to specify code that will work with a variety of types. Every language supporting generic programming must address the issue of how to specify the interface between generically written code and the set of concrete types on which it operates. Several comparative studies have been conducted in support for generic programming and type constraints [18]. These studies were leveraged to support the definition of concepts for C++

ML relies on *signatures* to specify the interface of modules and constrain the type parameters of functors [27]. Operations in the signature are matched against those defined by a structure to determine conformance. In Haskell, *type classes* denote sets of types that can be used with same operations [24]. A type is made a member of that set by explicitly declaring it an *instance* and implementing the requisite operations. Type classes are used as constraints on type parameters in the signatures of polymorphic functions. Similarities between Haskell type classes and C++ concepts have also been explored [6,7]. AXIOM *categories* are used to define the syntax and semantics of algebraic domains [10]. The specification of algebraic structures helped motivate the design of its category system [9,11]. Requirements on type parameters in Eiffel, Java, and C# are specified in terms of inherited interfaces. Checking the conformance of a supplied type argument entails determining if it is a subtype of the required interface or class [8,25,26,41].

From the earliest days of the design of C++ templates, people have been looking for ways to specify and constrain template arguments [35,36]. For the C++0x standards effort two proposals (with many variants) were considered [15,20]. A language framework supporting generic programming was developed in support of these proposals [32,33].

There has been less work on the design of concepts themselves (as opposed to studying language support). The dominant philosophy of concept design has focused on "lifting" algorithms from specific implementations to generic algorithms with specific requirements on arguments [28]. However, applying the same process to the concepts (iterative generalization) can lead to an explosion in the number of concepts as requirements are minimized for each algorithm in isolation. The (ultimately unsuccessful) design for C++0x included a carefully crafted and tested design for the complete C++0x standard library including around 130 concepts [5]. In [13], Dehnert and Stepanov defined properties of regular types and functions. Stepanov and McJones carefully worked out a set of concepts for their variant of the STL in EoP [34]; Dos Reis implemented verification for those and found a few improvements [14].

Other research has focused on the use of concepts or their expression in source code. Bagge and Haveraaen explored the use of axioms to support testing and the semantic specification of algebraic concepts [4]. Pirkelbauer et al [29] and Sutton and Maletic [40] studied concepts through mechanisms for automatically extracting requirements from actual source code. Also, many aspects of concepts can be realized idiomatically in C++0x [12,39]; this is the basis of our implementation in this work.

3 Requirements for Concept Design

The immediate and long-term goals of this research are to develop an understanding of the principles of concepts and to formulate practical guidelines for their design. A midterm goal is to apply that understanding to the design of language features to support the use of concepts, especially in C++.

Here, we present an experiment in which we seek a conceptual specification for the STL that defines a minimum number of concepts. This goal is in stark contrast to previous work [5] where the explicit representation of even the subtlest distinctions in requirements was the ideal.

The minimization aspect of the experiment constrains the design in such a way that the resulting concepts must be abstract and expressive. A concept represents a generic algorithm's requirements on a type such as a *Number* or an *Iterator*. A concept is a predicate that can be applied to a type to ascertain if it meets the requirements embodied by the concept. An expressive concept, such as Iterator, allows a broad range of related expressions on a variety of types. In contrast, a simple syntactic requirement, such as requiring default construction, is a constraint on implementers and does not express a general concept.

The effect of this minimization is ease of learning and ease use. In particular, it provides library designers and users with a simple, strong design guideline that could never be achieved with (say) 100 primitive requirements (primitive in the mathematical sense). The design of a concept library is the result of two minimization problems: concept and constraint minimization.

Concept minimization seeks to find the smallest set of concepts that adequately represent the abstractions of a generic library. The problem is na?vely solved by defining a single concept that satisfies the requirements of all templates. For example, mutable random-access iterators work with practically all STL algorithms so a minimum concept specification might simply be a **Random_access_iterator**. This would result in over-constrained templates and make many real-world uses infeasible. For example, a linked-list cannot easily and economically support random access and an input stream cannot support random access at all. Conversely, the popular object-oriented notion of a universal *Object* type under-constrains interfaces, so that programmers have to rely on runtime resolution, runtime error handling, and explicit type conversion.

Constraint minimization seeks to find a set of constraints that minimally constrain the template arguments. This problem is na?vely solved by naming the minimal set of type requirements for the implementation of an algorithm. If two algorithms have non-identical but overlapping sets of requirements, we factor out the common parts, which results in three logically minimal sets of requirements. This process is repeated for all algorithms in a library. Concepts developed in this manner resemble the syntactic constructs from which they are derived; the number of concepts is equal to the number of uniquely typed expressions in a set of the algorithms. This results in the total absence of abstraction and large numbers of irregular, non-intuitive concepts. In the extreme, every implementation of every algorithm needs its own concept, thus negating the purpose of "concepts" by making them nothing but a restatement of the requirements of a particular piece of code.

Effective concept design solves both problems through a process of iterative refinement. An initial, minimal set of concepts is defined. Templates are analyzed for requirements. If the initial set of concepts produces overly strict constraints, the concept definitions must be refactored to weaken the constraints on the template's arguments. However, the concepts must not be refactored to the extent that they no longer represent intuitive and potentially stable abstractions.

4 Concepts = Constraints + Axioms

Previous work on concepts use a single language mechanism to support both the abstract interfaces (represented to users as concepts) and the queries about type properties needed to eliminate redundancy in concept implementations. From a language-technical point of view, that makes sense, but it obscures the fundamental distinction between interface and implementation of requirements. Worse, it discourages the programmer from making this distinction by not providing language to express the distinction. We conclude that we need separate mechanisms for the definition of concepts and for their implementation.

We consider three forms of template requirements in our design:

- *Constraints* define the statically evaluable predicates on the properties and syntax of types, but do not represent cohesive abstractions.
- *Axioms* state semantic requirements on types that should not be statically evaluated. An axiom is an invariant that is assumed to hold (as opposed required to be checked) for types that meet a concept.
- *Concepts* are predicates that represent general, abstract, and stable requirements of generic algorithms on their argument. They are defined in terms of constraints and axioms.

Constraints are closely related to the notion of type traits, metafunctions that evaluate the properties of types. We choose the term "constraint" over "trait" because of the varied semantics already associated the word “trait.” The word “constraint” also emphasizes the checkable nature of the specifications. The terms “concept” and “axiom” are well established in the C++ community [1,2,5,16,18,22,34,39].

Constraints and axioms are the building blocks of concepts. Constraints can be used to statically query the properties, interfaces, and relationships of types and have direct analogs in the Standard Library as type traits (e.g., **is_const**, **is_constructible**, **is_same**). In fact, a majority of the concepts in C++0x represent constraints [5].

Axioms specify the meaning of those interfaces and relationships, type invariants, and complexity guarantees. Previous representations of concepts in C++ define axioms as features expressed within concepts [5,16]. In our model, we allow axioms to be written outside of concept specifications, not unlike *properties* in the EoP book [34]. This allows us to distinguish between semantics that are inherent to the meaning of a concept and those that can be stated as assumed by a particular algorithm. The distinction also allows us to recombine semantic properties of concepts without generating lattices of semantically orthogonal concepts, which results in designs with (potentially far) fewer concepts.

The distinctive property that separates a concept from a constraint is that it has semantic properties. In other words, we can write axioms for a concept, but doing so for a constraint would be farfetched. This distinction is (obviously) semantic so it is possible to be uncertain about the classification of a predicate, but we find that after a while the classification becomes clear to domain experts. In several cases, the effort to classify deepened our understanding of the abstraction and in four cases the “can we state an axiom?” criterion changed the classification of a predicate, yielding—in retrospect—a better design. These improvements based on the use of axioms were in addition to the dramatic simplifications we had achieved using our earlier (less precise) criteria of abstractness and generality.

Our decision to differentiate concepts and constraints was not made lightly, nor was the decision to allow axioms to be decoupled from concepts. These decisions are the result of iteratively refining, balancing, and tuning concepts for the STL subject to the constraints of the experiment. These insights, once made, have resulted in a clear, concise, and remarkably consistent view of the abstractions in the STL. The distinction between concepts, constraints, and axioms is a valuable new design tool that supports modularity and reuse in conceptual specifications. We expect the distinction to have implications on the design of the language support for concepts. For example, if we need explicit modeling statements (concept maps [20]; which is by no means certain), they would only be needed for concepts. Conversely, many constraints are compiler intrinsic [22]. These two differences allow for simpler compilation model and improved compile times compared to designs based on a single language construct [5,19].

As an example of the difference, consider the C++0x concepts **HasPlus**, **HasMinus**, **HasMultiply**, and **HasDivide**. By our definition, these are not concepts. They are nothing but requirements that a type has a binary operator **+**, **-**, *****, and, **/**, respectively. No meaning (semantics) can be ascribed to each in isolation. No algorithm could be written based solely on the requirement of an argument type providing (say) - and *. In contrast, we can define a concept that requires a combination of those (say, all of them) with the usual semantics of arithmetic operations conventionally and precisely stated as axioms. Such concepts are the basis for most numeric algorithms.

It is fairly common for constraints to require just a single operation and for a concept to require several constraints with defined semantic relationships among them. However, it is quite feasible to have a single-operation concept and, conversely, a multi-operation constraint. For example, consider the interface to balancing operations from a framework for balanced-binary tree implementations [3]:

```
constraint Balancer<typename Node> {
   void add_fixup(Node*);
   void touch(Node*);
   void detach(Node*);
}
```

This specifies the three functions that a balancer needs to supply to be used by the framework. It is clearly an internal interface of little generality or abstraction; it is an implementation detail. If we tried hard enough, we might come up with some semantic specification (which would depend on the semantics of **Node**), but it would be unlikely to be of use outside this particular framework (where it is used in exactly one place). Furthermore, it would be most unlikely to survive a major revision and extension of the framework unchanged. In other words, the lack of generality and the difficulty of specifying semantic rules are strong indications that Balancer is not a general concept, so we make it a constraint. It is conceivable that in the future we will understand the application domain well enough to support a stable and formally defined notion of a **Balancer** and then (and only then) would we promote **Balancer** to a concept by adding the necessary axioms. Partly, the distinction between concept and constraint is one of maturity of application domain.

Constraints also help a concept design accommodate *irregularity*. An irregular type is one that almost meets requirements but deviates so that a concept cannot be written to express a uniform abstraction that incorporates the irregular type. For example, **ostream_iterator** and **vector<bool>::iterator** are irregular in that their value type cannot be deduced from their reference type. Expression templates are paragons of irregularity: they encode fragments of an abstract syntax tree as types and can support lazy evaluation without additional syntax [42]. We can't support every such irregularity without creating a mess of "concepts" that lack proper semantic specification and are not stable (because they essentially represent implementation details). However, such irregular iterators and expression templates are viable type arguments to many STL algorithms. Constraints can be used to hide these irregularities, thus simplifying the specification of concepts. A long-term solution will have to involve cleaner (probably more constrained) specification of algorithms.

5 Concepts for the STL

Our concept design for the STL is comprised of only 14 concepts, 17 supporting constraints, and 4 independent axioms. These are summarized in Table 1.

We present the concepts, constraints, and axioms in the library using syntax similar to that developed for C++0x [5, 15, 20]. The syntax used in this presentation can be

Table 1. Concepts, constraints, and axioms

Concepts		***Constraints***	
Regularity	*Iterators*	*Operators*	*Language*
Comparable	**Iterator**	**Equal**	**Same**
Ordered	**Forward_iterator**	**Less**	**Common**
Copyable	**Bidirectional_iterator**	**Logical_and**	**Derived**
Movable	**Random_access_iterator**	**Logical_or**	**Convertible**
Regular		**Logical_not**	**Signed_int**
		Callable	
Functional	*Types*	*Initialization*	*Other*
Function	**Boolean**	**Destructible**	**Procedure**
Operation		**Constructible**	**Input_iterator**
Predicate		**Assignable**	**Output_iterator**
Relation			
Axioms			
Equivalence_relation			
Strict_weak_order			
Strict_total_order			
Boolean_algebra			

mapped directly onto the implementation, which supports our validation method. Also, the concept and constraint names are written as we think they should appear when used with language support, not as they appear in the Origin library.

To distinguish C++ from purely object-oriented or purely functional type systems, we preface the presentation of these concepts with a summary view of values and objects within the context of the C++ type system. In brief, a *value* is an abstract, immutable element such as the number 5 or the color red. An *object* resides in a specific area of memory (has identity) and may hold a value. In C++, values are represented by *rvalues*: literals, temporaries, and constant expressions (**constexpr** values). Objects are *lvalues* that support mutability in the forms of assignment and move (i.e., variables). Objects are uniquely identified by their address. A *constant* of the form **const T** is an object that behaves like a value in that it is immutable, although it still has identity. A *reference* is an alias to an underlying value (rvalue reference) or object (lvalue reference). In order for our design to be considered viable, it must address the differences between the various kinds of types in the C++ type system [37].

5.1 Regular Types

In our design, all abstract types are rooted in the notion of *regularity*. The concept Regular appears in some form or other in all formalizations of C++ types [22, 32, 34]; it expresses the notion that an object is fundamentally well behaved, e.g. it can be constructed, destroyed, copied, and compared to other objects of its type. Also, *Regular* types can be used to define objects. Regularity is the foundation of the value-oriented semantics used in the STL, and is rooted in four notions: *Comparability*, *Order*, *Movability*, and *Copyability*. Their representation as concepts follow.

```
concept Comparable<typename T> {
    requires constraint Equal<T>; // syntax of equality
    requires axiom Equivalence_relation<equal<T>, T>; // semantics of equivalence

    template<Predicate P>
    axiom Equality(T x, T y, P p) {
        x==y => p(x)==p(y); // if x==y then for any Predicate p, p(x) == p(y)
    }
    axiom Inequality(T x, T y) {
        (x!=y) == !(x==y); // inequality is the negation of equality
    }
}
```

The **Comparable** concept defines the notion of equality comparison for its type argument. It requires an operator **==** via the constraint **Equal**, and the meaning of that operator is imposed by the axiom **Equivalence_relation**. The **Equality** axiom defines the actual meaning of equality, namely that two values are equal if, for any **Predicate**, the result of its application is equal. We use the C++0x axiom syntax with the**=>** (implies) operator added [16]. The **Inequality** axiom connects the meaning of equality to inequality. If a type defines **==** but does not a corresponding **!=**, we can automatically generate a canonical definition according to this axiom (as described in Sect. 6). The **Inequality** axiom requires that user-defined **!=** operators provide the correct semantics.

We define the notion of *Order* similarly:

```
concept Ordered<Regular T> {
    requires constraint Less<T>;
    requires axiom Strict_total_order<less<T>, T>;
    requires axiom Greater<T>;
    requires axiom Less_equal<T>;
    requires axiom Greater_equal<T>;
}
```

We factor out the axioms just to show that we can, and because they are examples of axioms that might find multiple uses:

```
template<typename T>
axiom Greater(T x, T y) {
    (x>y) == (y<x);
}
template<typename T>
axiom Less_equal(T x, T y) {
    (x<=y) == !(y<x);
}
template<typename T>
axiom Greater_equal(T x, T y) {
    (x>=y) == !(x<y);
}
```

As with **Comparable**, the definition of requirements is predicated on a syntactic constraint (Less) and a semantic requirement (**Strict_total_order**). Obviously, not all types inherently define a total order; IEEE 754 floating point values define only a partial order when considering NaN values. Because this is an axiom and can't be proven by a C++ compiler, we are allowed to assume that it holds. The required axioms connect the meaning of the other relational operations to **<**.

```
concept Copyable<Comparable T> {
   requires constraint Destructible<T> && Constructible<T, const T&>

   axiom Copy_equality(T x, T y) {
      x==y => T{x}==y && x==y; // copy construction copies (non-destructively)
   }
};
```

A **Copyable** type is both copy constructible and **Comparable**. The **Copy_equality** axiom states that a copy of an object is equal to its original. **Copyable** (and also **Movable**) types must be **Destructible**, ensuring that the program can destroy the constructed objects.

```
concept Movable<typename T> {
   requires constraint Destructible<T> && Constructible<T, T&&>

   axiom Move_effect(T x, T y) {
      x==y => T{move(x)}==y && can_destroy(x); // original is valid but unspecified
   }
}
```

A **Movable** type is move constructible. Moving an object puts the moved-from object in a valid but unspecified state. The C++0x axiom syntax provides no way of expressing "valid but unspecified" so we introduce the primitive predicate **can_destroy()** to express that requirement.

A *Regular* type can be used to create objects, declare variables, make copies, move objects, compare values, and default-construct. In essence, the notion of regularity defines the basic set operations and guarantees that should be available for all value-oriented types.

```
concept Regular<typename T> {
   requires Movable<T> && Copyable<T>;
   requires constraint Constructible<T> // default construction
                    && Assignable<T, T&&> // move assignment
                    && Assignable<T, const T&>; // copy assignment

   axiom Object_equality(T& x, T& y) {
      &x==&y => x==y; // identical objects have equal values
   }
   axiom Move_assign_effect(T x, T y, T& z) {
      x==y => (z=move(x))==y && can_destroy(x); // original is valid but unspecified
   }
```

```
    axiom Copy_assign_equality(T x, T& y) {
        (y = x) == x; // a copy is equal to the original
    }
}
```

The **Object_equality** axiom requires equality for identical objects (those having the same address). The **Move_assign_effect** and **Copy_assign_equality** axioms extend the semantics of move and copy construction to assignment. Note that the requirement of assignability implies that **const**-qualified types are not **Regular**. Furthermore, **volatile**-qualified types are not regular because they cannot satisfy the **Object_equality** axiom; the value of a **volatile** object may change unexpectedly. These design decisions are intentional. Objects can be **const**- or **volatile**-qualified to denote immutability or volatility, but that does not make their *value's* type immutable or volatile. Also, including assignment as a requirement for Regular types allows a greater degree of freedom for algorithm implementers. Not including assignability would mean that an algorithm using temporary storage (e.g., a variable) would be required state the additional requirement as part of its interface, leaking implementation details through the user interface.

We note that *Order* is not a requirement of *Regular* types. Although many regular types do admit a natural order, others do not (e.g., **complex<T>** and **thread**), hence the two concepts are distinct.

This definition is similar to those found in previous work by Dehnert and Stepanov [13] and also by Stepanov and McJones [34] except that the design is made modular in order to accommodate a broader range of fundamental notions. In particular, these basic concepts can be reused to define concepts expressing the requirements of **Value** and **Resource** types, both of which are closely related to the **Regular** types, but have restricted semantics. A **Value** represents pure (immutable) values in a program such as temporaries or constant expressions. **Value**s can be copy and move constructed, and compared, but not modified. A **Resource** is an object with limited availability such an **fstream**, or a **unique_ptr**. **Resource**s can be moved and compared, but not copied. Both **Value**s and **Resource**s may also be **Ordered**. We omit specific definitions of these concepts because they were not explicitly required by any templates in our survey; we only found requirements for **Regular** types.

By requiring essentially all types to be **Regular**, we greatly simplify interfaces and give a clear guideline to implementers of types. For example, a type that cannot be copied is unacceptable for our algorithms and so is a type with a copy operator that doesn't actually copy. Other design philosophies are possible; the STL's notion of type is value-oriented; an objected-oriented set of concepts would probably not have these definitions of copying and equality as part of their basis.

5.2 Type Abstractions

There are a small number of fundamental abstractions found in virtually all programs: *Boolean*, *Integral*, and *Real* types. In this section, we describe how we might define concepts for such abstractions. We save specific definitions for future work, pending further investigation and experimentation.

The STL traditionally relies on the bool type rather than a more abstract notion, but deriving a Boolean concept is straightforward. The *Boolean* concept describes a generalization of the **bool** type and its operations, including the ability to evaluate objects in Boolean evaluation contexts (e.g., an **if** statement). More precisely, a *Boolean* type is a *Regular*, Ordered type that can be operated on by logical operators as well as constructed over and converted to **bool** values. The **Boolean** concept would require constraints for logical operators (e.g., **Logical_and**) and be defined by the semantics of the **Boolean_algebra** axiom.

Other type abstractions can be found in the STL's numeric library, which is comprised of only six algorithms. Although we did not explicitly consider concepts in the numeric domain, we can speculate about the existence and definition of some concepts. A principle abstraction in the numeric domain is the concept of *Arithmetic* types, those that can be operated on by the arithmetic operators **+**, *****, **-**, and **/** with the usual semantics, which we suspect should be characterized as an *Integral Domain*. As such, all integers, real numbers, rational numbers, and complex numbers are Arithmetic types. Stronger definitions are possible; an *Integral* type is an *Arithmetic* type that also satisfies the semantical requirements of a *Euclidean Domain*. We note that *Matrices* are not *Arithmetic* because of non-commutative multiplication.

The semantics of these concepts can be defined as axioms on those types and their operators. Concepts describing *Groups*, *Rings*, and *Fields* for these types should be analogous to the definition of **Boolean_algebra** for **Boolean** types. We leave the exact specifications of these concepts as future work as a broader investigation of numeric algorithms and type representations is required.

5.3 Function Abstractions

Functions and function objects (functors) are only "almost regular" so we cannot define them in terms of the **Regular** concept. Unlike regular types, functions are not default constructible and function objects practically never have equality defined. Many solutions have been suggested such as a concept **Semiregular** or implicitly adding the missing operations to functions and function objects.

To complicate matters, functions have a second dimension of regularity determined by *application equality*, which states that a function applied to equal arguments yields equal results. This does not require functions to be pure (having no side effects), only that any side effects do not affect subsequent evaluations of the function on equal arguments. A third property used in the classification of functions and function objects is the homogeneity of argument types. We can define a number of mathematical properties for functions when the types of their arguments are the same.

To resolve these design problems, we define two static constraints for building functional abstractions: *Callable* and *Procedure*. The *Callable* constraint determines whether or not a type can be invoked as a function over a sequence of argument types. The *Procedure* constraint establishes the basic type requirements for all procedural and functional abstractions.

```
constraint Procedure<typename F, typename... Args> {
  requires constraint Constructible<F, F const&> // has copy construction
                   && Callable<F, Args...>; // can be called with Args...
  typename result_type = result_of<F(Args...)>::type;
}
```

Procedure types are both copy constructible and callable over a sequence of arguments. The variadic definition of this concept allows us to write a single constraint for functions of any *arity*. The **result_type** (the *codomain*) is deduced using **result_of**.

There are no restrictions on the argument types, result types or semantics of **Procedures**; they may, for example, modify non-local state. Consequently, we cannot specify meaningful semantics for all procedures, and so it is static constraint rather than a concept. A (random number) *Generator* is an example of a Procedure that takes no arguments and (probably) returns different values each time it is invoked.

A *Function* is a Procedure that returns a value and guarantees application equivalence and deterministic behavior. It is defined as:

```
concept Function<typename F, typename... Args> : Procedure<F, Args...> {
  requires constraint !Same<result_type, void>; // must return a value

  axiom Application_equality(F f, tuple<Args...> a, tuple<Args...> b) {
    (a==b) => (f(a...)==f(b...)); // equal arguments yield equal results
  }
}
```

The **Application_equality** axiom guarantees that functions called on equal arguments yield equal results. This behavior is also invariant over time, guaranteeing that the function always returns the same value for any equal arguments. The syntax **a...** denotes the expansion of a tuple into function arguments. The hash function parameter of unordered containers is an example of a **Function** requirement.

An *Operation* is a *Function* whose argument types are homogeneous and whose domain is the same as its codomain (result type):

```
concept Operation<typename F, Common... Args> : Function<F, Args...> {
   requires sizeof...(Args) != 0; // F must take at least one argument
   typename domain_type = common_type<Args?>::type;
   requires constraint Convertible<result_type, domain_type>;
}
```

From this definition, an **Operation** is a **Function** accepting a non-empty sequence of arguments that share a **Common** type (defined in Sect 6). The common type of the function's argument types is called the **domain_type**. The **result_type** (inherited indirectly from **Procedure**) must be convertible to the **domain_type**.

Note that the concept's requirements are not applied directly to **F** and its argument types. Instead, the concept defines a functional abstraction over **F** and the unified domain type. Semantics for **Operation**s are more easily defined when the domain and result type are interoperable. This allows us to use the **Operation** concept as the basis of algebraic concepts, which we have omitted in this design. We note that many of the function objects in the STL's functional library generate **Operation** models (e.g., **less**, and **logical_and**).

Parallel to the three functional abstractions *Procedure*, *Function*, and *Operation*, we define two predicate abstractions: *Predicate* and *Relation*. A *Predicate* is a *Function* whose result type can be converted to bool. **Predicate**s are required by a number of STL algorithms; for example, any algorithm ending in **_if** requires a **Predicate** (e.g., **find_-**

if). This specification also matches the commonly accepted mathematical definition of a predicate and is a fundamental building block for other abstractions functional abstractions. A *Relation* is a binary *Predicate*. The axioms **Strict_weak_order** and **Strict_total_order** are semantic requirements on the **Relation** concept. These concepts are easily defined, and their semantics are well known.

5.4 Iterators

The distinction between concept and constraint has a substantial impact on the traditional iterator hierarchy [22]. We introduce a new general *Iterator* concept as the base of the iterator hierarchy. The input and output aspects of the previous hierarchy are relegated to constraints.

An *Iterator* is a *Regular* type that describes an abstraction that can "move" in a single direction using **++** (both pre- and post-increment) and whose referenced value can be accessed using unary *****. The concept places no semantic requirements on either the traversal operations or the dereferencing operation. In this way, the *Iterator* concept is not dissimilar from the ITERATOR design pattern [17], except syntactically. As with the previous designs, there are a number of associated types. Its definition follows:

```
concept Iterator<Regular Iter> {
    typename value_type = iterator_traits<Iter>::value_type;
    Movable reference = iterator_traits<Iter>::reference;
    Signed_int difference_type = iterator_traits<Iter>::difference_type;
    typename iterator_category = iterator_traits<Iter>::iterator_category;

    Iter& Iter::operator++(); // prefix increment: move forward
    Dereferenceable Iter::operator++(int); // postfix increment
    reference Iter::operator*(); // dereference
}
```

Here, the pre-increment operator always returns a reference to itself (i.e., it yields an **Iterator**). In contrast, the result of the post-increment operator only needs to be **Dereferenceable**. The weakness of this requirement accommodates iterators that return a state-caching proxy when post-incremented (e.g., **ostream_iterator**). The **Movable** requirement on the reference type simply establishes a basis for accessing the result. This also effectively requires the result to be non-**void**.

The definition presented here omits requirements for an associated pointer type and for the availability of **->**: the arrow operator. The requirements for **->** are odd and conditionally dependent upon the iterator's value type [15]. In previous designs, the **->** requirement was handled as an intrinsic; that's most likely necessary.

Forward, *Bidirectional*, and *Random Access Iterators* have changed little in this design. These concepts refine the semantics of traversal for *Iterators*. However, *Input Iterators* and *Output Iterators* have been "demoted' to constraints (see below). A *Forward Iterator* is a multi-pass *Iterator* that abstracts the traversal patterns of singly-linked lists:

```
concept Forward_iterator<typename Iter> : Iterator<Iter> {
    requires constraint Convertible<iterator_category, forward_iterator_tag>;
    requires constraint Input_iterator<Iter>;
    Iter Iter::operator++(int); // postfix increment---strengthen Iterator's requirement

    axiom Multipass_equality(Iter i, Iter j) {
        (i == j) => (++i == ++j); // equal iterators are equal after moving
    }
    axiom Copy_preservation(Iter i) {
        (++Iter{i}, *i) == *i; // modifying a copy does not invalidate the original
    }
}
```

An **Input_iterator** only requires its reference type to be convertible to its value type. For a **Forward_iterator** this requirement strengthened so that, like the pre-increment operator, its post-increment must return an **Iterator**. The two axioms specify the semantic properties of multi-pass iterators: equivalent iterators will be equivalent after incrementing and incrementing a copy of an iterator does not invalidate the original.

Finally (and notably), **Forward_iterator**s are statically differentiated from **Input_-iterators** by checking convertibility of their iterator categories. This allows the compiler to automatically distinguish between models of the two concepts without requiring a concept map (as was needed in the C++0x design). Consider:

```
template<typename T, Allocator A>
class vector {
    template<Iterator Iter>
    vector(Iter first, Iter last) { // general version (uses only a single traversal)
        for( ; first != last; ++first)
            push_back(*first);
    }
    template<Forward_iterator Iter>
    vector(Iter first, Iter last) {
        resize(first, last); // traverse once to find size
        copy(first, last, begin()); // traverse again to copy
    }
};
```

The **vector**'s range constructor is optimized for **Forward_Iterators**. However, this is not just a performance issue. Selecting the wrong version leads to serious semantic problems. For example, invoking the second (multi-pass) version for an input stream iterator would cause the system to hang (wait forever for more input). With the automatic concept checking enabled by the **Convertible** requirement, we leverage the existing iterator classification to avoid this problem.

A *Bidirectional Iterator* is a *Forward Iterator* that can be moved in two directions (via **++** and **–**). It abstracts the notion of traversal for doubly linked lists:

```
concept Bidirectional_iterator<Iter> : Forward_iterator<Iter> {
    Iter& Iter::operator--(); // prefix decrement: move backwards
    Iter Iter:: operator--(int); // postfix decrement
    axiom Bidirectional(Iter i, Iter j) {
        i==j => --(++j)==i;
    }
}
```

A *Random Access Iterator* is an *Ordered Bidirectional Iterator* that can be moved multiple "jumps" in constant time; it generalizes the notion of pointers:

```
concept Random_access_iterator<Ordered Iter> : Bidirectional_iterator<Iter> {
    Iter& operator+=(Iter, difference_type);
    Iter operator+(Iter, difference_type);
    Iter operator+(difference_type, Iter);
    Iter& operator-=(Iter, difference_type);
    Iter operator-(Iter, difference_type);
    difference_type operator-(Iter, Iter); // distance
    reference operator[](difference_type n); // subscripting

    axiom Subscript_equality(Iter i, difference_type n) {
        i[n] == *(i + n); // subscripting is defined in terms of pointer arithmetic
    }
    axiom Distance_complexity(Iter i, Iter j) {
        runtime_complexity(i - j)==O(1); // the expression i-j must be constant time
    }
}
```

The requirements on the result type of the decrement operators are analogous to the requirements for increment operators of the **Forward_iterator** concept. The **Random_-access_iterator** concept requires a set of operations supporting random access (**+** and **-**). Of particular interest are the requirement on **Ordered** and the ability to compute the distance between two iterators (in constant time) via subtraction. Semantically, the **Subscript_equality** axiom defines the meaning of the subscript operator in terms of pointer arithmetic. The **Distance_complexity** axiom requires the computation of distance in constant time. Similar requirements must also be stated for random access addition and subtraction, but are omitted here. Here, **runtime_complexity** and **O** are intrinsics used to describe complexity requirement.

Input_iterator and **Output_iterator** are defined as constraints rather than concepts. To allow proxies, their conversion and assignment requirements are expressed in terms of a type "specified elsewhere." The C++0x design relies on associated types, which are expressed as member types. These are implicit parameters that cannot be universally deduced from the iterator. As is conventional, we rely on type traits to support this association. A constrained template or other concept must be responsible for supplying the external argument. However, our definition provides the obvious default case (the iterator's value type):

```
constraint Input_iterator<typename Iter, typename T = value_type<Iter>> {
     requires constraint Convertible<iterator_traits<Iter>::reference, T>;
}
constraint Output_iterator<typename Iter, typename T = value_type<Iter>&&> {
    requires constraint Assignable<iterator_traits<Iter>::reference, T>;
}
```

Here, we assume that **value_type<Iter>** is a template alias yielding the appropriate value type. The constraints for **Input_iterator** and **Output_iterator** support the ability to read an object of type **T** and write an object of type **T** respectively.

6 Constraints

In this section, we give an overview of constraints that have been described in previous sections. Many discussed thus far have direct corollaries in the C++ Standard Library type traits or are easily derived; they are basically a cleaned-up interface to what is already standard (and not counted as concepts in any design). Several that we have used in our implementation are described in Table 2.

Table 2. Type constraints

Constraint	*Definition*
Same<Args...>	Defined in terms of **is_same**
Common<Args..>	True if **common_type<Args...>** is valid
Derive<T, U>	**is_base_of<T, U>**
Convertible<T, U>	**is_convertible<T, U>**
Signed_int<T>	**is_signed<T, U>**

The **Same** constraint requires that all argument types be the same type. It is a variadic template that is defined recursively in terms of **is_same**. The **Common** constraint requires its argument types to all share a common type as defined by the language requirements of the conditional operator (**?:**). Finding the common type of a set of types implies that all types in that set can be implicitly converted to that type. This directly supports the definition of semantics on the common type of a function's arguments for the **Operation** concept. **Derived** and **Convertible** are trivially defined in terms of their corresponding type traits. We note that **Signed_int** is a constraint because its definition is closed: only built-in signed integral types satisfy the requirement (e.g., **int**, and **long**).

Constraints describing the syntax of construction, destruction, and assignment are given in Table 3, and constraints describing overloadable operators in Table 4. The constraints for construction, destruction, and assignment are trivially defined in terms of existing type traits. The operator constraints require the existence of overloadable operators for the specified type parameters and may specify conversion requirements on their result types. For binary operators, the second parameter defaults to the first so that, say, **Equal<T>** evaluates the existence of **operator==(T, T)**.

The **Equal** and **Less** constraints require their results to be bool, which allows the constrained expression to be evaluated in a Boolean evaluation context. The other operators do not impose constraints on their result types because we cannot reasonably define universally applicable conversion requirements for all uses of those operators. In essence, these constraints are purely syntactic; any meaning must be imposed by some other concept or template.

Table 3. Constraints for object initialization

Constraint	*Definition*
Destructible<T>	**is_destructible<T>**
Constructible<T, Args..>	**is_constructible<T, Args...>**
Assignable<T, U>	**is_assignable<T, U>**

Table 4. Constraints for overloadable operators

Constraint	*Definition*
Equal<T, U=T>	**bool operator==(T, U)**
Less<T, U=T>	**bool operator<(T, U)**
Logical_and<T, U=T>	**auto operator&&(T, U)**
Logical_or<T, U=T>	**auto operator||(T, U)**
Logical_not<T>	**auto operator!(T)**
Derefernce<T>	**auto operator*(T)**

7 Implementation and Validation

We implemented the described concepts and traits by building a custom library of constraint classes in C++11. Our approach blends traditional techniques for implementing constraint classes [31] with template metaprogramming facilities [1]. The result is a lightweight concept emulation library that can be used to statically enforce constraints on template parameters and supports concept overloading using concept-controlled polymorphism (**enable_if**) [23]. The library is implemented as a core component of the Origin Libraries [38].

To simplify experimentation, the library does not differentiate between concepts and constraints except through naming conventions. Concepts names in the implementation are written **cConcept**, constraint names are written **tConstraint** ("t" stands for "type property") and axioms, **aAxiom**. This naming convention is chosen for the implementation so that it will not collide with "real" concepts when defined with language support. For example, the **Constructible** constraint described in 6 is implemented in Origin as **tConstructible**:

```
template<typename T, typename... Args>
struct tConstructible {
  tConstructible() { auto p = constraints; }

  static void constraints(Args... args) {
    T{forward<Args>(args)...}; // use pattern for copy construction
  }

  typedef tuple<std::is_constructible<T, Args...>> requirements;
  typedef typename requires_all<requirements>::type type;
  static constexpr bool value = type::value;
};
```

tConstructible is a constraint because no sensible semantics can be defined for construction, beyond what the language already guarantees for constructors. The **tConstructible** constructor is responsible for instantiating the constraints function, which contains the use patterns for the concept, which are similar to those introduced in [15]. The function parameters of the constraints function introduce objects, which simplifies the writing of use patterns.

The type and value members satisfy the requirements of a type trait or Boolean metafunction. These members, especially value, are used to reason about the type at compile time without causing compiler errors. This is used to select between overloads based on modeled satisfied requirements using **enable_if** [23].

All concepts in the library are automatically checked. Implementing a concept library that requires users to write explicit concept maps would require us to do so for every data structure tested. That approach is not needed for the STL and, in our opinion, does not scale well. Axioms are not (and cannot be) checked by the compiler so for our validation we treat them as comments.

Function template requirements are specified by explicit constructions of temporary objects. For example:

```
template<typename T>
T const& min(T const& x, T const& y) {
    cOrdered<T>{}; // requires that T has operators <, >, <=, and >=
    return y < x ? y : x;
}
```

Here **cOrdered<T>** denotes a requirement on the template parameter **T**. Instantiating the algorithm entails instantiating the **cOrdered<T>** constructor and its nested requirements. Compilation terminates if a constraint class is instantiated with template arguments that do not satisfy the required use patterns.

For class templates, the requirements are specified as base classes. For example:

```
template<typename T> class Vector : private cRegular<T> { /* ... */};
```

This ensures that the compiler instantiates the concept checks when constructing objects of the constrained class. Requirements within constraint classes are written in exactly the same way: explicit construction is used in conjunction with use patterns, and inheritance is used to emulate concept refinement.

There are no memory or performance costs induced by the use of Origin's constraint classes. Constraint instances are optimized out of the generated code either through dead-code elimination or the empty base optimization. Compile times can be increased marginally but are no worse than using any other concept checking library.

We applied the constraint classes to a subset of the Clang and GCC implementations of the Standard Library: the functional, algorithm, and iterator components (the STL). Class and function template constraints were written for each data structure and algorithm exported by those components. We iteratively refined both the concepts and the constraints as required by the limits of the experiment.

We use the libraries' test suites (just over 9,000 programs at the time of writing) to check the correctness of the concepts. A more substantial validation of our design could be achieved by compiling a large number of C++ applications against the modified libraries, but the test suite cover a sufficient number of instantiations so we are confident in the design.

Test suite failures generally indicated overly strict constraints on a type or algorithm. In some cases, such failures also indicated what we perceive as problems with the original conceptual specification for the library. In such cases, these failures are due to the representation of irregular types as legitimate concepts. For example, strict output iterators such as **ostream_iterator** are *Iterators* in principle but are neither default constructible nor equality comparable. We modified these irregular cases so they would model the required concepts. There are only four such iterators in the STL, and they are easily adapted to model our proposed concepts.

8 Conclusions

We studied concept design rather than language design for expressing concepts with the aim of bringing empirical evidence to the center of language design discussions. We found that explicitly differentiating between concepts and constraints based on semantic requirements (axioms) improved our analyses and clarified long-standing "dark corners" of the STL design. It led to spectacular simplification and a dramatic reduction in the number of concepts needed to describe the STL interfaces (14 rather than 108). The STL interfaces were already considered well understood after more than a decade's use and much analysis, so we conclude that our concept design technique is nowhere near as obvious as it seems in retrospect. Our technique is rooted in classical algebraic theory, so we further conjecture that it will be very widely applicable.

Our conclusions on language design for concepts are, as we expected them to be, very tentative. However, we have demonstrated a central role for axioms, a feature that was widely conjectured to be unnecessary during the C++0x design. Beyond that, we found constraints classes so expressive and manageable in our implementation that we want to re-examine the use-pattern approach for expressing syntactic requirements.

We continue to investigate concepts for the C++ Standard Library by broadening our conceptual analysis to cover numeric and scientific computing domains and containers. With Origin, we are pursuing concepts related to heaps and graphs [30]. Exploring conceptual designs in different domains supports language design by addressing a broader set of use cases. In particular, we plan to examine uses of concepts in algorithm specifications with the aim of simplifying such specifications.

Concepts are abstract, general, and meaningful. Consequently, they are unlikely to be specific to a specific library. As concepts mature, they become a repository of fundamental domain knowledge. Thus, we expect concepts to cross library boundaries to become more widely useful. We expect that our exploration of the definition and use of concepts will decrease the total number of concepts (among all libraries) while improving their quality and utility.

Acknowledgements. Thanks to Matt Austern, Paul McJones, Gabriel Dos Reis, and Alex Stepanov for comments that led to major improvements and will be the basis for further work. This project was partially supported by NSF grants A3350-32525-CS and A0040-32525-CS and Award KUS-C1-016-04, made by King Abdullah University of Science and Technology (KAUST).

References

1. Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. C++ In-Depth. Addison Wesley (2004)
2. Austern, M.: Generic Programming and the STL: Using and Extending the C++ Standard Template Library, 7th edn. Addison-Welsey Longman, Boston (1998)
3. Austern, M., Stroustrup, B., Thorup, M., Wilkinson, J.: Untangling the Balancing and Searching of Balanced Binary Search Trees. Software: Practice and Experience 33(13), 1273–1298 (2003)
4. Bagge, A.H., David, V., Haveraaen, M.: The Axioms Strike Back: Testing with Concepts and Axioms in C++. In: 8th International Conference on Generative Programming and Component Engineering (GPCE 2009), Denver, Colorado, pp. 15–24 (2010)
5. Becker, P.: Working Draft, Standard for the Programming Language C++. Tech. Rep. N2914, ISO/IEC JTC 1, Information Technology Subcommittee SC 22, Programming Language C++ (2009)
6. Bernardy, J.P., Jansson, P., Zalewski, M., Schupp, S., Priesnitz, A.: A Comparison of C++ Concepts and Haskell Type Classes. In: Workshop on Generic Programming (WGP 2008), Victoria, Canada, pp. 37–48 (2008)
7. Bernardy, J.-P., Jansson, P., Zalewski, M., Schupp, S.: Generic Programming with C++ Concepts and Haskell Type Classes–A Comparison. Journal of Functional Programming 20(3-4), 271–302 (2010)
8. Bracha, G., Odersky, M., Stoutamire, D., Wadler, P.: Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In: 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 1998), Vancouver, Canada, pp. 183–200 (1998)
9. Davenport, J.H., Gianni, P.M., Trager, B.M.: Scratchpad's View of Algebra II: A Categorical View of Factorization. In: International Symposium on Symbolic and Algebraic Computation (ISSAC 1991), Bonn, Germany, pp. 32–38 (1991)
10. Davenport, J.H., Sutor, R.S.: AXIOM: The Scientific Computation System. Springer (1992)
11. Davenport, J.H., Trager, B.M.: Scratchpad's View of Algebra I: Basic Commutative Algebra. In: Miola, A. (ed.) DISCO 1990. LNCS, vol. 429, pp. 40–54. Springer, Heidelberg (1990)
12. David, V.: Concepts as Syntactic Sugar. In: 9th International Working Conference on Source Code Analysis and Manipulation (SCAM 2009), Alberta, Canada, pp. 147–156 (2009)
13. Dehnert, J.C., Stepanov, A.: Fundamentals of Generic Programming. In: Jazayeri, M., Musser, D.R., Loos, R.G.K. (eds.) Dagstuhl Seminar 1998. LNCS, vol. 1766, pp. 1–11. Springer, Heidelberg (2000)
14. Dos Reis, G.: Personal Communication (October 2010)
15. Dos Reis, G., Stroustrup, B.: Specifying C++ Concepts. In: 33rd Symposium on Principles of Programming Languages (POPL 2006), Charleston, South Carolina, pp. 295–308 (2006)
16. Dos Reis, G., Stroustrup, B., Merideth, A.: Axioms: Semantics Aspects of C++ Concepts. Tech. Rep. N2887, ISO/IEC JTC 1, Information Technology Subcommittee SC 22, Programming Language C++ (2009)
17. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns Elements of Reusable Object-Oriented Software. Addison-Wesley (1994)
18. Garcia, R., Järvi, J., Lumsdaine, A., Siek, J., Willcock, J.: An Extended Comparative Study of Language Support for Generic Programming. Journal of Functional Programming 17, 145–205 (2007)
19. Gregor, D.: ConceptGCC (2008), http://www.generic-programming.org/software/ConceptGCC/

20. Gregor, D., Järvi, J., Siek, J., Stroustrup, B., Dos Reis, G., Lumsdaine, A.: Concepts: Linguistic Support for Generic Programming in C++. In: ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2006), Portland, Oregon, pp. 291–310 (2006)
21. Hewlett-Packard: Standard Template Library Programmer's Guide (1994), http://www.sgi.com/tech/stl/index.html
22. International Organization for Standards: International Standard ISO/IEC 14882. Programming Languages — C++ (2003)
23. Järvi, J., Willcock, J., Lumsdaine, A.: Concept-Controlled Polymorphism. In: Pfenning, F., Macko, M. (eds.) GPCE 2003. LNCS, vol. 2830, pp. 228–244. Springer, Heidelberg (2003)
24. Jones, M.P.: Type Classes with Functional Dependencies. In: Smolka, G. (ed.) ESOP 2000. LNCS, vol. 1782, pp. 230–244. Springer, Heidelberg (2000)
25. Kennedy, A., Syme, D.: Design and Implementation of Generics for the .NET Common Language Runtime. In: ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI 2001), Snowbird, Utah, pp. 1–12 (2001)
26. Meyer, B.: Eiffel: The Language. Prentice-Hall (1991)
27. Milner, R., Harper, R., MacQueen, D., Tofte, M.: The Definition of Standard ML - Revised. The MIT Press (1997)
28. Musser, D., Stepanov, A.: Algorithm-oriented Generic Libraries. Software: Practice and Experience 24(7), 623–642 (1994)
29. Pirkelbauer, P., Dechev, D., Stroustrup, B.: Support for the Evolution of C++ Generic Functions. In: Malloy, B., Staab, S., van den Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 123–142. Springer, Heidelberg (2011)
30. Siek, J., Lee, L.-Q., Lumsdaine, A.: The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley (2001)
31. Siek, J., Lumsdaine, A.: Concept Checking: Binding Parametric Polymorphism in C++. In: 1st Workshop on C++ Template Programming, Erfurt, Germany (2000)
32. Siek, J., Lumsdaine, A.: Essential Language Support for Generic Programming. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005), Chicago, Illinois, pp. 73–84 (2005)
33. Siek, J., Lumsdaine, A.: Language Requirements for Large-Scale Generic Libraries. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 405–421. Springer, Heidelberg (2005)
34. Stepanov, A., McJones, P.: Elements of Programming. Addison Wesley, Boston (2009)
35. Stroustrup, B.: Parameterized Types for C++. Computing Systems 2(1), 55–85 (1989)
36. Stroustrup, B.: The Design and Evolution of C++. Addison-Wesley (1994)
37. Stroustrup, B.: "New" Value Terminology (2010), http://www2.research.att.com/~bs/terminology.pdf
38. Sutton, A.: Origin C++0x Libraries (2011), http://code.google.com/p/origin
39. Sutton, A., Holeman, R., Maletic, J.I.: Identification of Idiom Usage in C++ Generic Libraries. In: 18th International Conference on Program Comprehension (ICPC 2010), Braga, Portugal, pp. 160–169 (2010)
40. Sutton, A., Maletic, J.I.: Automatically Identifying C++0x Concepts in Function Templates. In: 24th International Conference on Software Maintenance (ICSM 2004), Beijing, China, pp. 57–66 (2008)
41. Torgersen, M., Hansen, C.P., Ernst, E., von der Ahé, P., Bracha, G., Gafter, N.: Adding Wildcards to the Java Programming Language. In: ACM Symposium on Applied Computing, SAC 2004, Nicosia, Cyprus, pp. 1289–1296 (2004)
42. Veldhuizen, T.: Expression Templates. C++ Report 7(5), 26–31 (1995)

Join Token-Based Event Handling: A Comprehensive Framework for Game Programming

Taketoshi Nishimori and Yasushi Kuno

Graduate School of Business Sciences, University of Tsukuba,
Tokyo, 3-29-1 Otsuka, Bunkyo-ku, Tokyo, 112-0012, Japan
nis@nisnis.jp
kuno@gssm.otsuka.tsukuba.ac.jp
http://www.gssm.otsuka.tsukuba.ac.jp/

Abstract. In action game programming, programmers have to control multiple concurrent activities on the screen corresponding to multiple game characters. To address this difficulty, many game-oriented scripting languages have been proposed so far. However, current scripting languages seem to lack support for interactions among multiple concurrent activities in a state-dependent manner. To overcome this problem, we propose an event handling framework called "join token" in which the states of game characters can be expressed as tokens and interactions can be described as handlers specifying multiple tokens. For the purpose of evaluation, we have developed a game scripting language called "Mogemoge," and wrote several sample games in this language. In this paper, we describe experiences of using join token framework for sample games and compare the code written in Mogemoge against a code written in an existing scripting language.

Keywords: video game, programming language, event handling framework, scripting language.

1 Introduction

Video game programming, especially that used for action games, has the distinguishing characteristic that programmers have to manage multiple concurrent activities on the screen corresponding to multiple game characters. For example, in many shooting games, multiple missiles are concurrently moving on the game screen, and when those missiles "hit" various objects, the resulting effects are different depending on the kind of objects and their states such as whether they have a shield or not.

Managing concurrent activities in general purpose programming languages such as C++ or Java is notoriously difficult and complex.

One way to deal with this problem is to use game-oriented scripting languages. Scripting languages can provide language mechanisms and/or frameworks to support concurrent activities of multiple game characters, so that programmers can describe the logic of the game in a more straightforward manner.

For example, Stackless Python [12][15] supports micro-threads that make it feasible to assign a dedicated thread to each of the game characters. However, this approach does not address the problem of interaction among the characters, which is the main focus of this paper.

UnrealScript[14] supports concurrent objects called "actors." In this scripting language, methods are invoked under some corresponding conditions. However, UnrealScript also does not address the problem of interaction among the characters.

To address this problem, we propose an event handling framework called "join token" that coordinates multiple, state-dependent, concurrent activities required for a game description[9]. To assess the effectiveness of this mechanism, we have designed and implemented an experimental game-oriented scripting language called "Mogemoge" that incorporates join token as a built-in coordination mechanism. For the purpose of evaluation, we have written several demo games using Mogemoge.

The concept of join token is based on join-calculus [4] and Linda[7] computational models. Join-calculus models the coordination of multiple concurrent tasks. Linda models the decoupling of the message sender and receiver. As far as we know, there are several programming languages based on either of these models, but no language has combined both of these models.

The major contribution of this paper is our interim evaluation of join token framework through two demonstration game implementations, one of which was also written in Ruby for comparison purposes. The concept of the join token framework and an overview of Mogemoge language are already described in [9]; we briefly presented these in this paper because these are necessity to understand the main point of the research.

The structure of this paper is as follows. In Section 2, we explain the idea and design of the join token framework and discuss its characteristics. In Section 3, we provide an overview of Mogemoge language, along with its implementation. In Section 4, we explain two sample games implemented in Mogemoge, and present a comparison with one of the games implemented in Ruby/Tk. In Section 5, we explain related works and discuss the strong points of join token. In Section 6, other issue including concurrency and performance are discussed, and finally the conclusion is drawn in Section 7.

2 Join Token: an Event Handling Framework Suitable for Games

The majority of game programs and/or game scripting languages are based on object orientation, because many games are based on simulated behavior of real (or virtual) objects. In object orientation, behaviors (actions) are described as methods attached to one of those objects. Methods are implemented as subroutines and are called from other methods (or from the main routine).

However, the above design differs significantly from interactions in game programs, as follows:

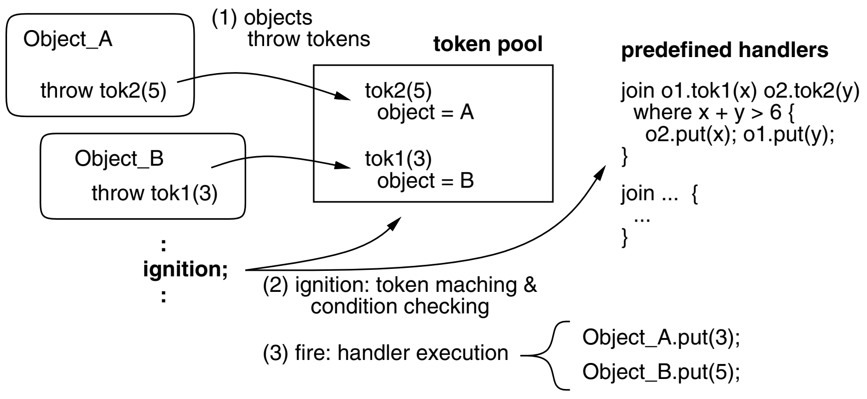


Fig. 1. Idea of join token

- Interactions in games are associated with two or more characters, while methods are attached to a single object.
- Interactions in games are initiated when some conditions are met, while methods are invoked from some other methods.
- Interactions are controlled by the states of each associated character, while method invocations are controlled solely by the calling object.

These differences make it difficult to express the programmers' (or the game designers') idea in a straightforward manner when using an object-oriented (O-O) language.

To overcome these problems, we propose the new event handling framework called "join token," as a supplementary mechanism to conventional object orientation (**Fig.1**).

- Each object participating in an interaction expresses its willingness to participate by generating a "token." A token is associated with the object that generated the token and a list of parameters specified in the code.
- Tokens are generated when methods execute "throw statements," and the generated tokens are automatically put into the global "token pool."
- An interaction (multiple object action) is described as a "join statement" that defines a "join handler" (hereafter referred to as "handler"). A handler specifies the set of tokens that participate in the interaction, the optional conditions, and the body statements that are executed when the interaction occurs.
- Interactions are started when the special "ignition" statement is executed in the program; this statement corresponds to the "handle event" phase of a game program, and is expected to be used in the main loop of the game program. When the ignition statement is executed, the token pool scans the list of defined handlers one by one, and tries to select the tokens that match with a handler.

– When all the handler's token specifications are matched with the existing tokens and the handler's associated condition is true, if specified, the handler "fires," and the body of the handler is executed. Within the body, each token's associated objects and parameters are available. The tokens that participated in a fire are removed from the token pool unless otherwise specified.

The major benefit of the above design is that handlers are neutral to all objects and are associated with the global token pool. The separation of object interactions (handlers) and each object's behavior (methods) simplifies the structure of game scripts, as shown in later sections.

Our target is not a parallel programming language, but rather a game scripting language, in which event ordering should by strictly defined and controllable. Therefore, a list of handlers is scanned in the order of their definition (source program), and each handler consumes as many tokens as possible when it is considered, with matching being performed in the order described in the corresponding join statement. Token matching is also performed strictly as per the orderings; an older token in the pool is considered earlier. Note that a handler can fire multiple times when there are sufficient tokens and when conditions permit. The tokens generated within the method bodies invoked from the handler bodies can participate in subsequent matching. Therefore, one can consume a token by a handler and immediately regenerate the same token in its body if necessary.

Tokens are identified with their names and the originating object. Therefore, when an object throws tokens with the same name twice and the first token is not consumed, the second token replaces the first one; the number of arguments may vary among these tokens. Such operations are useful when one would like to overwrite the arguments of some tokens. Alternatively, one can withdraw a token with the "`dispose`" statement.

3 The Mogemoge Language

Mogemoge is an experimental game scripting language equipped with a join token mechanism. The purpose of developing this language is to evaluate the usefulness and descriptive potential of join token. Therefore, Mogemoge was designed to be a minimal, compact and simple programming language, except for the part executing the join token mechanism.

3.1 Basic Features

We have used prototype-based object orientation similar to that used in JavaScript and Self[16], because it can lead to a compact language definition. Therefore, Mogemoge provides the syntax to create concrete objects, and those objects can be used as the prototypes from which subordinate objects (logical copies) can be created.

In the same manner, we have designed the functionalities of Mogemoge to be minimal, namely, (1) object definition/creation, (2) method definition/invocation and (3) action descriptions through executable statements. Below, we show these functionalities with the use of small code examples.

The following Mogemoge code creates a game character object:

```
Char = object {
  x = 0; y = 0;
  init_char = method(_x, _y) { x = _x; y = _y; }
  ...
};
```

The above code creates an ordinary object and assigns it to the global variable named "`Char`." Within the object definition, variable assignments define and initialize the object's instance variables. Note that the methods are also ordinary objects and are stored in instance variables.

A "`new`" operator creates an object through copying:

```
c = new Char;
```

In the above code, the "`new`" operator creates a fresh object and then copies all of the properties including the variables and their values from the Character object. The resulting object is assigned to the variable "`c`." To invoke methods on an object, the dot notation is used, similar to that used in Java or C++.

An assignment stores a value to the specified variable. When the variable does not exist, a new one is created. A "`my`" modifier forces the creation of local variables for the surrounding scope. Although the syntax was borrowed from Perl, its intention is more close to that of the local "`var`" in JavaScript.

```
foo = method() {
  my x = 1;
  result method(d) { result x; x = x + d; }
};
```

The "`result`" statement specifies the return value of the method. Therefore, the `foo` method returns an anonymous method object, which increments the value stored in `x` by `d` and returns the old value of `x`.

Mogemoge also has the following features, which we will not describe in here.

– C# like delegation
– Composition (compose objects and create a new object)
– Injection (modify an object by adding variables)
– Extraction (modify an object by deleting variables)

3.2 Join Token Feature

A "`throw`" statement adds a token to the global token pool:

```
throw tok1(1, 2);
```

Conversely, a dispose statement removes a token from the token pool. The following statement removes tok1 that is thrown by the object executing the method code:

```
dispose tok1;
```

A join statement defines a handler. The following is an example of a handler definition:

```
join r1.tok1(a, b) r2.tok2(c, d) {
  print "a + c = " + (a + c);
  print "b + d = " + (b + d);
};
```

In the above example, the handler fires when both tokens tok1 and tok2 are in the token pool. The term r1.tok1(a, b) means that the handler matches the tok1 token and two arguments can be extracted; when the number of actual arguments is not 2, the extra values are discarded and nil values are used for the missing values.

When the handler is invoked, a and b represent the corresponding argument values for the matched token, and r1 represents the object that has thrown the matched token. The term r2.tok2(c, d) can be read likewise. When the body of the handler is being executed, the matched values can be used.

The tokens matched against a handler are removed from the pool by default, but when a token specifier is prefixed with the symbol "*," the token is retained in the pool. Following is an example:

```
join r1.tok1(a) *r2.tok2(b) { ... }
```

Note that the tokens left in the pool can be consumed by a sussessive handler defined in the code, or can remain in the pool until the next ignition.

Join handlers may optionally be guarded by Boolean expressions introduced by a where clause. In the following example, the handler is invoked only when the arguments of the two tokens are identical:

```
join r1.tok1(a) r2.tok2(b) where a == b { ... };
```

Aside from join statements, the existence of a token can be examined by an exist operator, as follows:

```
if (exist tok) { ...  }
```

3.3 Implementation

We have implemented Mogemoge using Java and the SableCC[6] compiler compiler framework. The lexical and syntax definition (about 160 lines of code) is translated by SableCC to Java code, which implements the lexical analyzer and the parser. The parser generates an abstract syntax tree (AST) from the source program. Our interpreter inherits from the tree walking code (also generated by SableCC) and executes the program actions while traversing the tree. The total size of the Mogemoge interpreter is about 2400 lines of code, including the Java and the SableCC definitions.

Since Mogemoge runs on the Java Virtual Machine(JVM), it is easy to interface Mogemoge code with Java code. Actually, in our implementation, game graphics routines are written in Java and called from Mogemoge code. A special syntax is provided to declare Mogemoge-callable Java method signatures. Conversely, Java code can call Mogemoge routines and can access Mogemoge data structures (tokens and the token pool). However it is a bit difficult, because calling conventions have to be maintained.

The token pool, tokens, and join handlers are represented using Java data structures and the associated lookup code. When an ignition statement is executed, the list of defined join handlers is examined one by one, while searching for matched tokens in the pool. When a sufficient token for the handler is found, the `where` clause (if any) is executed, and then, if the condition is satisfied, the handler body is executed. Note that the `where` clauses and handler bodies are represented as AST data structures and are stored within the handler object.

The current algorithm for a token-handler match uses a simple linear search, and so far, this algorithm has not caused any performance problems with approximately 100 handlers and 1000 tokens. If necessary, we could implement additional index data structures to speed up the search.

4 Evaluation of the Join Token Framework

We have implemented several example games using the Mogemoge language and the join token framework. In the following sections, we describe two such games and discuss our model. In addition, we have implemented one of the games using an existing programming language (namely, Ruby with Tk graphics) and compared the resulting code with the Mogemoge version.

4.1 “Baloon” Game

“Balloon” is a simple shooting game written in Mogemoge; the number of lines of code is 357. Its screenshots are shown in **Fig.**2. In this game, balloons with hanging bombs come down from the sky. The player controls a battery (at the left-bottom corner of the game screen), and shoots/explodes missiles to destroy the bombs, so that bombs do not hit buildings at the bottom of the screen.

One missile can destroy only one balloon or bomb when the missile hits them directly. Alternatively, explosion of the missile can destroy multiple balloons and bombs that fall within the explosion area.

Rules of this game are summarized as follows:

R1. A battery’s direction is controlled by the player.
R2. The player can shoot/explode a missile by pressing/releasing a key.
R3. A balloon falls slowly from the top of screen. A bomb is bound to the tip of a string hang from the balloon.

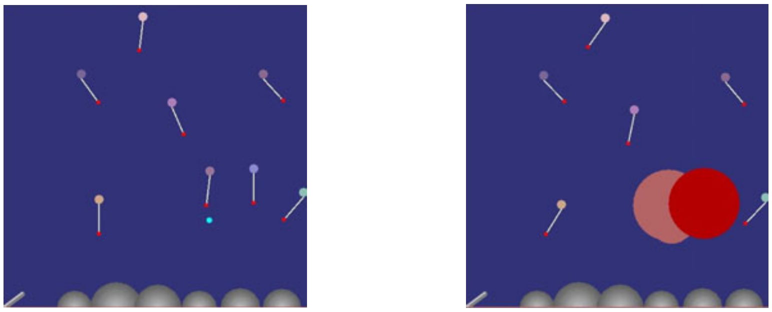


Fig. 2. Screenshot of "Balloon" Game

R4. A missile can destroy a balloon or bomb *without an explosion.*
R5. If either a balloon or a bomb is destroyed while connected, the remaining one continues to fall. A bomb falls faster than a balloon.
R6. An explosion of a missile/bomb can destroy balloons.
R7. An explosion of a missile/bomb can explode bombs.
R8. A bomb and an explosion can destroy a building at the bottom of the game screen.
R9. An explosion is not destroyed by any objects except for a building. An explosion cannot destroy two or more buildings.

These rules are classified into two categories: rules that specify relationships (or interactions) between game characters and other rules. Relationship rules are **R3**, **R4**, **R5**, **R6**, **R7**, **R8**, and **R9**. Non-relationship rules are **R1**, **R2**. Non-relationship rules can be implemented as ordinary methods associated with corresponding objects in a straightforward manner. However, when ordinary method are used, some elaborated coding will be required to implement relationship rules, because two or more objects are involved with these rules.

With our join token framework, these relationship rules can be represented as one or more join handlers in a clear and straightforward manner.

The following code initializes a balloon and a bomb:

```
Balloon = object {
  init = method( x ) {     # a method which initializes a balloon
      bomb = new Bomb; bomb.init( x, 0 ); # create a bomb
      throw balloon(bomb); # throws a token with a bomb
  }
};
Bomb = object {
  init = method( x, y ) {  # a method which initializes a bomb
    throw bomb;            # throws a bomb token
  }
};
```

When a balloon is created, it also creates a `bomb` and a `balloon` token associated with itself (as a throwing object) and the bomb (as an argument).

Similarly, other objects such as a missile or an explosion throws a token named `missile` or `explosion` (this time with no arguments) correspondingly, at the time of creation.

The rules related to collisions/explosions (**R4**, **R5**, **R6**, **R7**, **R8**, **R9**) can be implemented with the following handler code:

```
# rule R4+R5: A missile destroys a balloon.
join m.missile() bln.balloon(bomb) where m.is_collided( bln ) {
  m.destroy(); bln.destroy();  # destroys the missile and the balloon
  bomb.set_vel( 0, 2 );        # add falling velocity to the bomb
}
# rule R4(+R5): A missile destroys a bomb.
join m.missile() b.bomb() where m.is_collided( b ) {
  m.destroy(); b.destroy();    # destroy the missile and the bomb
}
# rule R6(+R5): An explosion destroys a balloon.
join *e.explosion() bln.balloon(bomb) where e.is_collided( bln ) {
  bln.destroy();               # destroys a balloon
  bomb.set_vel( 0, 2 );        # add falling velocity to the bomb
}
# rule R7(+R5): An explosion explodes a bomb.
join *e.explosion() b.bomb() where e.is_collided( b ) {
  b.destroy();
  e = new Explosion;           # an explosion takes place
  e.init( b.x, b.y, 1.2 );     # initialize position and size
}
# rule R8: A bomb destroys a building.
join b.bomb() bld.building() where b.is_collided(bld) {
  bld.destroy(); b.destroy();  # destroys the building and the bomb
}
# rule R8+R9: An explosion destroys a building.
join e.explosion() bld.building() where e.is_collided(bld) {
  bld.destroy();               # destroys the building
}
```

The handlers corresponding to rules **R6** and **R7** use the `*` symbol to retain `explosion` tokens in the pool for some duration, so that they can destroy multiple balloons and bombs. The handler corresponding to the rule **R9** (the last handler) is an exception; an `explosion` token is deleted when an explosion collides with a building.

The binding rule, **R3**, is implemented by the following handler code:

```
# rule R3: a bomb is bound to a balloon.
join *bln.balloon(child) *b.bomb() where child == b {
    b.set_pos( bln.tipx, bln.tipy );
};
```

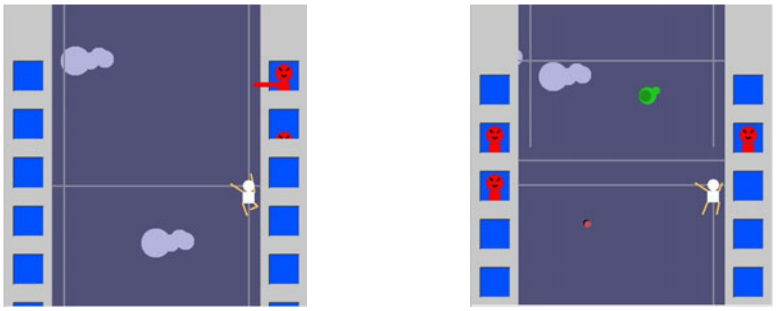


Fig. 3. Screenshot of "Descender" Game

If there is a pair of `balloon` token and `bomb` token such that an argument of the `balloon` is the object associated with the `bomb` token, this means that the corresponding bomb is bound to the corresponding balloon. Therefore, the position of a bomb is set to the tip of the string of a balloon (variables `tipx` and `tipy` of a balloon means the position of the tip of the string). This handler is intended to fire many times to continually adjust the positions of bombs. So, `*` symbols are used to retain the corresponding tokens in the pool

The rule **R5** is implemented as a supplementary code in the handler for rules **R4**, **R6**, and **R7**. When a balloon is hit by a missile, the falling velocity of the remaining bomb is increased. However, when a bomb is hit by a missile, the speed of the corresponding balloon does not change, so no action is required.

As shown in the above listings, our handlers correspond to game rules in a fairly straightforward manner, and the code described in the handler bodies are simple and readable, suggesting the usefulness of our join token framework.

4.2 "Descender" Game

"Descender" (**Fig.3**) is an action game that is more complicated than the one described in the previous section; the number of lines of code is 847. The aim of the player in this game is to descend infinitely along the wall of two buildings using horizontal and vertical ropes. The player controls where to stretch ropes and his movement along them. Birds drops bombs, and building inhabitants occasionally cut vertical ropes. If a bomb hits the player or an inhabitant cuts the rope that the player is currently hanging on, the game is over.

Rules of this game are summarized as follows:

R1. The player can move along a horizontal or vertical rope from his current position.

R2. When the player is holding the bottom of a vertical rope, or there is no vertical rope under the player, he can extend a vertical rope to the bottom of the screen.

R3. All objects are scrolled upward when the player descends (the vertical coordinate of the player is fixed).

R4. Clouds are scrolled up more slowly than other scrolling characters; they are for visual decoration and have no effect on other objects.
R5. When the player is holding a vertical rope and there is no horizontal rope around him, he can stretch a horizontal rope.
R6. Birds flying in the air occasionally drop bombs.
R7. Inhabitants in some of the windows occasionally cut the vertical rope in front of them.
R8. If a bomb hits the player, the player falls and the game is over.
R9. If an inhabitant cuts the rope that the player is currently holding, the player falls and the game is over.
R10. A vertical rope extends to the bottom infinitely until an inhabitant cuts it.

All of the above rules except **R6** are implemented with join handlers.

In this game, a player is in one of three modes: descending mode (holding a vertical rope), horizontal moving mode (holding a horizontal rope), and falling mode (the game is over). A player's action in each mode is implemented as a corresponding method, and a variable named `update` stores the currently effective mode, as follows:

```
Player = object {
  update_descending = method() { ... }
  update_moving_horizontally = method() { ... }
  update_falling_horizontally = method() { ... }
  update = update_descending;   # the initial mode is descending.
}
```

The following is the descending mode method:

```
update_descending = method() {
  if ( guiKeyPressed( KEY_DOWN ) ) {
    throw cmd_descend;
  }
  if ( guiKeyOn( KEY_LEFT ) ) {
    throw cmd_shoot_hrope;
  } elif ( guiKeyOn( KEY_RIGHT ) ) {
    throw cmd_shoot_hrope;
  }
  is_left_side = ( x == LEFT_PLAYER_X );
  if (guiKeyPressed( KEY_RIGHT ) and is_left_side) {
    throw cmd_go_side;
  } elif (guiKeyPressed( KEY_LEFT ) and not is_left_side) {
    throw cmd_go_side;
  }
};
```

The actions the player can perform in a descending mode are to descend, to stretch a horizontal rope, and to switch to horizontal moving mode. These actions are expressed by throwing either a `cmd_descend`, `cmd_shoot_hrope`, or `cmd_go_side` token. For example, a `cmd_descend` token (thrown when the down arrow key is pressed) is handled by the following two handlers:

```
join p.cmd_descend *r.v_rope where r.is_on( p.x, p.y ) {
  d = min( r.by - p.y, 2 );
  if ( d > 0 ) {
    throw scroll( d );
    p.anim_descend();
  } else {
    r.extend_to_bottom();
  }
};
join p.cmd_descend { };
```

A `v_rope` token is thrown by a vertical rope at its creation time and remains in the pool as long as the rope is available (designated by a * symbol).

A body of the first handler is executed when there is a `cmd_descend` token and a player is on a vertical rope. This handler implements the rules **R1** and **R2**. When the vertical rope has some margin below the player to descend (**R1**), all objects are scrolled up (as the player descends) Otherwise, the vertical rope is extended to the bottom of the screen(**R2**). Note that the `scroll` token is thrown when the player is descending, as explained shortly.

The second handler has an empty body; it simply consumes a `cmd_descend` token when it is not processed by the first handler, e.g., either the player is not descending or the player is not on a vertical rope.

Other command tokens thrown in a descending mode are implemented similarly. All command tokens thrown at a certain animation frame are handled within the frame, and are consumed by an empty handler when they are not effective in that frame.

The game screen scrolls up according to the player's descending action (**R3**). Therefore, all game characters except for the player update their vertical coordinate. As shown above, the handler of rule **R1** throws `scroll` token when the player is descending, and this token is handled by the following handlers (the argument of a `scroll` token represents the number of pixels to scroll):

```
join *any.scroll(d) *o.bg_object {
  o.y = o.y - d;
  if ( o.y < SCROLL_OUT_LIMIT_Y ) { o.destroy(); }
};
join *any.scroll(d) *r.v_rope {
  if ( r.y > 0 or d > 0 ) { #
    r.y = r.y - d;
    if ( r.y < 0 ) { r.y = 0; }
  }
  if ( r.by < HEIGHT ) {
    r.by = r.by - d;
    if ( r.by < 0 ) { r.destroy(); }
  }
};
```

```
join *any.scroll(d) *c.cloud {
  c.y = c.y - d / 2.0;
};
join any.scroll { };
```

Note that the `scroll` token is thrown by the handler that consumes the token `cmd_descend` and is processed by another handlers in the same ignition. Therefore, handlers that processes `scroll` should be placed below the handler that throws `scroll`.

All objects that scroll but have no specific action while scrolling throw a `bg_object` token at initialization; they simply move upward and destroy themselves when they go out of the screen. This is implemented by the first handler.

The second handler is for a vertical rope. A vertical rope, whose top and bottom vertical coordinates are held by `y` and `by` correspondingly, behaves a little differently. As the rule **R10** states, the bottom of a vertical rope does not move upward if its bottom is at the bottom of the game screen.

The third handler implements a cloud that goes slowly (at half the speed of other scrolling objects) up the screen (**R4**). When a cloud goes out of the screen, it resets its position to the bottom to "reuse" itself (this behavior is described in the cloud object and is not shown here).

The fourth handler is defined at the end of handlers processing the `scroll` token to consume a `scroll` token when its job is done, as in the other command tokens.

The rule **R5** is implemented by the following handlers:

```
join p.cmd_shoot_hrope *r.h_rope_flying
      where abs( p.y - r.y ) < BREADTH_TO_CHANGE_ROPE {
};
join p.cmd_shoot_hrope *r.h_rope
      where abs( p.y - r.y ) < BREADTH_TO_CHANGE_ROPE {
};
join p.cmd_shoot_hrope {
  hr = new HorizontalRope;
  if ( p.x == LEFT_PLAYER_X ) {
    hr.init( p.y, false );
    p.anim_shoot_hrope( false );
  } else {
    hr.init( p.y, true );
    p.anim_shoot_hrope( true );
  }
};
join p.cmd_shoot_hrope { };
```

Note that a horizontal rope throws either a `h_rope` token (on which a player can hang) or a `h_rope_flying` token (on which player cannot hang on because it is not completely stretched between buildings).

A horizontal rope can be stretched only when there is no other horizontal rope nearby (**R5**). This behavior is expressed by the first 2 handlers. If there is any horizontal rope within `BREADTH_TO_CHANGE_ROPE` pixels, a `cmd_shoot_hrope` token is simply deleted so that no further action occurs.

The third handler stretches a horizontal rope leftwards or rightwards according to the current position of the player. Note that this hander specifies only one token. Its role is to execute the stretching action when the `cmd_shoot_hrope` token was not removed by the previous handlers.

The fourth handler removes the token when its task is done as in the other handlers that process command tokens.

An inhabitant tries to cut a vertical rope (the rule **R7**). An inhabitant consists of two objects: the inhabitant itself and his arm. An arm object throws a `cmd_cut_rope` token if it has fully extended from its body, and this token is processed by the following handlers (method `can_cut` checks if the arm can actually cut the rope):

```
join a.cmd_cut_rope *r.v_rope where a.can_cut( r ) {
    r.cut( a.y );
};
join a.cmd_cut_rope { };
```

Method `cut` actually cuts the rope and throws a `cut_vrope` token with two arguments, which are the top and bottom y coordinate of the cut part. This token is processed by the handlers corresponding to **R9**, which simply checks if the player should fall and changes its state accordingly:

```
join *p.player rope.cut_vrope( by, cut_y )
    where p.x == rope.x
      and rope.y <= p.y and p.y <= by
      and cut_y < p.y {
    p.update = p.update_falling;
};
join any.cut_vrope { };
```

The rule **R8** is implemented similarly:

```
join *p.player o.bitch where p.is_collided( o ) {
    p.update = p.update_falling;   # create an effect object....
};
```

Although the "Descender" game has some complexity, the methods of the game objects are simple and the join handlers concisely express the corresponding rules in a straightforward manner.

4.3 Comparison with Ruby

As an evaluation, we have implemented our "Balloon" game also in Ruby (precisely Ruby/Tk; Tk library is used for graphics and input handling). In this section, we present the comparison between a join token-based code in Mogemoge and an ordinary O-O code in Ruby.

The resulting game was mostly identical except for the speed and/or look and feel owing to differences in graphics library. The numbers of lines of code is 357 for the Mogemoge version and 436 for the Ruby version. The object definitions are mostly similar for both the versions, but there are large differences in the event handling part.

In this game, most of the interactions among characters are collisions, e.g., an interaction occurs when two characters collide with each other. Therefore, we can factor out collision detection onto a single iteration method (a kind of coroutine in Ruby) as follows:

```
def check_collision( c1, c2 )
    o1s = $obj_list.find_all { |o| o.kind_of? c1 }
    o2s = $obj_list.find_all { |o| o.kind_of? c2 }
    o1s.each do |o1|
        o2s.each do |o2|
            yield o1, o2 if o1 != o2 && o1.is_collided( o2 )
        end
    end
end
```

With the help of a `check_collision` method, an interaction handling code can be written as follows:

```
def check_collision_all
    check_collision( Explosion, Bomb ) do |e,b|
        b.destroy()
        Explosion.new( b.x, b.y, 1.2 )
    end
    check_collision( Explosion, Balloon ) do |e,bln|
        bln.bomb.set_vel( 0, 2 ) if !bln.bomb.nil?
        bln.destroy
    end
    check_collision( Missile, Balloon ) do |m,bln|
        m.destroy; bln.destroy
        bln.bomb.set_vel( 0, 2 ) if !bln.bomb.nil?
    end
    check_collision( Missile, Bomb ) do |m,b|
        m.destroy; b.destroy
    end
    check_collision( Building, Bomb ) do |bld,b|
        bld.destroy; b.destroy
    end
    check_collision( Explosion, Building ) do |e,bld|
        if e.active then e.active = false; bld.destroy end
    end
end
```

Each of the calls to `check_collision` corresponds to handlers in the Mogemoge version, but there are the following differences:

– The kinds of characters participating in each interaction are explicitly described as class names, but their roles in the interaction are not shown; they are expressed as token names in Mogemoge.
– Additional objects participating in the interaction have to be stored in and extracted from the participating object; they are expressed as token arguments in Mogemoge.
– Checks for condition prior to actions are embedded in the code; they are represented as "where" clauses in Mogemoge.
– Unavailability of an object for multiple interactions have to be managed by the code through flags (`active` property in the above code); they are automatically managed by token semantics and `*` symbols in Mogemoge.
– The above Ruby code does not address rule **R3** because it is not a collision. We had to make a balloon and a corresponding bomb to refer to each other via their reference variables and had to maintain this relation manually, as in the following code, in the `destroy` method:

```
def destroy  # a balloon must not refer to a destroyed bomb
  @parent.bomb = nil if !@parent.nil?
end
```

  Using a join token, such references were not necessary and **R3** could be described in a straightforward manner within the handler.

In more complex games with many interactions other than collisions (as in the "Descender" game), the complexity of the Ruby code will increase to a great extent.

In addition, the above code runs nested loops for every combinations of interacting characters for clarity; if the performance becomes a problem, we will have to merge some of the loops, further decreasing the readability of the code. In the case of Mogemoge, we can implement various speedup techniques as necessary without affecting the existing code.

5 Related Works

There are many aspects in our join token framework, so we shall examine the related works with respect to each of them.

Game Scripting Languages. Since our goal is to ease game programming, we first examine the related works that treat game scripting languages. As noted previously, an action game programmer has to control multiple concurrent activities of game characters, along with their interaction in a state-dependent manner.

Micro-threads of Stackless Python [12], [15] allows assigning a dedicated thread to each of the game character objects, so that those objects seemingly act autonomously and concurrently; this view is very natural for game designers. Several websites including [2] recommend this style of game scripting.

Yet, another awkwardness of game programming is that each of the characters may have their own state, and they interact with each other in a state-dependent manner. Some of the game scripting languages, including UnrealScript[14] provides a notion of state; in such languages, game programmers can explicitly describe states in their code. However, "interaction" poses another difficulty, because two or more characters (with their own states) are involved in an interaction.

Coordination Models. From the above discussion, it seems necessary to introduce some coordination model to the game scripting language in order to ease the description of interactions among concurrent activities (game characters).

Linda[7] is a coordination model that uses a "tuple space" as a communication media among concurrent activities. In Linda, both the sender and the receiver of a message (a "tuple" in Linda terminology) are separated in time (at which timing) and space (at which portion of the code). This relieves the programmer from the awkward control of details. However, the demerit of Linda is that the coordination is not symmetric and a bit too low-level; the sender simply emits its tuple, and the receiver must actively select tuples matching its needs.

Join-calculus[4] is yet another coordination model in which the atomic join handler of two or more concurrent activities can be specified. The merit of join-calculus is its high level description and symmetry. On the other hand, the target of the coordination is the thread itself and the two activities are tightly coupled at the join handler; loose coupling of Linda will be more desirable in this respect.

Therefore, we have combined the advantages of both the models and designed a join token framework. Tokens and token pool corresponds to tuples and tuple space in Linda, respectively. A join handler was derived from join-calculus, although our handlers join tokens, not threads. Moreover, we have associated an originating object to each of the tokens in order to ease the object-orientation style of programming, which is common in game programming.

Reactive/Event Programming. Join token can be viewed a kind of reactive and/or event-based programming, which has a long series of history.

First, rule engines, long used for expert systems, have the facility to gather multiple facts (similar to tuples in Linda) and invoke rules when matches are found. Moreover, recent rule engines such as Jess [5], [8] allows Java objects to be used as facts: thus, it can be used as a coordination mechanism for game program written in Java. However, such code will be awkward to write, because every coordination activity must be converted to Jess API calls. Alternatively, it is quite possible to use Jess or a similar rule engine as an implementation device for token pool and take advantage of the efficient Rete[3] algorithm built into it. We will revisit this topic later.

Second, Dynamic aspect-oriented programming (AOP) as in [1], [10] makes it possible to insert join points to existing class code at runtime. It could be used to install callbacks (join points) when an object becomes ready for interaction and would like to wait for one of the other objects to express willingness to participate in an interaction. However, this approach is similar to a thread-based join such as in join-calculus, with its drawbacks, as noted previously. It might also be too general and powerful; a more domain-specific solution would be desirable.

Third, data binding as seen in JavaFX[11] and reactive programming[13] can trigger events when the values of some variables have changed, and appropriate action can be specified. Their major usage for now is to reflect values of some portions to other parts of the system (e.g., user interface components or accompanying objects or so on), but a more flexible setting (for game logic programming) is also possible. However, such customizations might be awkward. Therefore, they might be used as back-end mechanisms to implement join tokens, as in the rule engine case.

6 Discussion

In this section, we discuss the various aspects of join token frameworks and discuss their related issues.

Concurrency Issue. As noted in Section 2, the join token framework described in this paper is targeted to game scripting, in which event ordering and processing order should be strictly controlled by the programmer. Therefore, true concurrency and the resulting non-determinism are intentionally excluded. Perhaps, thanks to this strict ordering property, we got little surprises when debugging the join token-based code.

However, Linda and join-calculus, from which the join token was derived, are actually concurrency coordination models. Therefore, the join token model might also be useful for a true concurrent setup also. We might encounter more "surprises" with such setup, and might require additional coordination (order controlling) mechanisms. We would like to investigate this issue in the future.

Restriction Regarding Token Overwriting. As explained in Section 2, each object can throw multiple tokens with different names into the pool, but can have only one token with a specific name, because the latter throw statement with identical name replaces the previous one. Although this may pose some restriction on the usage of tokens, we have chosen this condition for clarity and simplicity; we have felt no inconvenience so far.

Applicability to More Complex Games. The games we have implemented with Mogemoge and join token so far are fairly simple and small ones, so their applicability to more complex (commercial-scale) games is not yet known.

However, we note that token names are hard-coded in the source code and cannot be changed at runtime. Therefore, although the token pool looks like a global chaos, it is not so in fact; logically, there are many small pools for each distinct token names. When developing large scale games, a token naming convention can be used to safeguard against interference among program modules.

Performance Issue. As noted above, we have only implemented small games using the join token, and so have not encountered any performance problems so far. We expect that this situation might change in the case of larger games.

However, we note that current video games use the majority of their CPU cycles in 3D high resolution graphics, so we guess that CPU cycles used for game logic computation will be negligible even on fairly complex games.

When dealing with the computational complexity of token matching, given that tokens with different names are totally distinct, the number of tokens and handlers with the same name matters. In a naive implementation (which we use for current Mogemoge implementation), with M handlers and N tokens for a specific token name, the computational complexity will be $O(MN)$.

If this becomes a problem, we could incorporate a clever algorithm such as Rete[3]. However, since the Rete algorithm caches the outcome of Boolean guard expressions ("`where`" clauses in join tokens), we need a guarantee that the value of guard expressions does not change without notice. One way to achieve this might be to restrict the guard expressions to use only local values (handler associated objects and their instance variables). We consider a detailed analysis as our future work.

7 Conclusion

Game scripting languages are an effective approach to develop complex games. In the case of action games, the difficulty in development mainly resides in describing complicated interactions among the multiple concurrent behaviors of objects in a state-dependent way.

The join token mechanism that was described in this paper addresses this problem by means of the global token pool and join handlers. This mechanism combines the advantages of the join-calculus and the Linda computational models.

To show the effectiveness of join token mechanism, we have designed and developed an experimental game scripting language called Mogemoge. Mogemoge is an interpreted, prototype-based object language equipped with join token. We have developed Mogemoge using Java and SableCC (a Java-based compiler-compiler framework).

For the purpose of evaluation, we have implemented several demo action games with Mogemoge, including the two described in this paper. We have also compared Mogemoge against an ordinary scripting language through experiments. As a result, the Mogemoge code could be easily derived from the game rules and is comprehensive in general.

At present, we have developed only a few simple sample games with Mogemoge. We would like to evaluate the effectiveness of join token in more complex, realistic games in the future.

Acknowledgement. The authors would like to thank the reviewers of Software Language Engineering Conference 2011 for their helpful suggestions to improve this paper.

Implementation Status and Availability

The implementation of Mogemoge and its sample programs are available at the web site at `http://www.nisnis.jp/mogemoge/`.

References

1. Bornér, J.: What are the key issues for commercial aop use: how does aspectwerkz address them? In: 3rd International Conference on Aspect-Oriented Software Development, pp. 5–6 (2004)
2. Dorf, M.: Need high levels of concurrency? Try stackless Python (July 2010), `http://www.learncomputer.com/stackless-python/`
3. Forgy, C.L.: Rete: a fast algorithm for the many pattern/many object pattern matching problem. Artifical Intelligence 19(1), 17–37 (1982)
4. Fournet, C., Gonthier, G.: A Calculus of Mobile Agents. In: Sassone, V., Montanari, U. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 406–421. Springer, Heidelberg (1996)
5. Friedman-Hill, E.: Jess in Action: Rule-Based Systems in Java. Manning Publications Co, Greenwich (2003)
6. Gagnon, E.: SableCC, an object-oriented comiler framework, Master's Thesis, McGill University (1998)
7. Gelernter, D.: Generative communication in Linda. ACM Transactions on Programming Languages and Systems 7(1), 80–112 (1985)
8. Jess: The rule engine for the Java platform, `http://www.jessrules.com/`
9. Nishimori, T., Kuno, Y.: Join token: A language mechanism for interactive game programming (2011) (under submission)
10. Popovici, A., Alonso, G., Gross, T.: Just-in-time aspects: Efficient dynamic weaving for java. In: 2nd International Conference on Aspect-Oriented Software Development, pp. 100–109 (2003)
11. Slovàcek, V., Macik, M., Klíma, M.: Development framework for pervasive computing applications. SIGACCESS Newsletter 95, 17–29 (2009)
12. Stackless Python, `http://www.stackless.com/`
13. Susini, J.-F.: The reactive programming approach on top of Java/J2ME. In: Proceedings of the 4th International Workshop on Java Technologies for Real-Time and Embedded Systems, pp. 227–236 (2006)
14. Sweeney, T.: UnrealScript language reference, `http://udn.epicgames.com/Three/UnrealScriptReference.html`
15. Tismer, C.: Continuations and Stackless Python. In: Proceedings of the 8th International Python Conference (2000)
16. Ungar, D., Smith, R.B.: Self: the power of simplicity. In: OOPSLA 1987, pp. 227–242 (1987)

Reusing Pattern Solutions in Modeling: A Generic Approach Based on a Role Language

Christophe Tombelle[1], Gilles Vanwormhoudt[1,2], and Emmanuel Renaux[1]

[1] Institut TELECOM
{tombelle,vanwormhoudt,renaux}@telecom-lille1.eu
[2] LIFL/CNRS - University of Lille 1 (UMR 8022)
59655 Villeneuve d'Ascq cedex - France

Abstract. Design patterns are a means to capture and reuse good practices and working solutions acquired by experts in various domains of specification and design. A lot of work has been done to try to express the solution part of design patterns in a computer understandable language but most of it is centered on the UML, especially on the Class diagrams. Model engineering techniques make it easier to design domain-specific languages and we think that design patterns can be identified in any engineering domain. So, a language able to describe accurately design pattern solutions for any specification or design language, along with its reuse process, would be of great interest. This paper presents Gipsie, a specification language that approaches, at an abstract level, this goal through the notion of generic roles, i.e. parameterized by any metamodel.

1 Introduction

Whatever be the language of specification and design, some good practices and working solutions to a given problem can be identified by experts. Design patterns [7] are a mean to capture and reuse such valuable items so that they can be profitable to various design teams. In this paper, we are particularly interested in the specification of small reusable pieces of model expressing the solution part of design patterns with the goal of incorporating such pieces into larger models.

With Model-driven engineering promoting the building and use of a diversity of Domain-Specific Modeling Languages [10], appears a need to represent as accurately as possible the solution part of design patterns for any language. A language would be accompanied with its set of common design patterns.

In fact, several works [18,4,1] have been proposed to express the solution part of design patterns in a computer understandable form. Unfortunately, most of these works were designed to specific domain, defined by a specific metamodel (mostly UML). Moreover, they tend to use a specific language to express the design patterns. As a result, the application of design patterns over new metamodels has been obstructed, since each of them requires to develop a new specific support. If one wants to quickly establish design patterns support for new domain-specific languages, generic methods and tools are needed.

This paper presents the metamodel of Gipsie, a GenerIc Pattern SpecIfication languagE aiming at describing the solution part of design patterns for any language. In our proposal, a design pattern solution represents an abstraction of model fragments and is primarily described in terms of roles. The process of reusing a design pattern solution in a target model is also considered and we propose a metamodel and a process for a *pattern realization* (or *binding*) i.e. a particular application of a given pattern solution to a given target model. Compared to other existing approaches, Gipsie limits the design effort by not requiring to specify new metamodels or to adapt existing ones for a specific modeling languages.

Section 2 presents some motivations for a language capable to specify accurately the solution part of design patterns. Section 3 presents our proposition for specifying pattern solution. Section 4 describes how a pattern solution can be applied to a target model. Tools supporting the approach are described in Section 5. Section 6 presents related works prior to conclude with Section 7.

2 Motivating Examples

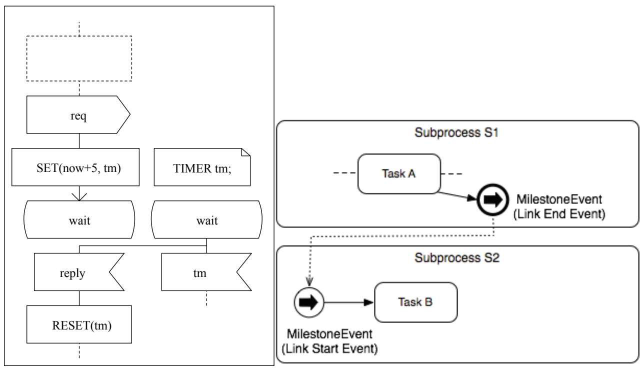
2.1 Existence of Design Patterns in Any Design Language

A lot of work has been done to describe design patterns themselves as models but most of it is centered on the UML, especially on the Class diagrams. However, some good practices can be identified in any domain, language or modeling tool. For instance, we find the need of patterns in the domain of communicating processes [2] or the domain of workflow [21]. In the following, we will use examples of patterns from these two domains to motivate some general requirements for patterns realization in any modeling language. In the first domain, one can identify the 'Timeout on no answer' design patterns for the SDL specification language [8]. The solution part of this pattern is pictured in the left part of Figure 1. This pattern recommends the use of a Timer (*tm*) when a request (*req*) is sent to a peer entity while expecting an answer from it. If the answer Signal (*reply*) is not received within the delay specified by the timer, a timeout Signal (*tm*) allows the automaton not to wait indefinitely for the answer in the same state (*wait*). In the second domain, we find the 'Milestone' pattern [21]. This pattern is shown in the right part of Figure 1 using the Business Process Modeling Notation (BPMN). It expresses that the completion of an activity (Task A) in one Sub-Process[1] is required before an activity (Task B) in another Sub-Process can start. In the pattern solution, a Link End Event (MilestoneEvent) that follows task A is used to trigger a corresponding Link Start Event that precedes Task B and an Association link is used to reinforce their relationship.

2.2 Requirements for a Generic Pattern Specification Language

In design patterns described in natural language [7], the class diagram drawn in the *solution part* is willing to be a description of the general solution brought by

[1] whatever are its predecessor and successor tasks in the Sub-process.



'Time out on no reply' SDL pattern **'Milestone' BPMN pattern**

Fig. 1. SDL and BPMN patterns

the pattern. Indeed, such a class diagram captures the structural constraints of the pattern solution by using regular model elements of the UML metamodel. To get a description of the pattern solution reusable in case tools, we may think about directly transposing this class diagram as a computer understandable model and manipulate the result as a prototypical instance of the pattern solution. However, this prototypical approach puts too many constraints as we will see hereafter.

Unneeded Constraints Issue: First of all, the prototypical approach forces some model elements to exist and to have all properties specified by their metaclass whereas the pattern tells nothing about most of them. In addition to these constraints relative to the metalevel, other unneeded constraints are due to model elements themselves. Sometime the value of a model element property may be a requirement for the pattern but it may often be a detail (especially name properties). In 'Time out on no reply' pattern, the state and signal names are just a detail: the pattern does not require the state and the answer signal to be respectively called *wait* and *reply*. Another unneeded constraint is also the position of an element in an ordered feature. For instance, the 'Time out on no reply' pattern does not require the timer setting to follow strictly the output signal. The only requirement is that these elements belong to the same branch.

Ideally, a pattern description should be a fragment specification only including elements that are part of the pattern intent. It should not include unneeded elements so that no unneeded constraints are imposed to the models targeted for its applications.

Identity Requirement: Some patterns impose some constraints on properties, such as having the same value as another property. In our pattern examples, this first requirement is shown by the name of the timer (resp. Link End Event) and the name of the timeout signal (resp. Link Start Event) : it must be the same for ensuring consistency. In the same way, often, a pattern requires same model fragments to be contained -or just referred to- by distinct model elements such as methods with the same signature in distinct classes. An illustration of this second requirement is given by the 'HeartBeat' pattern (see Figure 5) : the action for the initial setting of the timer and its setting after sending the live signal must have the same parameters. The pattern description should enforce these identity constraints.

Validity Issue: A pattern prototype usually includes model elements that are not always required by the pattern. Their presence is only required for the sake of model conformance to its metamodel. In a description of 'Time out on no reply' (Figure 1), the presence of an initial state is not required because it plays no role in the pattern. A prototypical approach would probably specify the initial state (see the metamodel Figure 4). A similar observation can be made for the 'Milestone' pattern where the Subprocess has no initial activity and the Task B is not linked to any predecessor. Indeed, a pattern need not specify a model conforming to the complete metamodel, only the target model must do so.

Need for Abstraction: A pattern intent can often be achieved by different means. A common intent is to specify elements of the pattern without being too accurate about their metaclass that even may be abstract. In 'Timeout on no answer', one need to specify that the *req* output signal can follow either an initial state or an input signal, that is a branch element (abstraction represented by a dotted box in Figure 1). A pattern specification language should support this kind of abstraction to potentially allow more candidates playing a role.

Another common intent is to specify that a model element should be present in a container without specifying the containment relation nor if this containment is direct or indirect. 'HeartBeat' shows the initial set-up of the *tm* timer using a 'SET(Now+5, tm)' action. The container of this action is the process but the containment is indirect and multiple direct containers are possible such as the initial state or any input signal of the target process.

2.3 Requirements for Pattern Application

The issue of describing a design pattern solution can not be addressed without considering its application to a target model. This application can follow various scenarios.

When applying a pattern to an existing model, some elements specified by the pattern may already be present in the model, while others have to be inserted. If the target element exists, it is just identified as playing the role. If it does not exist, it must be created then inserted in the target model. Following the application context, existing and inserted elements are not always the same. Extreme situations are a model where the pattern is already present (all elements must be identified as playing the role) and an empty model (all elements must be inserted). To accommodate all these situations, the application of pattern should provide flexible mechanisms to select which elements are matched and which ones are inserted.

At application time, a major requirement is the ability to apply a same design pattern multiple times in different ways or to apply different design patterns to a same target model. This implies that a same target model element should be able to play multiple roles so that different responsibilities are documented separately.

Once applied, pattern-aware tools should keep track of the pattern application. For the sake of model exchangeability, non pattern-aware tools should still be able to load the target model.

In the next section, we propose Gipsie, a generic pattern specification language that meets these requirements.

3 Role Modeling for Patterns

3.1 Our Approach of Design Patterns

In our approach, a pattern solution is specified by a *role model* that complements constraints expressed by the metamodel. A role model specifies constraints on the target model with a structure of roles. In our case, a role specifies that an element should exist (existence constraint), should be in relation with such other (relationship constraint) and some of its properties should respect some values (values constraint)[2]. Whereas a metamodel specifies meta-elements for defining the structure of its compliant models, a role model specifies role elements for abstracting model fragments in these compliant models. A target model containing a pattern solution must comply with its metamodel and a *fragment* of this model must comply with the corresponding role model.

Our approach is depicted in Figure 2. This figure shows the structure of the two involved modeling spaces (TargetModel space and Pattern space) as well as their relationships using one of our previous example. The TargetModel space is considered similar to the OMG multi-level metamodeling architecture (TargetModel, Metamodel, Metametamodel). The Pattern space is contributed by our approach. In this space, we provide the metamodel of Gipsie to express role structures (Role models) capturing pattern solutions. As Gipsie is willing to be generic, its metamodel will not refer to any specific metamodel but only to the metametamodel[3]. This enables the parametrization of role model elements with elements of a target metamodel : for instance, a role model capturing the 'Timeout on no answer' pattern solution will have roles that refers to *Input* and *Timer* metaclasses of the SDL metamodel. In a role model, this parametrization complements the structure of roles : it serves to state the meta-type expected for elements playing a role in a target model.

In our approach, one application of a pattern solution to a target model is specified by a realization containing binding elements. Each binding element is associated to a role element and specifies how the role is concretely realized in the context of the target model : either it is bound to a target model element or it is inserted as a new element in the target model. Note that a target model element may play multiple roles but is always instance of its metaclass. The

[2] This notion of role differs from [16] and UML where a role captures a behaviour in a collaboration of objects.

[3] In our experimentation, we have used Ecore Metamodel of EMF.

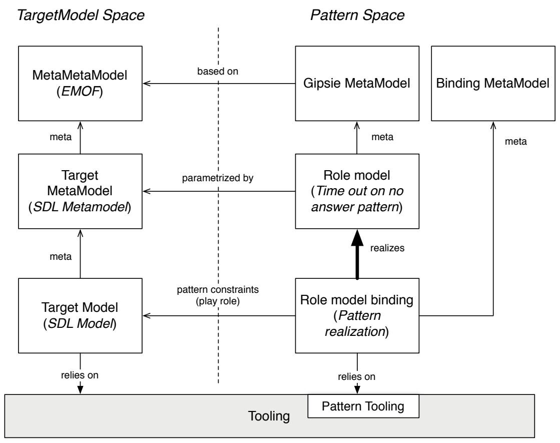


Fig. 2. The pattern big picture

instance of relation should not be confused with the *plays role* relation. In the following, we detail each component of the Pattern space.

3.2 The Gipsie Language

Figure 3 shows the metamodel of Gipsie. Role is the main concept of Gipsie metamodel. A role has a name and specifies a meta-element that defines the kind of element able to play the role. There are different kinds of roles: an object role *RObject* specifies a metaclass (i.e. an EClass instance), a link role *RLink* specifies a metalink (i.e an EReference), a property role *RProperty* specifies a meta-property (i.e an EAttribute), a value role *RValue* specifies a literal. Such a fine grain structuring is justified by accuracy and expressive power targeted for Gipsie[4].

Figure 4 depicts the instantiation of this metamodel for the 'Time out on no answer' pattern[5]. The structure of this role model reflects the structure of the pattern solution given by Figure 1 except that role elements have replaced concrete elements. All these role elements specify what it is expected from any target model to comply with the pattern. For instance, the *wait* object role linked with the *reply* object role assumes the existence of a *State* element linked to an *Input* element in any target model. The use of role elements enables us to express expected fragment of models without having to specify unneeded constraints imposed by the metamodel.

Figure 4 also gives the target metamodel used to parameterize the roles of the pattern[6]. This target metamodel describes simple SDL processes with *Process, Action, State,...* metaclasses, instances of EClass.

[4] A formalization of the approach is available at the Gipsie site (see footnote 11).

[5] The notation for role models is adapted from [17].

[6] The parametrization of a role element with a particular meta-element is noted between < and > character. For instance, req:<Output> means that the *req* object role is parametrized by the *Output* metaclass.

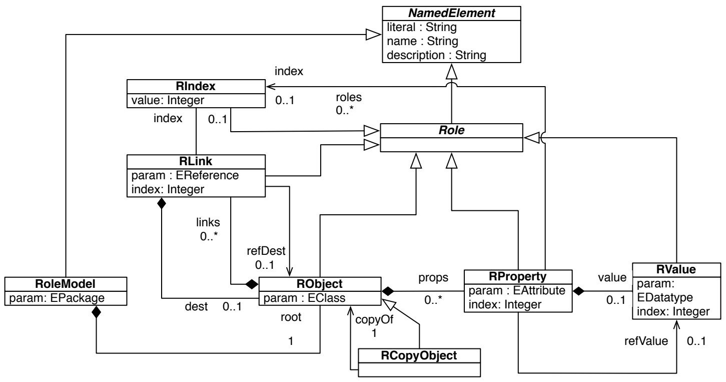


Fig. 3. Gipsie metamodel

In the following, we give some explanations about the concepts provided by the Gipsie metamodel.

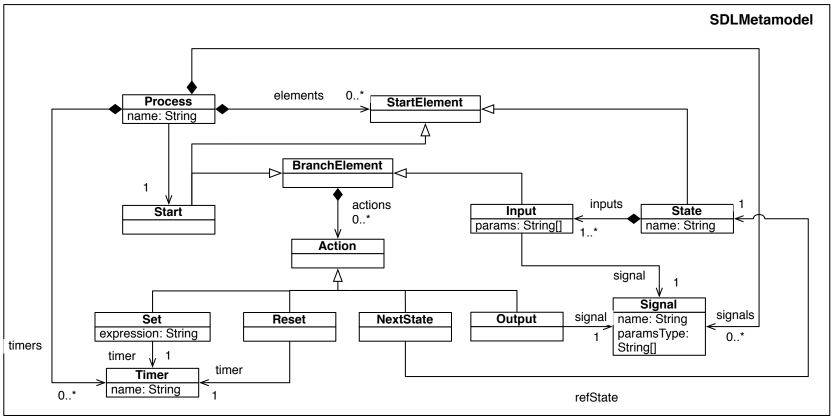
Role Model: This concept specifies the pattern solution and references the target metamodel (i.e an EPackage) and contains the top level object roles. A pattern solution has a hierarchical structure: an object role can contain link roles that can contain other object roles.

Object Roles: The presence of an object role in a pattern solution specifies that an element should exist in the target model to play the role. In Figure 4, the *wait* role is represented by an instance of *RObject* which is parameterized by the *State* metaclass to specify that only a state must play this role. This role puts no constraints on the name property of the State element.

Abstract Object Roles: An object role may refer to a base metaclass even if it is abstract. This allows any instance of a concrete subclass of this metaclass to play this role in the target model. An example of abstract role in Figure 4 is the *be* role which specifies the *BranchElement* metaclass. Along with abstract link roles, this is a way to meet the *need for abstraction* cited in the previous section.

Property and Value Roles: In order to constrain a property, a property role (*RProperty*) must be added to a pattern object role. In Figure 4, a *RProperty* is added to the *st:<Set>* object role in order to specify that a constraint exists on the 'expression' attribute. The nature of the constraint is specified by a value role (*RValue*) added to the property role. A value role specifies a value for a property role. In order to be generic and usable with properties of any data type (EInt, EChar or any specific enum type), the value is specified with a string literal. For instance, the expression property of *st:<Set>* should be valued 'Now + 5'.

Value Sharing: In the intent of 'Time out on no reply', the name of the timer and the one of the timeout signal is not mandatory. What really matters is that these names must be the same for understanding. This can be enforced in



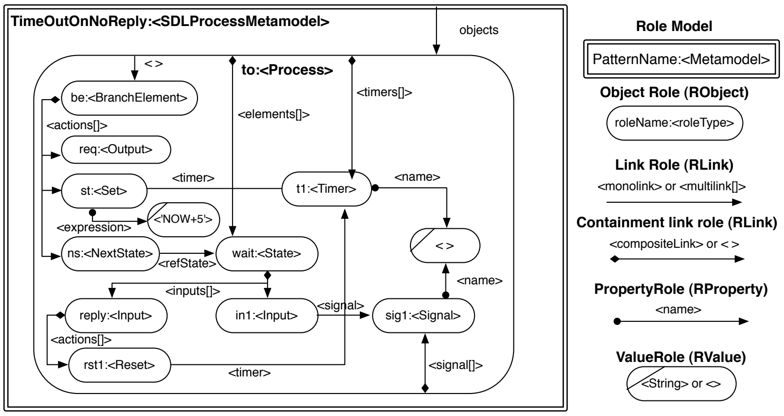


Fig. 4. Metamodel for simple SDL Process and 'Time out on not answer' described with Gipsie

Gipsie, with *t1* and *sig1* attached property roles sharing the same value role. This feature is called 'value sharing'. In this case, the value role literal is left unset (<> in the figure) to specify that the value of the name does not matter as long as it is the same for the two objects.

Link Roles: A link role is contained in its source object role and points to its destination object role. In the target model, this specifies that a link should exist between objects playing roles specified by linked roles. A link role specifies with a meta-link the type of link that must exist between these objects. For instance, a <*refState*> link role between *ns* and *wait* object roles specifies that the target of the NextState playing the *ns* role should be the state playing the *wait* role. The meta-link is oriented from the containing role to the destination role. The opposite meta-link, if any, is implicitly specified by the metamodel.

Abstract Link Roles: An *RLink* that does not specify any meta-link is not an incorrect pattern element. For the target model, this specifies that object playing the source role must contain the object playing the destination role, following any containment feature. Of course, metaclasses must be specified in both source and destination object roles. This linking constraint intersects with ones imposed by the metamodel, so that such a containment feature should exist, in order the pattern to be valid. If such a feature does not exist between the source and destination metaclasses, an indirect abstract containment is specified. This feature meets the 'Need for abstraction' discussed earlier. It is illustrated by the containment link from the *to:<Process>* role object to the *be:<BranchElement>*.

Link Index and Equality: A link role can have an index to constrain the position of the linked object in the target model. For instance, if a pattern requires the parameter of an operation to be the first one, an index value of 0 will specify this requirement. If the first position is not required, the index is not specified. Indexes may be shared among multiple link roles. This constrains the destination objects position to be the same.

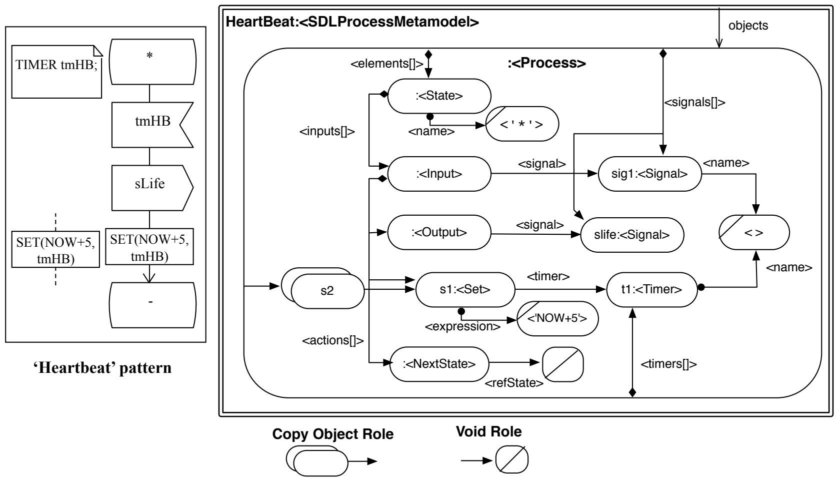


Fig. 5. Copy object role illustration in 'HeartBeat'

The *copy* Object Role: In some patterns, objects playing roles must share the same constraints. An illustration can be drawn from design patterns for an object oriented programming language which commonly describe method redefinitions. The method and its parameters are a model fragment that must occur a first time in the base class and a second time in the redefining class. In the target model, this constrains the two operations to be a copy one from the other. The 'copy object role' feature of Gipsie (*RCopyObject*) is dedicated to this issue. It is a role that refers to another role and constrains the bound object in the target model to be a copy of the other[7]. In Figure 5, we show the use of this feature in the HeartBeat pattern for SDL (left part). This pattern generates a periodic ‘alive’ signal to a peer entity. After the initialization or an input signal, the timer (*tmHB*) is set to the duration of an interval and after the timeout, a signal (*sLife*) is generated and propagated. In the corresponding role model (right part), the s2 role is specified as a copy of *s1:<Set>* to constraint the two timer setting of the pattern to have the same parameters.

3.3 Role Model Validation

Link and object roles specify graphs of any shape. A pattern designer can refer to any metaclass in an object role and to any meta-link in a link role. This can lead to invalid pattern descriptions. Editing and validation processes ensure consistency for pattern descriptions. The role model validation process starts from the root object role and propagates first to sub-object roles through containment link roles. Non containment link roles are then validated, ending with property and value roles.

The validation process uses meta-elements specified by the different roles. Metaclass for the role (e.g. *to* role in Figure 4) may be chosen with no other constraints than being part of the target metamodel. The meta-link specified in a link role must be present in the metaclass specified by its origin object role (*signal* meta-link is present in the *State* metaclass specified in *wait*). In the SDL process metamodel, *signal* is actually a meta-link of the *Input* metaclass. Its type (*signal* type is *Signal*) must be the metaclass specified in the destination object role (*sig1* metaclass is *State*). The metaclass specified in a non root object role must be assignment compatible with the type of the meta-link of its containment link role.

4 Pattern Application with Role Modeling

4.1 Our Approach of Pattern Application

At application time, following the target model context, a role will be bound to an existing target element or inserted. In case of binding, an existing target element is just identified to play the role. In case of insertion, a new model element is created then inserted into the target model and finally bound too. If an object role features no property roles, the object is created with metamodel specified defaults, else property roles are applied. In the same way, if the object role features link roles, they are identified to existing links or inserted.

While describing a given pattern consists in writing a model of Gipsie metamodel, applying this pattern to a given target model requires the manipulation of application-time entities which represent the pattern realization (see fig. 6). We call these entities ‘bindings’ and propose to define them in a ‘role model

[7] ’copy’ means that sub objects are deep copied while referred objects are shallow copied.

binding'. A 'role model binding' maintains links both to the role model capturing the pattern solution and to a target model. Figure 6 gives an example of role model binding to apply the 'Time out on no answer' pattern solution on a target model and the model resulting from this application.

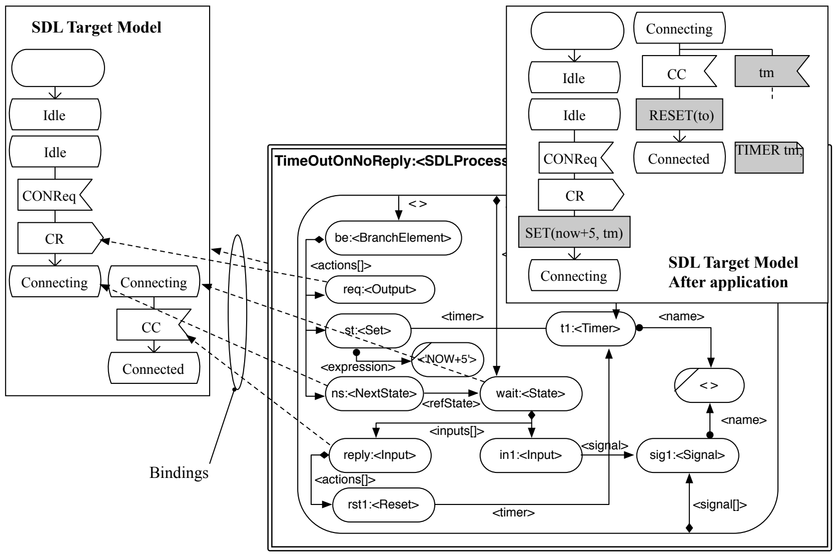


Fig. 6. Application of the 'Time out on no answer' pattern to a target model

4.2 The Binding Metamodel

Figure 7 depicts the binding metamodel and its instantiation for the previous example. To simplify the binding operation, this metamodel is minimized. In particular, it does not include concept for the binding of a role link (resp. role property) since its correspondence with a target link (resp. target property) can be deduced from the binding of roles at its ends (resp. the bound owner role). We explain the concepts of this metamodel hereafter :

The Role Model Binding: A role model binding (*RoleModelBinding*) references a role model and a target model. It contains a tree of object bindings. The structure of this tree is computed automatically (by the *createBinding()* method) to be isomorphic to the roles and the containment relationship between roles (see right part of Figure 7 as example).

The Object Binding: An object binding (*BObject*) is a realization of an object role. It holds a *role* link to the role it realizes. It also keeps track of the application of an object role to a target object with a *target* link to the target model object (*EObject*). A *targetContainer* link must be specified for root object bindings intended for insertion. A *targetContRef* is also useful if multiple containment meta-links exist in the container metaclass. *targetContainer* and *targetContRef* must also be specified if the object role is contained in an *abstract link role*. Because the metaclass associated with its object role may be an abstract metaclass, a concrete metaclass must be chosen among the subclasses. This allows the object role to be bound to any model element instance of one of its concrete subclasses (eventually inserted one), providing for variability in the application. The object binding holds this concrete metaclass as *concreteMetaclass*.

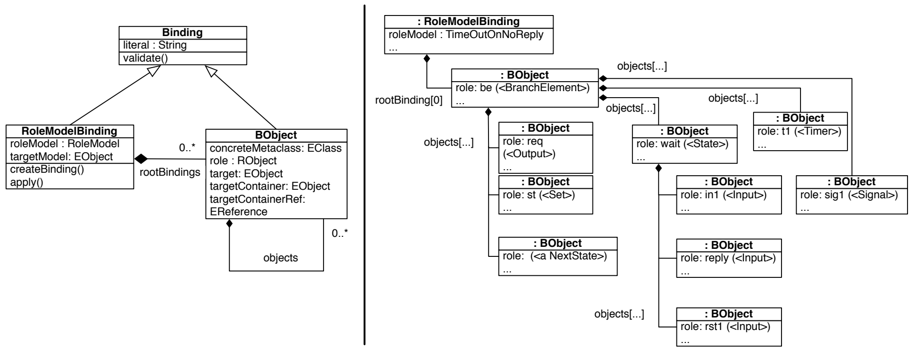


Fig. 7. The binding metamodel and one binding model

4.3 The Application Process

The application or binding process consists in choosing how to apply a role model to a target model. The binding model is a trace of these choices. It is checked through a validation process based on a set of constraints. The binding process is semi-automatic. The manual part is guided by the validation process.

In the first phase, a binding model is derived from the structure of the role model. In this model, binding elements are created with a reference to the role they realize and are linked according to the containment relationships of roles. At this stage, the role model is not bound to a target model and the validation process reports the binding model as 'unbound'.

The purpose of the second phase is to define what must be inserted (unbound roles) and where (in objects bound to roles). This phase is guided by the validation process and begins by designating a compliant target model in the RoleModelBinding. This changes the status of the binding model to 'bound' instructing the validation process to report about the applicability of the role model. A valid binding model would mean the role model is applicable i.e. ready for the third phase. However, at this stage, it is no longer valid and requires minimal editing : object roles such as be:<BranchElement> which are contained via an abstract link role must either be bound to a target object either have their target container and container reference defined. Object roles not bound to a target object are intended for creation and insertion of a new object in the target model. If the role specifies an abstract metaclass, the choice of a concrete subclass is required in the object binding. When a constraint is violated the user corrects the binding model. Bound target elements must comply with meta-elements specified by their roles. Copy constraints between roles are checked if roles are both bound to the target model. If a value binding does not impose a value but is shared by multiple property bindings related to bound target objects, the validation reports these properties in existing objects should have the same value.

The third phase applies the roles to the target model resulting in the insertion or modification of elements. It is itself divided in two steps :

– In the first step, new elements are created. This is achieved by traversing the tree of binding elements. For an unbound role, a new object is created and initialized with its specific property values. For an unbound copy object role, the target object referenced by the other role is cloned. In both cases the role element is bound to the newly created or cloned object. After this phase, all roles are bound to an object.

– In the second step, the target model is updated. This is done by a second traversal. Several operations are performed during this traversal. Firstly, previously created objects are attached to their container. Secondly, link roles with no corresponding link in the target model are identified thanks to an analysis of links existing between target objects playing roles. For such link roles, a corresponding link is added between the target objects. If a link index is specified in the role model, it defines the insertion point. Lastly, properties of target objects are modified according to their value role or value binding.

After this final step, the target model has all the elements to comply with the pattern solution. However, the target model holds no references to the binding model nor to the role model so that it can still be manipulated by tools unaware of role and binding models. The role model is also independent of any target model. As discussed in the next section, a role model can be applied multiple times to a same target model using different bindings. Multiple role models can also be applied to a same target model. However, multiple bindings may introduce conflicting constraints. The user is the only judge of the utility of applying a role model.

4.4 Capacities in the Application Process

Figure 8 shows some of the main capacities provided by the application process using the example of a role model parameterized with a metamodel for simple class-based models. This role model expresses a simple structural pattern for wrapping elements. We can find such a pattern as part of larger patterns like Adapter or Decorator. It is composed of a role for the 'wrapper' class, an abstract role for the 'wrapped' classifier and a role for the property that links the role players. By being abstract, the 'wrapped' role gives the possibility to select a class or a datatype for its target element. The table in lower part of the figure represents several applications of this role model[8,9], exhibiting the following capacities :

– The capacity to apply a role model to various base models including an empty one (illustrated by row 1) or a target model already containing the pattern (illustrated by row 6 and 7) or some of its parts (row 5). This is made possible because a binding can vary according to the target.

– The capacity to get distinct results when applying a role model to a base model thanks to distinct bindings and/or distinct concretizations of abstract roles. In the table, this capacity is illustrated by rows 2 and 3 which target the same base model but lead to different resulting models.

– The capacity to apply a role model multiple times to a base model using different binding models. Multiple applications of a role model can be either independent or chained. The latter case occurs when elements introduced by an application are reused for another applications. Rows 2 and 3 give an example of two independent applications while the couple composed of rows 3 and 6 illustrates chained applications. For this couple, we can observe that the result of the first application (row 3) may serve as an input base model for the second application (row 6) and that roles of the latter application are played by previously inserted elements. For multiple applications, we have ordering properties similar to those described in [14].

– The capacity to bind several object roles to a same target element within the same application. Illustration of this capacity is given by row 3 where the existing *c1* class plays both the 'wrapper' and 'wrapped' roles to wrap the same type of elements.

– The capacity of an existing element to play different roles in chained applications. This is illustrated by rows 5 and 7 considered as a chained application. In application of row 5, the *c2* class is inserted as a player of the 'wrapped' role while this class plays the 'wrapper' role in application of row 7, resulting in a chain of wrappers.

The capacities presented above show the flexibility provided by our approach to address a wide range of situations. Thanks to these capacities, a pattern solution can be reused multiple times in the same target model or be reused across several target models but it can also be incorporated in target models with some form of variability for inserted elements.

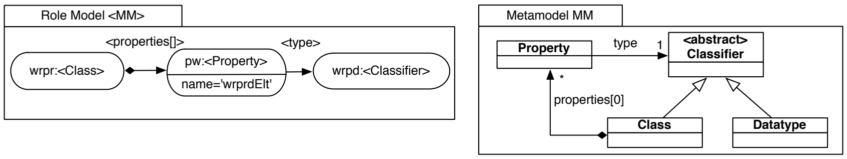
5 Tool Support

To evaluate the proposed approach, we have built and integrated some tools[10] into the Eclipse environment by relying upon its modeling framework (EMF).

[8] Base and result models of classes are represented as instances diagrams of the MM metamodel.

[9] To distinguish existing elements from added ones in a resulting model, they are represented by dotted box.

[10] see http://www.telecom-lille1.eu/people/tombelle/gipsie/gipsie.html



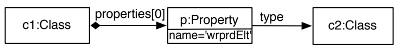
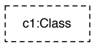
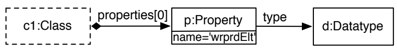
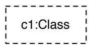
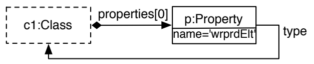
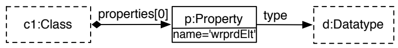
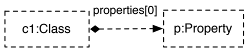
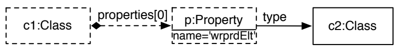
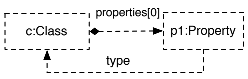
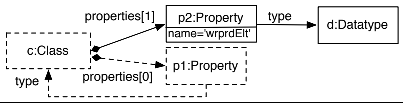
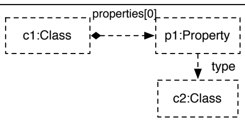
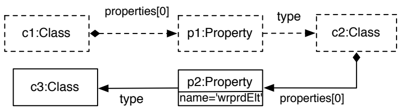
	Base Model	Binding	Abstract Role Concretization	Result Model After Application
1	Empty	wrpr->void pw->void wrpd->void	wrpd=>Class	 c1:Class properties[0] p:Property name='wrprdElt' type c2:Class
2		wrpr->c1 pw->void wrpd->void	wrpd=>Datatype	 c1:Class properties[0] p:Property name='wrprdElt' type d:Datatype
3		wrpr->c1 pw->void wrpd->c1	None	
4	c1:Class d:Datatype	wrpr->c1 pw->void wrpd->d	None	 c1:Class properties[0] p:Property name='wrprdElt' type d:Datatype
5		wrpr->c1 pw->p wrpd->void	wrpd=>Class	 c1:Class properties[0] p:Property name='wrprdElt' type c2:Class
6		wrpr->c pw->void wrpd->void	wrpd=>Datatype	 properties[1] p2:Property name='wrprdElt' type d:Datatype c:Class p1:Property type properties[0]
7		wrpr->c2 ref->void wrpd->void	wrpd=>Class	 c1:Class properties[0] p1:Property type c2:Class c3:Class p2:Property name='wrprdElt' type properties[0]

Fig. 8. Several applications of a role model to various base models

Figure 9 shows the suite of tools and their relationships. Role models and binding models can be edited thanks to smart editors existing in two versions : a tree-based one and a text-based one. These editors offer powerful completion assistance to parameterize role model elements. They also check the consistency of role and binding models as described previously. The Model2RoleTransformer tool allows to transform any model conforming to a metamodel into a role model targeted for this metamodel. This tool is useful to quickly get a first version of a role model from existing models so it can be further abstracted. The Concretizer tool goes in the other direction. It allows to create a fragment of model from a root role which has no abstract links[11]. Thanks to this last tool, designers can quickly test the specification of their role model by concretizing some parts.

[11] this creation is made possible by relying on defaults from the role model and defaults from the targeted metamodel.

When role models are ready-to-use, they can be put into a repository of role models. Within this repository, role models are automatically organized into groups related to their target metamodel. Finally, a tool for applying available role models to target models as explained previously is also provided.

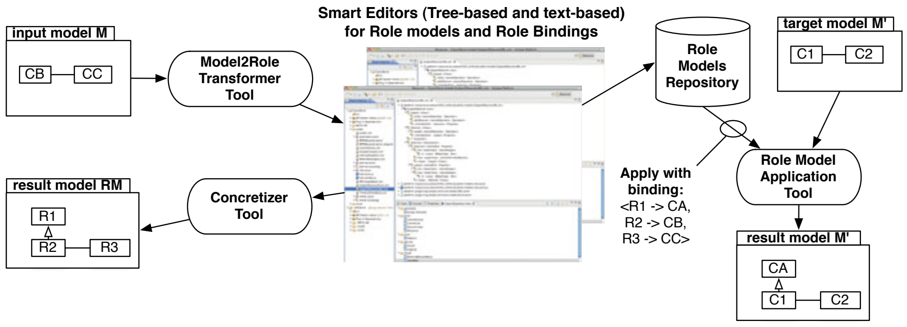

Smart Editors (Tree-based and text-based) for Role models and Role Bindings
input model M
CB
CC
Model2Role Transformer Tool
Role Models Repository
target model M'
C1
C2
Apply with binding: <R1 -> CA, R2 -> CB, R3 -> CC>
Role Model Application Tool
result model RM
R1
R2
R3
Concretizer Tool
result model M'
CA
C1
C2

Fig. 9. Tools for designing and applying role models

These tools have been exploited to conduct a preliminary experiment with several modeling languages : UML class and activity diagram, Ecore, SDL, Statechart, BPMN and Petri Net. Table 1 presents a subset of role models designed for both UML, Ecore[12] and SDL metamodels[13]. These role models correspond to pattern solutions found in [7] for UML and Ecore and SDL patterns found in [2,8]. For each role model, we give the number of main constructs used in each pattern : object role (abstract role), link role (abstract link role), role property, copy role. The numbers given for a row correspond to the set of elements required to comply with a pattern solution. From the provided values, we can see that experimented patterns have various complexity and exhibit different needs in terms of necessary constructs. Several remarks can be made from this experiment.

First, the diversity of experimented languages, even if their number is limited, confirmed the genericity claimed by our approach. In particular, we validated that our approach can be applied to small and huge modeling languages like UML as well languages modeling structural and behavioral parts of a system. In future work, we plan to investigate categories of modeling languages not studied here in order to identify patterns that are difficult to define with our approach.

Secondly, one can see from the table values that each pattern required to specify a significant amount of constructs. However, it is important to note that the number of constructs is minimized compared to those needed for expressing the pattern in a target model. As a result, our approach does not require additional

[12] We designed the same set of patterns for UML and Ecore in order to compare their differences in terms of needed constructs.

[13] Role models for all GOF patterns are available on the Web site.

Table 1. Some of the experimented patterns for UML, Ecore and SDLs

Metamodel	Role model	$\text{RObject}_{abstract}$	$\text{RLinks}_{abstract}$	RProps	RCopy
UML (Ecore)	AbstractFactory	16 (10)	14 (9)	2 (2)	0 (0)
	Adapter	15 (13)	9 (5)	4 (3)	0 (0)
	Bridge	13 (11)	6 (5)	7 (6)	1 (1)
	Builder	14 (11)	11 (9)	5 (2)	1 (1)
	ChainOfResponsability	8 (6)	5 (4)	1 (1)	1 (1)
	Command	17 (13)	13 (10)	2 (2)	1 (1)
	Composite	36 (31)	22 (20)	6 (8)	1 (1)
	Decorator	16 (14)	7 (6)	9 (9)	0 (0)
SDL	Heartbeat	9	2	2	1
	BlockingRequestReply	15_2	7	2	0
	TimeOutOnNoReply	11_1	4_1	3	0
	RepeatedEvents	14_2	6_1	10	0
	TimedRepeatedTrials	35_5	15_1	13	0
	TimerControlledRepeat	31	13	8	0
	Timer	23	10	4	0
	Watchdog	13_1	7	3	0

efforts for specifying patterns. There is just a learning curve to understand all subtleties of the proposed approach but this effort must be balanced with the benefit of reusing a role model across multiple target models and avoiding the effort of incorporating elements of a pattern solution by hands.

In this experiment, we also noted that the complexity of a pattern specification with a role model depends on the target metamodel. This is indicated by the values given for UML and Ecore on the same set of GOF patterns. The difference between values for roles and links comes from the fact that inheritance, relationships between classes and cardinalities are defined using simpler concepts in Ecore in comparison with UML.

This experiment also highlighted that the abstraction features supported by our approach only make sense for some target metamodels. In our experiment, we were only able to exploit abstraction features for SDL (abstract roles parameterized by BranchElement or Action), not for UML or Ecore class diagram. In general, it appears that concepts like abstract role object and abstract role link are more appropriate for metamodels that allow to construct models from a hierarchy (generalization or composition) of concepts. At the opposite, the concept of RCopy role were mainly relevant for GOF patterns and is generally more adapted to metamodels that include refinement relationships.

Finally, a few shortcomings have been identified for this experiment :

– Designing roles model requires to have a good knowledge of the target metamodel since proposed constructs must explicitly reference existing metamodel classes, links and properties.

– Role models provide limited support to reduce the complexity of pattern solution. The only mechanism available for this purpose is the RCopy role concept but, as discussed previously, it is only relevant to specific modeling language.

– Competition between role model and target model exists sometimes. This situation occurs when an element intended for insertion competes with an existing element. For instance, a binding for the base model of row 7 in figure 8 with values ($wrpr \rightarrow c1$, $pw \rightarrow p1$, $wrpd \rightarrow void$) would cause a competition between c2:Class and a new inserted classifier. Such situation generally requires update of an existing link for enabling insertion (here *type* link) and removing of existing element but at present time, we do not allow this modification to keep the target model compliant with another role model previously applied. In this case, we let the user edit the target model if he want to apply the role model.

6 Related Works

Initial works on representing pattern solutions at the modeling level are [18,4,1]. Some works have exploited the role concept for expressing pattern.

The idea of role modeling has been introduced by Reenskaug [16] for providing a general approach to modeling object and object collaborations. The work of [17] is the first to specifically address the use of roles for pattern design and integration. In this work, a pattern is represented by a role model corresponding to a set of interconnected roles where role represents the view some objects in the collaboration hold on another object playing that role. Composition between role models and constraints between roles is also provided. In UML standard, the concept of parameterized collaboration is used to represent the structure of a pattern solution in terms of roles. The application of a pattern solution into a particular context is represented by collaboration usages and bindings of roles to classes. While our approach gives the ability to specify and apply patterns for any modeling languages, these approaches are only suitable for object models. Another difference is that our approach offers pattern-specific features to address abstraction, identity, sharing in pattern specification and integration.

The use of roles as a metamodeling technique for specifying pattern solution has been applied in two works but they do not address pattern application. France and Al. [6] propose to specify UML patterns as specialized UML metamodels. Pattern roles are specified as subclasses of UML metaclasses and are related to each other through new meta-associations. This work also defines rules to establish the conformance of a particular UML model to a pattern metamodel using a binding relationship. Compared to this work, our approach differs by not using a specialized metamodel for expressing the structural properties of the pattern solution but another way relying upon an role-based language which is less constraining than the notion of specialized metamodel (see discussion in 2.2). This work does not claim either the ability to specify model fragments and to be generalizable to any metamodel.

A more generic proposal in the context of pattern detection is EPattern [5], a language to specify patterns for any MOF-compliant metamodels. With this language, a pattern is specified as a MOF metaclass containing references to classes and links of a target metamodel through roles and connectors. Gipsie shares with EPattern some meta-level architectural principles and the ability to specify patterns for any modeling language. However, there are also some differences due to their distinct usage: some constructs offered by Gipsie to cope with identity, sharing, abstraction, distinctness and variability in pattern specification do not exist in EPattern. Our approach also provides a richer and finer-grain structuring with property and value roles. Last, there is no mention of model fragment in EPattern and features to validate the pattern specification against the target metamodel are lacking.

Some works not based upon roles have also studied the integration of pattern solutions in target models. [12] presents a metamodel-based approach providing a representation of both the design problem solved by pattern and its solution. Each part is specified by a specialized metamodel like in [6]. Application of a pattern is implemented as a transformation that takes an input model conforming to the problem metamodel and produces an output model containing the solution. The work of [19] proposes a metaprogramming approach which applies a design pattern to a UML model by successive transformation steps leading to a final situation where the occurrence of the pattern is explicit. Aspect-oriented modeling (AOM) approaches[3,15,11,22] can also be seen as a technical solution to incorporate models representing pattern solutions into various base models (see examples in [3,11]). AOM approaches provide a notion of model-based aspect made of pointcuts and advices as well as a model weaving process that combines advice element with elements from the base model each time pointcuts match. In all these works, applying design patterns requires an existing initial situation whereas ours does not need a particular structure to apply a pattern. Another difference with our work is that these approaches are generally limited to a specific language which is often a part of UML[14]. Last, our approach provides several pattern-specific features to address abstraction, identity, sharing in pattern specification and integration that have no equivalent in the cited approaches.

A last category of works related to the present one concerns those that provide a notion of model component to construct models by reuse. In this category, we can cite works based on UML template [20,14] which allows representing generic models as packages parameterized by model elements, then produce other models through parameters substitution. UML template can be used to express and apply pattern solutions [14]. From this point of view, limitations of UML Template compared with our approach are to be metamodel-specific, to expect a matching structure limiting the application and to require strict conformance with the metamodel resulting in unneeded constraints. Another recent work providing a notion of model component for any domain-specific modeling language is Reuseware [9]. In Reuseware, components are model fragments with ports that can be composed by defining composition programs where ports are linked to combine elements from multiple components. The structure of components and component programs for a DSL are defined by a composition system that extends the DSL with new constructs for reuse. The need to provide new definitions for a specific language is a distinct feature with our approach which does not have

[14] In AOM, only the work of [13] can be used with arbitrary modeling language.

this requirement. Finally, this approach differs from ours by requiring to clearly identify elements of models fragment that can be composed whereas we only require such identification at binding time. In our case, this offers more flexibility to apply a pattern solution in many contexts.

7 Conclusion

In this paper, we have presented a generic role-based approach and its tools to describe accurately design patterns solutions and support their reuse through pattern realizations. In future works, we plan to extend the language to handle constraints between roles like the distinctness or equality of their respective target objects. We also plan to exploit binding models to check if a target model still complies with the pattern intents after a modification. We will explore the representation of roles played multiple times in a same pattern realization. Evolution of the binding model for deeper modification of the target model and more flexibility in the application process will also be studied. More generally, this work is a first step toward pattern engineering available for any metamodel.

References

1. Albin-Amiot, H., Guéhéneuc, Y.: Meta-Modeling Design Patterns: Application to Pattern Detection and Code Synthesis. In: Proceedings of ECOOP Workshop on Automating Object-Oriented Software Development Methods (2001)
2. Byun, Y., Sanders, B.A.: A Pattern-based Development Methodology for Communication Protocols. Journal of Information Science and Engineering 22 (2006)
3. Clarke, S., Walker, R.J.: Composition patterns: An approach to designing reusable aspects. In: Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001 (2001)
4. Mapelsen, D., Hosking, J., Grundy, J.: Design Pattern Modelling and Instantiation using DPML. In: Proceedings of 40th TOOLS, ACS (2002)
5. Elaasar, M., Briand, L.C., Labiche, Y.: A Metamodeling Approach to Pattern Specification. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 484–498. Springer, Heidelberg (2006)
6. France, R.B., Kim, D.-K., Ghosh, S., Song, E.: A UML-Based Pattern Specification Technique. IEEE Transactions on Software Engineering 30(3) (2004)
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J., Booch, G.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Westley (1995)
8. Geppert, B., Rossler, F.: The SDL pattern approach — a reuse-driven SDL design methodology. Computer Networks 35(6), 627–645 (2001)
9. Johannes, J., Fernández, M.A.: Adding Abstraction and Reuse to a Network Modelling Tool Using the Reuseware Composition Framework. In: Kühne, T., Selic, B., Gervais, M.-P., Terrier, F. (eds.) ECMFA 2010. LNCS, vol. 6138, pp. 132–143. Springer, Heidelberg (2010)
10. Kelly, S., Tolvanen, J.: Domain-Specific Modeling. Wiley & Sons (2008)
11. Lahire, P., Morin, B., Vanwormhoudt, G., Gaignard, A., Barais, O., Jézéquel, J.-M.: Introducing Variability into Aspect-Oriented Modeling Approaches. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MoDELS 2007. LNCS, vol. 4735, pp. 498–513. Springer, Heidelberg (2007)

12. Mili, H., El-Boussaidi, G.: Representing and Applying Design Patterns: What Is the Problem? In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 186–200. Springer, Heidelberg (2005)
13. Morin, B., Klein, J., Barais, O., Jézéquel, J.-M.: A Generic Weaver for Supporting Product Lines. In: International Workshop on Early Aspects at ICSE 2008 (2008)
14. Muller, A., Caron, O., Carré, B., Vanwormhoudt, G.: On Some Properties of Parameterized Model Application. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 130–144. Springer, Heidelberg (2005)
15. 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., Aksit, M. (eds.) Transactions on Aspect-Oriented Software Development I. LNCS, vol. 3880, pp. 75–105. Springer, Heidelberg (2006)
16. Reenskaug, T.: Working with Object, The OORAM Software Engineering Approach. Manning (1996)
17. Riehle, D.: Describing and composing patterns using role diagrams. In: Proceeding of WOON 1996 (1st Int'l. Conference on Object-Orientation in Russia) (1996)
18. Sanada, Y., Adams, R.: Representing Design Pattern in UML: Towards a Comprehensive Approach. Journal of Object Technology 1(2) (2002)
19. Sunyé, G., Le Guennec, A., Jézéquel, J.-M.: Design Patterns Application in UML. In: Bertino, E. (ed.) ECOOP 2000. LNCS, vol. 1850, pp. 44–62. Springer, Heidelberg (2000)
20. Auxiliary Constructs Templates. UML 2.0 Superstructure Specification (2003)
21. van der Aalst, W.M.P., ter Hofstede, A.H.M., et al.: Workflow Patterns. In: Distributed and Parallel Databases, vol. 14. Kluwer Academic Publishers (2003)
22. Whittle, J., Jayaraman, P., Elkhodary, A., Moreira, A., Araújo, J.: MATA: A Unified Approach for Composing UML Aspect Models Based on Graph Transformation. In: Katz, S., Ossher, H., France, R., Jézéquel, J.-M. (eds.) Transactions on AOSD VI. LNCS, vol. 5560, pp. 191–237. Springer, Heidelberg (2009)

An Architecture for Information Exchange Based on Reference Models

Heiko Paulheim[1], Daniel Oberle[2], Roland Plendl[2], and Florian Probst[2]

[1] Technische Universität Darmstadt
Knowledge Engineering Group
paulheim@ke.tu-darmstadt.de
[2] SAP Research
{d.oberle,roland.plendl,f.probst}@sap.com

Abstract. The goal of reference models is to establish a common vocabulary and recently also to facilitate semantically unambiguous information exchange between IT systems. However, IT systems are based on implementation models that typically deviate significantly from the reference models. This raises the need for a mapping mechanism, which is flexible enough to cope with the disparities between implementation model and reference model at runtime and on instance level, and which can be implemented without altering the established IT system. We present an architecture that solves this problem by establishing methods for representing the instances of an existing IT-System in terms of a reference model. Based on rules, the concrete nature of the representation is decided at run time. Albeit our approach is entirely domain independent, we demonstrate the feasibility of our approach in an industrial case study from the Oil and Gas domain, using the ISO 15926 ontology as a reference model and mapping it to different Java and Flex implementation models.

1 Introduction

Semantic modeling techniques have evolved in the areas of knowledge representation [1], object orientation [2,3], and recently ontologies [4]. However, neither software engineers nor knowledge engineers had considered enterprise applications for using semantic modeling techniques several decades ago. In the meantime, the complexity of enterprises as well as their need to exchange information ad-hoc continues to grow leading to an increased awareness of the need for semantically richer information. Since the early 1990s, the idea of modeling specific aspects of an enterprise has led to the definition of various *reference models* where semantic modeling techniques are used to formalize the concepts identified by the different standards bodies [5]. The initial purpose of such standards was to create a commonly accepted nomenclature for representing structural and operational aspects of enterprises. Once these standards had been accepted by a large enough community, industry solutions exploiting them have been implemented closing the loop to allow enterprises using standards-based IT systems. [6] This situation facilitates building new composite applications based on the reference model.

In order to illustrate this, we refer to the example of the Oil and Gas industry whose declared goal is to enable information integration to support, e.g., the integration of different data sources and sensors, the validation of information delivered by different sources, and the exchange of information within and between companies via web services [7]. This requires that Oil and Gas IT systems, e.g., asset management or facility monitoring, share information according to a reference model, and that they can interpret messages using that model. In the particular case of the Oil and Gas industry, the reference model is given by the ISO 15926 ontology which is formalized in the W3C Web Ontology Language (OWL) [8]. First middleware solutions are available to address the task of information integration, e.g., IBM's Information Integration Framework [9]. Such middleware solutions allow information exchange between the existing IT systems based on ISO 15926. Further, they facilitate the creation of new composite applications, such as production optimization or equipment fault protection. Such composite applications depend on information stemming from several existing IT systems, and, thus, benefit from semantically unambiguous information exchange.

Since the reference model is typically designed *ex post*, i.e., long after IT systems have been put in place, the IT systems' *implementation models* have to be mapped to such a reference model. Different IT systems feature implementation models specified in different implementation languages. Examples for such implementation languages are object-oriented languages or relational schemas. Therefore, the outlined setting requires to cope with *arbitrary* implementation languages. In addition, mapping between reference and implementation models is typically a non-trivial task which requires a *flexible* mechanism to cope with all kinds of disparities between the two kinds of models. The reason is that implementation models serve the purpose of providing a model which allows for simple programming resulting in efficiently executable code. In contrast, reference models serve the purpose of providing a clear, formal conceptualization of a domain. Further, the mapping has to be *bidirectional* since the IT system has to send and receive messages expressed according to the conceptualization underlying the reference model. In addition, for coping efficiently with the disparities between reference and implementation model, the mapping process itself must happen at *runtime* and on the *instance level.* For example, an asset management application might contain instance data about a specific pump. This instance data has to be represented by means of ISO 15926 for information exchange with other IT systems via a middleware solution. Finally, the established IT systems cannot be touched in most cases. That means, the mapping mechanism has to be implemented in a *non-intrusive* way.

Despite the existence of sophisticated solutions for model mapping [10], database integration [11], or ontology mapping [12], the existing approaches for mapping models of both kinds are still limited with respect to supporting the outlined settings. Therefore, we contribute a flexible, bidirectional, and non-intrusive approach for performing the mapping process between reference and implementation models at run-time and on the instance-level. The approach can be used with a multitude of arbitrary implementation languages and especially when the reference model is given ex post. Although we explain our approach along an example of the Oil and Gas domain, our approach is generic, meaning that it prescribes an architecture that can be instantiated differently depending on the language of the reference model, implementation model, IT system landscape or the application domain.

We start by introducing typical deviations between reference and implementation models in Section 2, using the ISO 15926 ontology as a running example. Section 3 surveys related approaches along our distinguishing features of being flexible, bidirectional, non-intrusive, runtime, instance level, and ex post. Section 4 introduces our reference architecture, which is instantiated in a case study from the oil and gas domain in Section 5. A scalability and performance evaluation can be found in Section 6. Finally, we give a conclusion in Section 7.

2 Typical Deviations

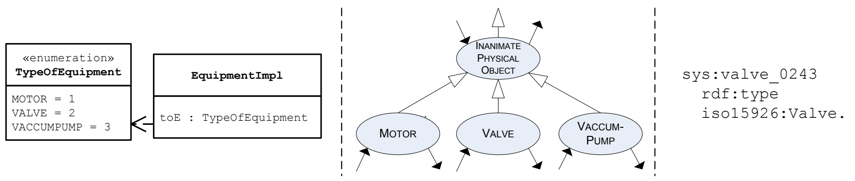
Reference models and implementation models are different by nature. The reason is that implementation models are task-specific, with the focus on an efficient implementation of an application. In contrast to reference models, modeling decisions are geared towards a pragmatic and efficient model. Due to those differences, one often faces the situation where implementation models and reference models are incompatible in the sense that a 1:1 mapping between them does not exist. This also holds when both kinds of models are specified in the same language.

To show some typical deviations between the two kinds of models, we use examples from the Oil and Gas domain. As discussed in the introduction, the ISO 15926 ontology serves as a reference model. Facilitating information exchange between IT systems using ISO 15926 requires serializing data (e.g., Java objects) from IT systems to RDF data and deserializing that data back to data in the receiving IT system. The W3C Resource Description Framework (RDF) [13] is a semi-structured, graph-based language that applies triples to represent statements about Web URIs. For example, the triple

```
sys:valve_0243 rdf:type iso15926:Valve.
```

states that `sys:valve_0243` is an instance of the ontology category `iso15926:Valve`. The subject (`sys:valve_0243`), predicate (`rdf:type`), and object (`iso-15926:Valve`) of the statement are all specified as URIs, using abbreviated namespaces [14].

Fig. 1 shows a typical mismatch between reference and implementation models, viz., a *multi-purpose class*. A class `EquipmentImpl` could be used to model different types of equipment, distinguished by the `toE` (type of equipment) flag. In the ISO 15926 ontology, several thousand types of equipment are defined as separate ontology categories. Representing each of the types as a separate class would lead to an ineffective class model, so using a single class with a flag is a more pragmatic solution.



```
sys:valve_0243
  rdf:type
  iso15926:Valve.
```

Fig. 1. Typical mismatch 1: Multi-purpose classes. The left hand side shows a class model, the middle depicts an excerpt of the ISO 15926 ontology, and the right hand side shows a sample desired RDF serialization

With a 1:1 mapping, however, an `EquipmentImpl` object cannot be serialized without information loss. A 1:1 mapping can only map `EquipmentImpl` to the ontology category INANIMATE PHYSICAL OBJECT, which serves as a common super category for all equipment categories. With that mapping, an `EquipmentImpl` object `valve_0243` would be serialized as

```
sys:_0243 rdf:type iso15926:InanimatePhysicalObject.
```

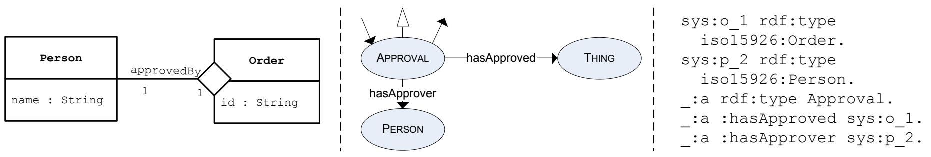
This serialization implies the loss of information stored in the `toE` attribute, as it does not support a deserialization to an `EquipmentImpl` with a proper value for the `toE` attribute. Therefore, an approach relying on a 1:1 mapping is not very useful here.

Other similar deviations encompass *conditional classes*, e.g., classes which depict objects that may or may not exist (usually determined via a `deleted` flag), and *artificial classes*, which depict objects of different kinds without a meaningful common super category (such as a class `AdditionalCustomerData` storing both a social security number and an email address). Furthermore, there are also *multi-purpose relations*, which may depict different relations in the ontology, depending on a flag or on the nature of the related object.

Another typical deviation between implementation and reference models are *shortcuts*. Shortcuts may span across different relations between objects, leaving out intermediate entities. Fig. 2 shows such a deviation: the ISO 15926 ontology defines a category APPROVAL, which has a relation both to the approved THING, as well as the approving authority. The APPROVAL itself has more detailed attributes, such as a DATE.

A corresponding implementation model defines an `Order` and a `Person` class (both of which are categories also present in the ISO 15926 ontology), but omits the intermediate APPROVAL category in favor of a direct relation, implemented as an attribute in the `Order` class. To properly serialize an `Order` object, an APPROVAL instance has to be created as well in the serialization, as depicted in Fig. 2.[1] During deserialization, this instance is then used to create the link

[1] The underscore namespace prefix is a standard RDF N3 notation which denotes an *anonymous resource*, i.e., an object that is known to exist, but whose identity is unknown [14].



```
sys:o_1 rdf:type
  iso15926:Order.
sys:p_2 rdf:type
  iso15926:Person.
_:a rdf:type Approval.
_:a :hasApproved sys:o_1.
_:a :hasApprover sys:p_2.
```

Fig. 2. Typical mismatch 2: Shortcuts

between the `Order` and the `Person` object. Such a serialization encompassing multiple categories in the ontology cannot be implemented using a 1:1 mapping.

Deviations may also occur on the attribute level. A typical example are *compound data types* such as dates, which are most often represented as one variable in a class model. In the ISO 15926 ontology, dates are represented using single individuals for the day, month, and year part of a date, respectively. Another deviation are *counting attributes*, such as an integer attribute `numberOfParts`, which has to be serialized as a set of (anonymous) individuals, and deserialized back to an integer number.

3 Related Work

The deviations introduced in the previous section require a careful mapping between implementation model and reference model if the goal is to exchange information based on the reference model. A semantically correct mapping is the prerequisite for unambiguous, lossless information exchange, and, thus, for building new composite applications relying on information from several IT systems. Related approaches have been proposed in many fields, such as information integration or ontology mapping, as depicted in Table 1. In this section, we provide a survey of such related approaches and conclude that none of them supports all required features.

As explained in the introduction, reference models are typically designed after IT systems' implementation models are established. Therefore, an adequate approach must support an *ex post* mapping. Moreover, simple 1:1 mappings between both types of models are not sufficient. Instead, *flexible* mappings are required to cope with the typical deviations addressed in the previous section. Since IT systems or new composite apps have to send and receive messages in terms of the reference model, the mapping must be *bidirectional*. Further, the mapping must happen at *runtime* of the IT system and on the *instance level*, i.e., concrete instance data has to be mapped that adheres to the implementation and reference model, respectively. Most IT systems are established and cannot be altered. Therefore, the mapping mechanism must be *non-intrusive*. If a multitude of IT systems is involved, it is likely that they use different implementation languages to specify their implementation models. Correspondingly, *arbitrary* implementation languages have to be supported.

The first category of related approaches shown in Table 1 is the field of *database design* which distinguishes between conceptual models and logical

Table 1. Categorization of approaches according to different criteria. A "yes" means that there are approaches in the category that fulfill the criterion, not that *each* approach in the category fulfills the criterion.

Approach \ Criterion	ex post	flexible	bidirectional	runtime	instance level	non-intrusive	arbitrary
Database Design	no	yes	no	no	no	no	no
Model-Driven Engineering	no	yes	yes	no	no	no	yes
Information Integration	yes	yes	yes	yes	yes	yes	no
Direct Semantic Programming Models	yes	no	yes	yes	yes	yes	no
Indirect Semantic Programming Models	no	yes	yes	yes	yes	no	no
API Generation from Ontologies	no	yes	yes	no	no	no	yes
Ontology Mapping	yes	yes	yes	no	yes	yes	no

models according to [15]. Both bear resemblance to our notion of reference and implementation models. CASE tools support the database designer to create a conceptual model, e.g., an ERM [16], and automatically transform it to a logical model, e.g., a relational schema. More recent tools, such as *Together* by Borland,[2] also support reverse engineering models from existing databases. However, ex post mappings to other schemata are typically not supported. Also, the approaches do not work on the instance level and usually not at runtime.

In the area of *model-driven engineering*, platform independent models (PIMs) are transformed to platform specific models (PSMs), which generally correspond to our notion of reference and implementation models. However, this transformation does not happen at runtime and is also not intended to work on the instance level. Originally conceived as unidirectional (transformation from PIM to PSM), recent approaches also allow bidirectional mappings by implementing reverse engineering [17,18].

Information integration deals with accessing information contained in different IT systems using one central, mediated schema. The integrated systems are addressed using *wrappers*, which provide an interface to the information contained in the system, typically to their database. Queries posed using the mediated schema are translated to sub queries posed to the individual wrappers, and the results are collected and unified by an information integration engine [19]. While there are a number of very powerful information integration systems, e.g., for using an ontology as a central, mediated schema [20], they are most often limited to integrating sources of one technology, i.e., SQL-based databases.

With a growing popularity of ontologies, different *semantic programming models* have been proposed for building ontology-based software. Those programming models can also be used for mapping instances in a program to a reference

[2] http://www.borland.com/us/products/together/

ontology. There are two types of semantic programming models: *direct* and *indirect* models [21].

Direct semantic programming models let the user work with any object-oriented programming language (e.g., Java) and allow for mapping classes of that language to categories in the ontology. With such a mechanism, it is possible to serialize programming language objects as RDF data according to a reference ontology, and vice versa. While most direct programming models are intrusive (see [22] for a survey), *ELMO*[3] and the work discussed in [23] provide non-intrusive implementations as well and foresee ways of dynamically creating mappings at runtime. Since the possibilities for expressing mappings are limited in all those approaches, it is required that the "the domain model should be rather close to the ontology" [23].

Indirect semantic programming models provide a set of meta-level programming language constructs (such as an `OWLClass` and an `OWLObjectRelation` class in Java), instead of providing mechanisms for mappings on the model level. The most well-known examples are *JENA* [24] and *OWL API* [25]. Since the developer works directly with the ontology-based constructs, the approach allows for flexibly using arbitrary ontologies. The drawback is that an ex post approach is not possible, since the concepts defined in the ontology are used directly in the code, typically as hard coded strings. Furthermore, since the indirect programming model is used directly and deeply in the software, the approaches cannot be regarded as non-intrusive.

Approaches for *generating APIs from ontologies* are a special type of model-driven engineering approaches (see above) which take ontologies, e.g., OWL files, as input for generating class models. Thus, they share the same set of characteristics as MDE approaches. Typical examples for such approaches are *RDFReactor* [26] and *OWL2Java* [27] (see [22] for a survey).

Ontology mapping approaches deal with creating mappings between different ontologies. If instance data are described using an ontology A, they can be interpreted using the terms of an ontology B, if a mapping from A to B exists. The first approaches to ontology mapping relied on tables storing pairwise correspondences between elements in each ontology, and were thus of limited expressivity. Recently, approaches such as *SPARQL++* [28] also allow for flexible mappings, which may also be stored in a non-intrusive way, i.e., external to the mapped ontologies, e.g., by using *C-OWL* [29], or build bridges between RDF and XML following different schemata, such as *XSPARQL* [30]. *Ontology matching* [31] aims at automatically discovering such mappings. However, a runtime mapping to the implementation model level is not foreseen and has to be provided by additional mechanisms, e.g., by employing a semantic programming model (see above).

In summary, there is a large number of approaches which can be exploited for implementing information exchange based on reference models. However, none of those approaches fulfills the complete set of requirements and provides the full flexibility which is needed in many real-world industrial projects.

[3] `http://www.openrdf.org/doc/elmo/1.5/`

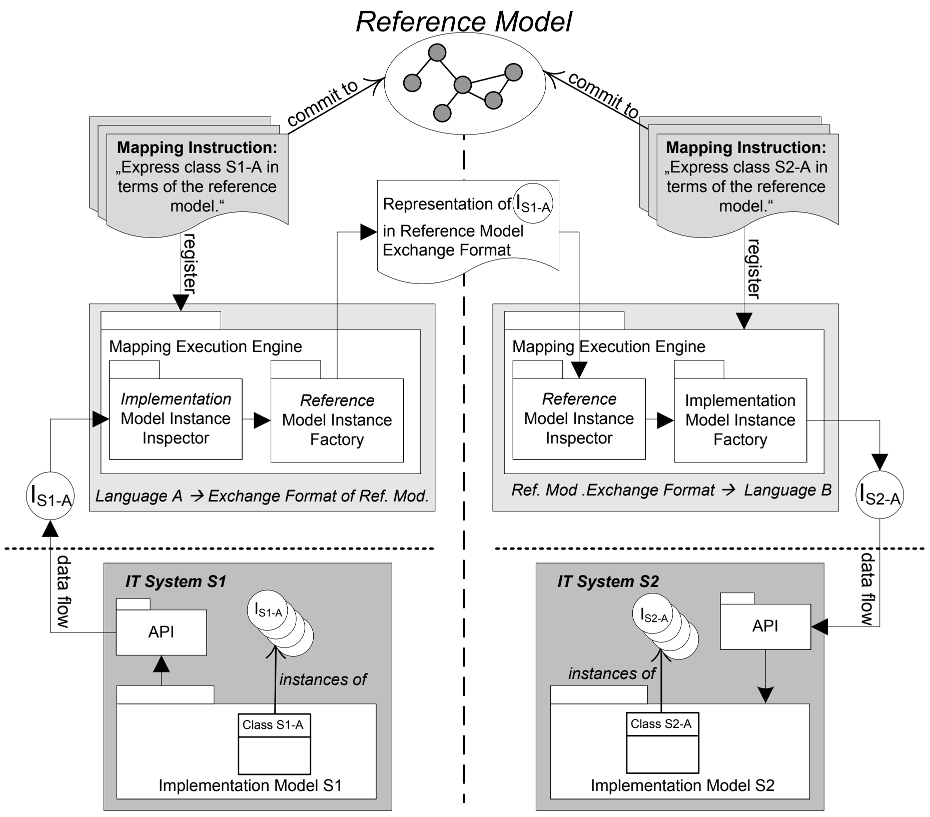

Reference Model
commit to
commit to
Mapping Instruction: „Express class S1-A in terms of the reference model."
Mapping Instruction: „Express class S2-A in terms of the reference model."
Representation of I_{S1-A} in Reference Model Exchange Format
register
register
Mapping Execution Engine
Implementation Model Instance Inspector
Reference Model Instance Factory
Language A → Exchange Format of Ref. Mod.
Mapping Execution Engine
Reference Model Instance Inspector
Implementation Model Instance Factory
Ref. Mod .Exchange Format → Language B
I_{S1-A}
I_{S2-A}
data flow
data flow
IT System S1
API
instances of
Class S1-A
Implementation Model S1
IT System S2
API
instances of
Class S2-A
Implementation Model S2

Fig. 3. General architecture for information exchange between two systems

4 Reference Architecture

Considering the shortcomings of existing mapping approaches, we propose a reference architecture for mapping between implementation and reference models. The architecture meets the criteria of flexible, bidirectional, and runtime mappings which operate on the instance level. The architecture is designed to be implemented in a non-intrusive way, so that it can be employed with legacy systems where the reference model is provided ex post. This also covers the development of new composite applications and mashups from existing applications. The instantiations can operate on arbitrary implementation languages.

Fig. 3 shows the central elements of the architecture which are needed for transferring information from one IT system (S1) into another one (S2) in a semantically consistent way and without changing the implementation models of S1 and S2.

The central means for interoperability is a *reference model*. For each class of the implementation models (or any other relevant element), mapping instructions are written that explain the classes (or other constructs) of the implementation models in terms of the reference model. In this sense, the mapping instructions establish a partial conceptual commitment of the implementation model to the reference model. Formalizing the mapping instructions is performed at design time and by an expert knowing the implementation model as well as the reference model. The mapping instructions are processed by a *mapping execution engine*. The mapping instructions are *executable*, i.e., they can be applied at run-time for providing flexible mappings.

For each IT system, two separate mapping execution engines are established, one for mapping from the implementation model to the reference model, and one for mapping from the reference model to the implementation model. Each of the two engines consists of a *model instance inspector* for the source model, and a *model instance factory* for the target model (Fig. 3 only depicts one mapping execution engine for each IT system, which is needed to illustrate the data flow from S1 to S2).

The information exchange between S1 and S2 is implemented as follows: Instances according to the implementation model of system S1 are in use. If the information carried by one of these instances is to be transferred to system S2, this instance is sent to the mapping execution engine via the API of IT System 1 (in Fig. 3 an particular instance $I_{S1\text{-}A}$ is used to demonstrate the data flow). First, the instance inspector identifies from which class the instance was derived and selects the appropriate mapping instruction. Then, the reference model instance factory creates one or more instances of a class (or different classes) of the reference model in such a way that all information of the instance $I_{S1\text{-}A}$ is represented appropriately, including the relations between the classes. This "instance" is sent to S2.

Note that the reference model can be represented in more than one representation language. Hence, the mapping instructions can be written for more than one representation language. However, the language in which the reference model is written serves as exchange format for the information between S1 and S2. The *mapping execution engine* of S2 receives the previously generated instance and identifies via the *reference model instance inspector* to which class or classes in the reference model the arrived information belongs to. In the final step the *implementation model instance factory* creates one or several instances according to the implementation model of S2. The created instance is handed over to S2 via the API of S2.

It is noteworthy that, since arbitrary mappings and implementation models are possible, the number of objects in S1 and S2 does not have to be the same. An object from S1 (and potentially a set of other objects related to that object) are serialized in a data structure which conforms to the reference model, and deserialized into an object or a set of objects for S2. Since those object models can be conceptually different, the set of objects created for S2 can be substantially different from the original set of objects in S1, and may also be implemented with a different language. Thus, the approach is able to bridge both *conceptual* as well as *technological* heterogeneities.

5 Case Study

The previous section introduced an architecture that can be implemented differently, depending on the language of the reference model, the implementation models, and the IT system landscape. For example, we have discussed an instantiation in the area of emergency management in [22]. In the following, we introduce an instantiation in an industrial case study in the outlined Oil and Gas domain and show how the generic elements of the architecture are implemented in that concrete scenario. The role of the reference model is played by the ISO 15926 Oil and Gas ontology [8] specified in OWL/RDF. For the sake of brevity, Figure 4 shows the instantiated architecture with *facility monitoring* as an existing IT system and *production optimization* as a new composite application. Further existing IT systems are in place (not shown in Figure 4), e.g., rotating equipment monitoring, engineering systems, or asset management, which exchange information with production optimization. With respect to the implementation languages, we assume that the facility monitoring is Java-based, and the production optimization is Flex-based, with class models (implemented in Java and Flex, respectively) as implementation models. The latter provides an interface for exchanging objects with JSON [32] to facilitate data exchange.

5.1 Mapping Specification

To facilitate an executable mapping, we use *rules* for expressing mappings between class models and the ISO 15926 ontology (and vice versa). These rules can be evaluated at runtime on Java and JSON objects to create the desired RDF graph describing the object, and on an RDF graph to create the corresponding set of objects. Fig. 5 shows how the mapping rules are used to transform a Java object into an RDF graph and back, using the example depicted in Fig. 1. We have employed a set of simple rule-based languages, which re-use elements from common querying languages, such as XPath [33] and SPARQL [34]. The next sections explain the different rule syntaxes in detail.

Mapping Class Models to ISO 15926. Our mapping approach uses tests on the objects to be mapped as rule bodies, and a set of RDF triples to be produced as rule heads. For each Java and Flex class, one rule set is defined. In cases where objects have relations to other objects, the rule sets of the corresponding related classes are executed when processing a related object. Rule sets defined for super classes are inherited to sub classes, however, the developer may also override inherited rules explicitly.

For defining dynamic mappings, XPath queries are used. Utilizing such queries, RDF representations for objects can be realized dynamically. Thus, our rules have the following form: the body consists of a test to be performed on an object. The head is a set of RDF triples, each consisting of a subject, a predicate, and an object, all three of which may depend on the object to transform. For defining tests and dependent values, we use XPath expressions. If the test is evaluated positively, one or more triples are generated, consisting of a subject,

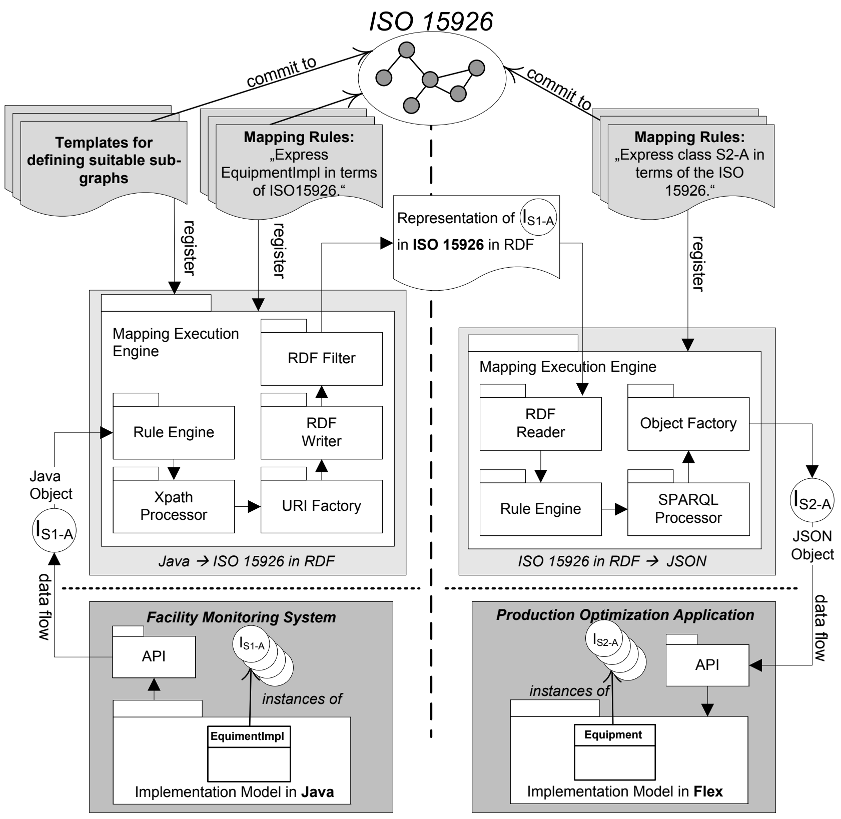


Fig. 4. Instantiated architecture in the Oil and Gas scenario

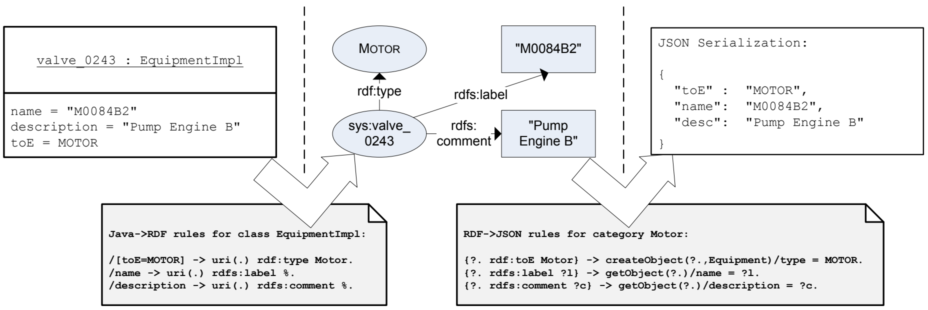
predicate, and object. The subject, predicate, and object may be either constants or XPath expressions as well. Thus, the syntax of our rules looks as follows:[4]

```
   Rule ::= XPathExpr "→" Triple {"," Triple } "." ;      (1)
Triple ::= 3 * (Constant|XPathExpr) ;                     (2)
```

In this syntax, `Constant` denotes an arbitrary sequence of characters enclosed in quotation marks, and `XPathExpr` denotes an XPath expression following the XPath standard [33], enhanced by the following extensions:

– The function `regex()`, called on a Java attribute, evaluates a regular expression [36] on that object and yields `true` if the regular expression matches the attribute value, false otherwise.
– The function `repeat(XPathExpr)`, called with an XPath expression as an argument in the rule body, causes the rule head to be executed as many times as there are results for the given XPath expression.

[4] Represented using the Extended Backus Naur Form (EBNF) [35].



```
valve_0243 : EquipmentImpl

name = "M0084B2"
description = "Pump Engine B"
toE = MOTOR
```

```
JSON Serialization:

{
  "toE" :  "MOTOR",
  "name":  "M0084B2",
  "desc":  "Pump Engine B"
}
```

```
Java->RDF rules for class EquipmentImpl:

/[toE=MOTOR] -> uri(.) rdf:type Motor.
/name -> uri(.) rdfs:label %.
/description -> uri(.) rdfs:comment %.
```

```
RDF->JSON rules for category Motor:

{?. rdf:toE Motor} -> createObject(?.,Equipment)/type = MOTOR.
{?. rdfs:label ?l} -> getObject(?.)/name = ?l.
{?. rdfs:comment ?c} -> getObject(?.)/description = ?c.
```

Fig. 5. Example rules for transferring Java to RDF and RDF to JSON

– The `%` symbol used in the head refers to the result of the XPath test performed in the body.
– The `.` symbol used in the head refers to the currently serialized object.
– The function `uri(XpathExpr)` assigns a unique URI to a Java or Flex object. The argument of the function is again an XPath expression, which may also use the `%` and `.` constructs, as described above.

The result of an XPath test in the body can be used as a variable in the head within the RDF triple to be generated (referred to with the `%` sign), as well as the current object (referred to with the `.` sign). The `uri` function used in a triple generates a unique URI for an object. The triples may also contain blank nodes, which are needed, e.g., to cope with shortcuts.

The XPath expressions may also contain regular expressions to deal with implicit background knowledge and non-atomic data types. Those can be used for conditions or for splitting data values. A `repeat` function can be used to cover object counting deviations (e.g., produce a set of n blank nodes for an attribute value of n). The left-hand side of Fig. 5 shows a set of rules for mapping two Java objects to a subset of the ISO 15926 ontology.

Mapping ISO 15926 to Class Models. The mapping rules from ISO 15926 to class models are similar. Again, rule sets are defined per ontology category, are inherited to sub categories, and rules can be explicitly overridden by the developer.

For rule bodies, SPARQL expressions are used. In the rule heads, objects are created by using `createObject`, and attribute values for created objects are set (by using `getObject` and an XPath expression identifying the attribute to be set). The execution order of rules is defined such that all `createObject` statements are executed first, assuring that all objects are created before attempting to set attribute values.

Typically, Java or Flex objects will be created when a condition is fulfilled, and values are set in these objects. This leads to the following rule syntax for mapping rules from ISO 15926 to class models:

$$\texttt{Rule ::= SPARQLExpr "}\rightarrow\texttt{" ObjFunction } \{ \texttt{ "/" SetObjValue } \} \texttt{ "." ;} \tag{3}$$

$$\texttt{ObjFunction ::= "getObject(" SPARQLVariable}\{\texttt{,SPARQLVariable}\}\texttt{ ","} \\ \texttt{ClassName ")" ;} \tag{4}$$

$$\texttt{SetObjValue ::= XPathExpr "=(ObjFunction|ValueFunction|Constant) ;} \tag{5}$$

$$\texttt{ValueFunction ::= "getValue(" (SPARQLVariable|BuiltinFunction) ","} \\ \texttt{ClassName ")" ;} \tag{6}$$

$$\texttt{BuiltinFunction ::= "count("SPARQLVariable")"} \\ \texttt{| "concat("SPARQLVariable,}\{\texttt{SPARQLVariable}\}\texttt{")" ;} \tag{7}$$

Like in the rules for creating RDF representations from objects, `XPathExpr` denotes an XPath expression. `SPARQLExpr` denotes the WHERE clause of a SPARQL expression, and `SPARQLVariable` denotes a variable defined in that WHERE part and is used for referencing the query's results. `ClassName` is the name of a Java class which is used when creating objects and object values (for handling primitive types, the corresponding wrapper classes are used as a class name). The right-hand side of Figure 5 shows how to use the rules to map RDF representations to Flex objects.

The built-in functions `count` and `concat` are used for counting results and concatenating strings, respectively. Both functions are on the feature list for the next version of SPARQL [37], thus, our proprietary support for those functions may be removed from the rule language once that new version becomes a standard with adequate tool and API support.

For determining which categories an RDF instance belongs to, and for executing SPARQL statements, reasoning on the RDF graph and the domain ontology can be used. To cover non-atomic data types, the rules may use a `concat` function for concatenating different results of a SPARQL query. For coping with object counting, a `count` function can be used to produce the corresponding attribute values (once SPARQL version 1.1, which supports counting, becomes a standard, this will be obsolete).

Although the rules for both directions look similar, there is one subtle difference. The XPath expressions used on the object model are executed with closed world semantics, while the SPARQL expressions used on the RDF model are executed with open world semantics[5]. The rationale is that the set of objects in an information system is completely known (and therefore forms a closed world), whereas an RDF graph representing a set of objects will typically only represent a subset of the original information.

5.2 Template-Based Filtering for Data Exchange

The rules discussed above are typically evaluated in a recursive manner. This may lead to problems when creating the data structure for an implementation model instance. When creating the RDF representation for an object, each object that is encountered underway is queued and processed. For very large connected object graphs, this means that the resulting RDF graph can grow fairly large. Especially when using that graph for data exchange between applications, such a large graph can be undesirable for reasons of performance.

A straight forward way would be defining different rule sets for each class, depending on which kind of object is currently serialized. Such a solution would lead to n^2 rule sets for n classes and thus be rather costly. If we would want to take arbitrary paths into account (e.g., include the address of a person's employer into the annotation, but not the addresses of that person's friends' employers), the complexity would even be exponential.

A better alternative is to use *templates* which define the sub graph that is to be generated for an object of a certain class. While rules define *the whole* possible graph that can be produced for an object and are thus universal, templates specifically restrict that graph to a sub-graph. This alternative reduces the complexity to n rule sets and n templates. In our approach, the templates can be written in plain RDF, which allows for a straight forward definition and re-use of existing tools. Furthermore, since the rules are universal, they may be reused for different information transmission use cases by only applying a different set of templates.

5.3 Non-intrusive Implementation

For our case study, we have implemented the solution sketched above in a prototype capable of exchanging objects between Java and Flex applications. In our integrated prototype, the Flex applications run encapsulated in Java containers, and their API provides and consumes Flex objects in JSON notation [38].

Fig. 4 shows the architecture as it was implemented for the case study. The left hand side depicts an equipment fault protection application, implemented in Java. The mapping execution engine shown in the figure produces RDF graphs from Java objects which are obtained from the application through its API. The rule engine processes the mapping rules discussed above and uses an object inspector implemented with *JXPath*[6] for performing tests on the Java objects.

[5] The `count` function discussed above counts *results* in the result set of a SPARQL query (which forms a closed world), not in the underlying graph.

[6] `http://commons.apache.org/jxpath/`

The rule engines processing our rule languages have been implemented using parsers generated from abstract grammars using *JavaCC*.[7]

By using Java's reflection API [39] and relying on the Java Beans specification [40] (a naming convention for object constructors and for methods for accessing property values), the implementation is non-intrusive and does not require changes to the underlying class model in Java. A *URI factory* keeps track of the RDF nodes created for each object and assigns unique URIs. An RDF writer, implemented with *JENA* [24], creates RDF files which conform to the ISO 15926 ontology. These files are the cetral elements for semantically correct informatione exchange in the proposed architecture.

As discussed above, it is desirable to reduce the set of transmitted RDF data as far as possible. Thus, we have implemented a filter based on templates expressed in RDF. Thus, the mapping rules from the Java model to the ontology have two parts: the rules themselves, generating the whole possible graph for an object, and the template reducing that graph to the desired subset.

On the right-hand side, a production optimization application is shown, which is implemented in Flex, and which is supposed to consume data from the Java-based equipment fault protection application. To that end, it receives RDF data based on the ISO 15926 ontology. This data is processed using the mapping rules described above, which are executed in a rule engine. The RDF data is analyzed using JENA as a SPARQL engine, and corresponding JSON data is produced and enriched using a Java-based reimplementation of *JSONPath*[8] as an object factory. The JSON objects created are then handed to the Flex application's API. For Flex-based applications, the transformation between RDF and JSON is done entirely in the Java container, and the Flex application is only addressed by using its JSON-based API, the implementation is also non-intrusive with respect to Flex-based applications.

6 Scalability and Performance Evaluation

As the examples in Section 2 show, information exchange between IT systems require a flexible approach for transforming information from an IT system into a representation that follows a reference model and back. When using Java and Flex-based applications in the Oil and Gas domain, this means mediating between Java and Flex-based class models and the ISO 15926 reference ontology.

The instantiation of the architecture shown in Section 5 is capable of handling all the typical deviations introduced. To build useful solutions, especially real-time systems, this implementation has to be able to handle larger amounts of data in short times. Therefore, we have run several performance tests on our approach.

For these performance tests, we have used artificially created objects graphs consisting of up to 10,000 interconnected objects and transformed them to RDF and back to Java and Flex with our mapping engine. Fig. 6 shows the processing

[7] `https://javacc.dev.java.net/`

[8] `http://goessner.net/articles/JsonPath/`

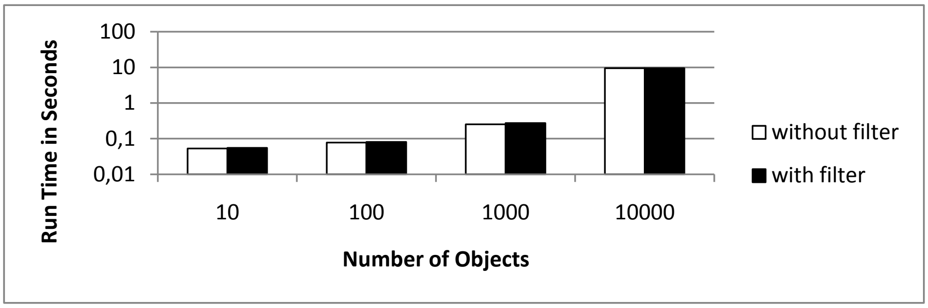


Fig. 6. Runtime behavior for serializing Java objects as RDF

time for serializing Java objects in RDF, once with and once without applying the template-based filtering mechanism. It shows that the time required per object is below one millisecond, and that the processing time scales linearly with a growing number of objects. The figures for transformation from Flex objects look very similar.

For deserializing Java objects from RDF graphs, different reasoning mechanisms can be used for evaluating the SPARQL queries. For the evaluation, we have used three different built-in reasoners in the JENA framework: a simple transitive reasoner only working on subclass and subproperty relations, an RDF(S) reasoner, and an OWL reasoner. Except for the latter, the processing time for each object is below ten milliseconds. In either case, the approach scales linearly with a growing number of objects. Again, the figures for transforming to Flex-based models look very similar.

The trade-off for using more powerful reasoning is a more complex definition of mapping rules for the developer, since certain information which may be required for the mapping (e.g., class membership of RDF instances) can be either inferred automatically by the reasoner, or encoded explicitly in a mapping rule.

Figures 6 and 7 also demonstrate the impact of the template-based filtering mechanism on performance: while applying the template during the serializing step does not lead to a significant performance impact, smaller RDF structures may be deserialized much faster than larger ones. Thus, it is beneficial to reduce the RDF structures before the transfer to another system as far as possible. For example, if the RDF structure to be transferred can be reduced by 50%, the total processing time is also decreased by 50%, as the time for creating Java objects from RDF decreases linearly, while there is no overhead in applying the filter.

In summary, the evaluations show that the dynamic, rule-based mapping algorithm can be implemented in a fast and high-performance manner, which does not add any severe run-time overhead to the information exchange between IT systems, and which also scales up to larger object graphs.

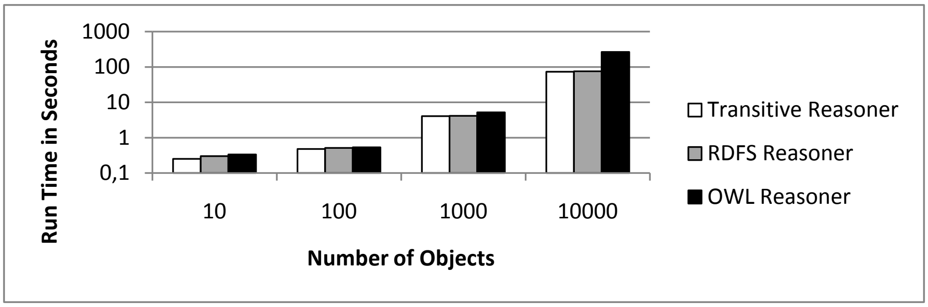


Fig. 7. Runtime behavior for deserializing RDF graphs as Java objects

7 Conclusion and Future Work

We have introduced a flexible, bidirectional, and non-intrusive approach for mapping between reference and implementation models on the instance level. The mappings are used at runtime while the mapping instructions can be specified after the implementation of the applications and the reference model. The work is motivated by conceptual deviations between implementation models and reference models that can be frequently observed in existing software systems. We discussed these domain independent deviations in detail in order to draw attention to an often ignored problem that occurs when reference models are used in software engineering. The central conclusion of this discussion is that 1:1 mappings between implementation models and reference models will not lead to the expected effect of syntactically and semantically correct information exchange between independent applications.

With a case study from the oil and gas domain, we have discussed and shown in an implementation how our approach can be used to bridge both conceptual and technological heterogeneities between applications, and to facilitate semantically correct information exchange between Java and Flex-based applications, using the ISO 15926 ontology. The central advantage of the presented approach is that not classes of the implementation models are mapped 1:1 to the reference model but only its instances. *Explaining* the instances of the implementation model in terms to one or more classes of the reference models allows for a great flexibility for the implementation model. Legacy systems can be annotated without making compromises in terms of ontological or conceptual soundness, and due to the non-intrusive approach, can also be used with modern enterprise buses without having to be modified to comply to a newly created reference model. Furthermore, newly developed implementation models can be specified having only computational efficiency and elegance in mind. Compliance to a domain vocabulary or standards can be established via the presented approach. This opens the possibility for establishing composite applications by reusing already existing systems and information sources.

The paper has focused on the use case of information integration and exchange. However, the mechanism of mapping different implementation models to one common reference model may also be used to access the information in different applications available as a unified linked data set, allowing for reasoning and for unified visualization [41], and for other purposes that require run-time access to a system's data in a form that can be processed by a reasoner, such as self-explaining systems, self-adapting user interfaces, or semantic event processing in integrated applications [42].

Currently, the developer has to specify the mapping rules by hand. A straight forward improvement is the provision of a tool set for assisting the developer in creating the mapping rules. In the future, techniques developed, e.g., in the field of ontology matching [31] and schema matching [43] may be employed to suggest mappings and rules to the developer in an interactive manner. However, this poses several challenges, since our rules are more complex (combining the full expressive power of XPath, SPARQL, and regular expressions) than those that can be discovered with state of the art tools. A possible solution would be to suggest an approximation of a mapping rule to the user, and let her refine that approximation to a complete mapping rule.

Acknowledgements. The work presented in this paper has been partly funded by the German Federal Ministry of Education and Research under grant no. 01ISO7009 (SoKNOS), 01IA08006 (ADiWa), and 13N10711 (InfoStrom).

References

1. Brachman, R.J., Schmolze, J.G.: An Overview of the KL-ONE Knowledge Representation System. Cognitive Science 9(2), 171–216 (1985)
2. Booch, G., Rumbaugh, J.E., Jacobson, I.: The Unified Modeling Language User Guide. J. Database Manag. 10(4), 51–52 (1999)
3. Stipp, L., Booch, G.: Introduction to object-oriented design (abstract). OOPS Messenger 4(2), 222 (1993)
4. Staab, S., Studer, R. (eds.): Handbook on Ontologies. International Handbooks on Information Systems. Springer (2009)
5. Rebstock, M., Fengel, J., Paulheim, H.: Ontologies-based Business Integration. Springer (2008)
6. Pletat, U., Narayan, V.: Towards an upper ontology for representing oil & gas enterprises. In: Position paper for W3C Workshop on Semantic Web in Energy Industries; Part I: Oil and Gas (2008)
7. Verhelst, F., Myren, F., Rylandsholm, P., Svensson, I., Waaler, A., Skramstad, T., Ornæs, J., Tvedt, B., Høydal, J.: Digital Platform for the Next Generation IO: A Prerequisite for the High North. In: SPE Intelligent Energy Conference and Exhibition (2010)
8. Kluewer, J.W., Skjæveland, M.G., Valen-Sendstad, M.: ISO 15926 templates and the Semantic Web. In: Position paper for W3C Workshop on Semantic Web in Energy Industries; Part I: Oil and Gas (2008)
9. Credle, R., Akibola, V., Karna, V., Panneerselvam, D., Pillai, R., Prasad, S.: Discovering the Business Value Patterns of Chemical and Petroleum Integrated Information Framework. Red Book SG24-7735-00, IBM (August 2009)
10. Bernstein, P.A., Melnik, S.: Model Management 2.0: Manipulating Richer Mappings. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, pp. 1–12 (2007)

11. Doan, A., Halevy, A.Y.: Semantic Integration Research in the Database Community: A Brief Survey. AI Magazine 26(1), 83–94 (2005)
12. Choi, N., Song, I.-Y., Han, H.: A survey on ontology mapping. SIGMOD Record 35(3), 34–41 (2006)
13. Manola, F., Miller, E.: RDF Primer. W3C Recommendation (February 2004), http://www.w3.org/TR/rdf-primer/
14. Berners-Lee, T.: Notation3 (N3) A readable RDF syntax (1998), http://www.w3.org/DesignIssues/Notation3
15. ANSI/X3/SPARC Study Group on Data Base Management Systems: Interim Report. FDT – Bulletin of ACM SIGMOD 7(2), 1–140 (1975)
16. Chen, P.P.: The Entity-Relationship Model - Toward a Unified View of Data. ACM Trans. Database Syst. 1(1), 9–36 (1976)
17. Hettel, T., Lawley, M., Raymond, K.: Towards Model Round-Trip Engineering: An Abductive Approach. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 100–115. Springer, Heidelberg (2009)
18. Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 16-18, pp. 4–16. ACM, Portland (2002)
19. Halevy, A.: Information Integration. In: Encyclopedia of Database Systems, pp. 1490–1496. Springer (2009)
20. Sahoo, S.S., Halb, W., Hellmann, S., Idehen, K., Thibodeau Jr., T., Auer, S., Sequeda, J., Ezzat, A.: A Survey of Current Approaches for Mapping of Relational Databases to RDF (2009), http://www.w3.org/2005/Incubator/rdb2rdf/RDB2RDF_SurveyReport.pdf, (accessed July 16, 2010)
21. Puleston, C., Parsia, B., Cunningham, J., Rector, A.: Integrating Object-Oriented and Ontological Representations: A Case Study in Java and OWL. In: Sheth, A.P., Staab, S., Dean, M., Paolucci, M., Maynard, D., Finin, T.W., Thirunarayan, K. (eds.) ISWC 2008. LNCS, vol. 5318, pp. 130–145. Springer, Heidelberg (2008)
22. Paulheim, H., Plendl, R., Probst, F., Oberle, D.: Mapping Pragmatic Class Models to Reference Ontologies. In: DESWeb 2011 - 2nd International Workshop on Data Engineering Meets the Semantic Web. In Conjunction with ICDE 2011, Hannover, Germany, April 11 (2011)
23. Hillairet, G., Bertrand, F., Lafaye, J.Y.: Bridging EMF applications and RDF data sources. In: Kendall, E.F., Pan, J.Z., Sabbouh, M., Stojanovic, L., Bontcheva, K. (eds.) Proceedings of the 4th International Workshop on Semantic Web Enabled Software Engineering, SWESE (2008)
24. Carroll, J.J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K.: Jena: Implementing the Semantic Web Recommendations. In: Feldman, S.I., Uretsky, M., Najork, M., Wills, C.E. (eds.) Proceedings of the 13th International Conference on World Wide Web - Alternate Track Papers & Posters, pp. 74–83. ACM (2004)
25. Bechhofer, S., Volz, R., Lord, P.W.: Cooking the Semantic Web with the OWL API. In: Fensel, D., Sycara, K., Mylopoulos, J. (eds.) ISWC 2003. LNCS, vol. 2870, pp. 659–675. Springer, Heidelberg (2003)
26. Völkel, M., Sure, Y.: RDFReactor - From Ontologies to Programmatic Data Access. In: Posters and Demos at International Semantic Web Conference (ISWC 2005), Galway, Ireland (2005)

27. Kalyanpur, A., Pastor, D.J., Battle, S., Padget, J.A.: Automatic Mapping of OWL Ontologies into Java. In: Maurer, F., Ruhe, G. (eds.) Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering (SEKE 2004), Banff, Alberta, Canada, June 20-24, pp. 98–103 (2004)
28. Polleres, A., Scharffe, F., Schindlauer, R.: SPARQL++ for Mapping Between RDF Vocabularies. In: Meersman, R., Tari, Z. (eds.) OTM 2007, Part I. LNCS, vol. 4803, pp. 878–896. Springer, Heidelberg (2007)
29. Bouquet, P., Giunchiglia, F., van Harmelen, F., Serafini, L., Stuckenschmidt, H.: C-OWL: Contextualizing Ontologies. In: Fensel, D., Sycara, K., Mylopoulos, J. (eds.) ISWC 2003. LNCS, vol. 2870, pp. 164–179. Springer, Heidelberg (2003)
30. Akhtar, W., Kopecký, J., Krennwallner, T., Polleres, A.: XSPARQL: Traveling between the XML and RDF Worlds – and Avoiding the XSLT Pilgrimage. In: Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M. (eds.) ESWC 2008. LNCS, vol. 5021, pp. 432–447. Springer, Heidelberg (2008)
31. Euzenat, J., Shvaiko, P.: Ontology Matching. Springer, Heidelberg (2007)
32. json.org: Introducing JSON (2010), http://www.json.org/
33. W3C: XML Path Language (XPath) 2.0 (2007), http://www.w3.org/TR/xpath20/
34. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C Recommendation (January 2008), http://www.w3.org/TR/rdf-sparql-query/
35. International Organization for Standardization (ISO): ISO/IEC 14977: Information technology – Syntactic metalanguage – Extended BNF (1996), http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.html?csnumber=26153
36. Friedl, J.: Mastering Regular Expressions. O'Reilly (2006)
37. W3C: SPARQL New Features and Rationale (2009), http://www.w3.org/TR/sparql-features/
38. Paulheim, H.: Seamlessly Integrated, but Loosely Coupled - Building UIs from Heterogeneous Components. In: ASE 2010: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 123–126. ACM, New York (2010)
39. Foreman, I.R., Forman, N.: Java Reflection in Action. Action Series. Manning Publications (2004)
40. Sun Microsystems: Java Beans API Specification (1997), http://www.oracle.com/technetwork/java/javase/ /documentation/spec-136004.html
41. Paulheim, H., Meyer, L.: Ontology-based Information Visualization in Integrated UIs. In: Proceedings of the 2011 International Conference on Intelligent User Interfaces (IUI), pp. 451–452. ACM (2011)
42. Paulheim, H., Probst, F.: Ontology-Enhanced User Interfaces: A Survey. International Journal on Semantic Web and Information Systems 6(2), 36–59 (2010)
43. Bonifati, A., Mecca, G., Papotti, P., Velegrakis, Y.: Discovery and Correctness of Schema Mapping Transformations. In: Bellahsene, Z., Bonifati, A., Rahm, E. (eds.) Schema Matching and Mapping, pp. 111–147. Springer (2011)

MoScript: A DSL for Querying and Manipulating Model Repositories

Wolfgang Kling[1,⋆], Frédéric Jouault[1], Dennis Wagelaar[3,⋆⋆], Marco Brambilla[2], and Jordi Cabot[1]

[1] AtlanMod, INRIA École des Mines de Nantes, LINA
{wolfgang.kling,frederic.jouault,jordi.cabot}@inria.fr
[2] Politecnico di Milano, Dipartimento di Elettronica e Informazione
marco.brambilla@polimi.it
[3] Vrije Universiteit Brussel, Software Languages Lab
dennis.wagelaar@vub.ac.be

Abstract. Growing adoption of Model-Driven Engineering has hugely increased the number of modelling artefacts (models, metamodels, transformations, ...) to be managed. Therefore, development teams require appropriate tools to search and manipulate models stored in model repositories, e.g. to find and reuse models or model fragments from previous projects. Unfortunately, current approaches for model management are either ad-hoc (i.e., tied to specific types of repositories and/or models), do not support complex queries (e.g., based on the model structure and its relationship with other modelling artefacts) or do not allow the manipulation of the resulting models (e.g., inspect, transform). This hinders the probability of efficiently reusing existing models or fragments thereof. In this paper we introduce MoScript, a textual domain-specific language for model management. With MoScript, users can write scripts containing queries (based on model content, structure, relationships, and behaviour derived through on-the-fly simulation) to retrieve models from model repositories, manipulate them (e.g., by running transformations on sets of models), and store them back in the repository. MoScript relies on the megamodeling concept to provide a homogeneous model-based interface to heterogeneous repositories.

Keywords: DSL, Megamodel, Model Management, Scripting, OCL.

1 Introduction

As Model-Driven Engineering (MDE) methods and tools are maturing and becoming more popular, the number of modelling artefacts consumed and produced

⋆ The author's work is partially supported by the Galaxy (ANR - French National) project.

⋆⋆ The author's work is funded by a postdoctoral research grant provided by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT-Flanders).

by software engineering processes (e.g., models, metamodels, and transformations) has increased considerably.

MDE for complex systems [4] is a typical example of this situation. In the model driven development of those systems, every artefact (e.g. requirements specifications, analysis and design documents, implementation artefacts, etc.,) is a model. Apart from being numerous, these artefacts are often large, heterogeneous, interrelated, with complex internal structure, and possibly stored in distributed model repositories.

MDE is partly to blame for this complexity, as it introduces new artefacts to deal with, such as models, metamodels, transformation models, and transformations engines. Whereas having special-purpose metamodels allows for reducing model complexity, the interrelations between transformations, models, and metamodels can become very complex. Global Model Management (GMM) aims to address this complexity problem by providing an explicit representation of the modelling artefacts and their interrelations, in a model called *megamodel* [10].

However, current GMM solutions only provide passive metadata. It is possible to query a megamodel, but not to access and manipulate the modelling artefacts represented in a megamodel (e.g. loading/saving models, executing transformations, etc.).

In this paper, we propose MoScript a textual DSL (domain-specific language) and megamodel agnostic platform for accessing and manipulating modelling artefacts represented in a megamodel.

MoScript allows to write queries that retrieve models from a repository, inspect them, invoke services on them (e.g. transformations), and to register newly produced models back to the repository. MoScript scripts allow the description and automation of complex modelling tasks, involving several consecutive manipulations on a set of models. As such, the MoScript language can be used for modelling task and/or workflow automation.

The MoScript architecture includes an extensible metadata engine for resolving and accessing modelling artefacts and invoke services from different transformation tools.

The remainder of this paper is structured as follows. Section 2 explains the motivations of this work. Section 3 describes the supporting architecture for MoScript. Section 4 presents the MoScript language. Section 5 puts everything together in the form of two examples. Section 6 describes how is MoScript implemented. Section 7 compares our work with other, related approaches. Finally, section 8 presents our conclusions and future work.

2 Motivation

Along this section we will present some of the problems that motivated the definition of MoScript. Then, in further sections, will illustrate how MoScript helps us to solve them.

Let's consider repositories of modelling artefacts represented by a megamodel, which are used to develop complex systems. Typically, these repositories would present the following characteristics:

- **Hundreds or thousands of heterogeneous artefacts**. The repositories contain models, metamodels, metametamodels, transformations, source code, file descriptors, data files, etc.
- **Artefacts are related to other artefacts** through predefined (e.g. conformsTo) and ad-hoc (e.g. weaving models) relationships.
- **Many different tools**, each one playing a specific role in the repository, like model to model (M2M) transformation engines, model to text transformation engines (M2T), documentation tools, compilers, script engines etc.
- **Several kind of users participate in the evolution of the repository** (e.g., stakeholders, analysts, developers, etc.).

On this repositories, there are several tasks users may want to accomplish. (1) Finding models, (2) combining modelling artefacts information (3) batch processing, (4) and registering newly generated artefacts in the system.

Finding models may be difficult depending on the search criteria and the size of the repositories. In the simplest case we may be interested in finding a single model whose name (e.g., file name) is known, so a simple search (e.g., with the file system search engine) will suffice. In many other cases models must be searched using more complex criteria, such as:

- **An internal characteristic** such as models with an element with a given value, metamodels containing elements of certain types, etc.
- **A computed characteristic** based on the models size or structure, such as models with more than two hundred elements or transformations that contain more imperative transformation excerpts than others etc.
- **Their relations with other artefacts**, such as models that are related to a given transformation or to other models e.g. by a trace model.

After finding the desired models, we may need to extract and combine their elements. The combination of information from several models is essential for extracting metrics from model repositories. For instance, we may want to compute the number of metamodel elements a transformation uses, or the number of models and elements involved in a model weaving etc.

Another common requirement when working on MDE repositories is to be able to execute batch processes of regular modelling tasks (e.g. transformations, model checks, projections etc.) involving large amounts of models. Furthermore, these batch processes should orchestrate the modelling tasks according to how the models are arranged in the repository. For instance, when a model is modified only the transformations which use the modified model should be executed and then all the transformations that use the output models of the executed transformations and so on.

Finally, since several users manipulate the repositories, they are in constant evolutionary state. Thus, it is important to have an updated view of the repository and to count with mechanisms for easily detect changes in the repository. For instance suppose there is a batch process that re-executes the transformations in the repository when their input models change. Now, if a user contributes a new transformation to the repository, the process will not be able to re-execute the new transformation if it does not rely on an updated view that reflects the new transformation and also its relation with its input models. Therefore, we require mechanisms for easily register, update and delete artefacts from such a repository view.

In the next sections we will see how MoScript enable us to perform all these tasks.

3 The MoScript Architecture

Fig. 1 shows an overview of the MoScript architecture, comprising both the basic components and information flows.

3.1 Architecture Components

The MoScript architecture is composed of six components: the MoScript DSL, a megamodel, a metadata engine, model repositories, transformation tools, and external DSLs, editors, and discoverers, as shown in Fig. 1 and described next.

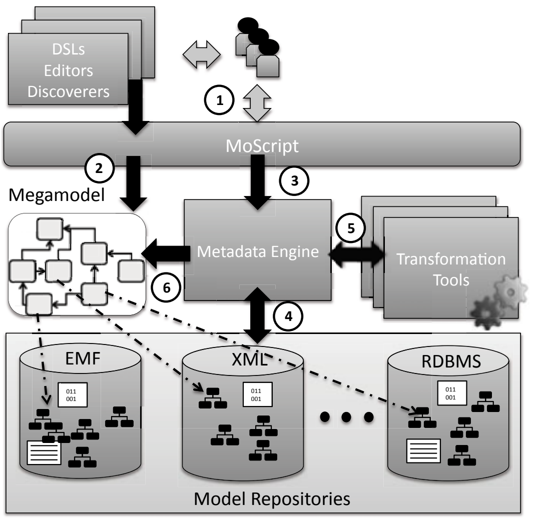


Fig. 1. The MoScript architecture

- **MoScript:** A textual DSL, which serves as an interface between the users and the modelling artefacts repositories. Users write and run their MoScript scripts for retrieving modelling artefacts and performing modelling tasks (e.g. inspect, transform, match, etc.) with them. MoScript uses the megamodel as cartography to navigate the repositories and select modelling artefacts to manipulate. As result of the manipulations, new modelling artefacts may be created in the repositories or existing modelling artefacts may be removed.

– **Megamodel**[3]**:** A model which describes artefacts within repositories (e.g. their location, kind, format, etc.), and how they are interrelated. A megamodel is a regular model, thus it conforms to a metamodel, which is shown in Fig. 2. For instance, the ***Entity*** element represents any MDE (i.e. artefacts that depend on well defined grammars) and non-MDE artefact (such as non structured documents, tools, libraries, etc).

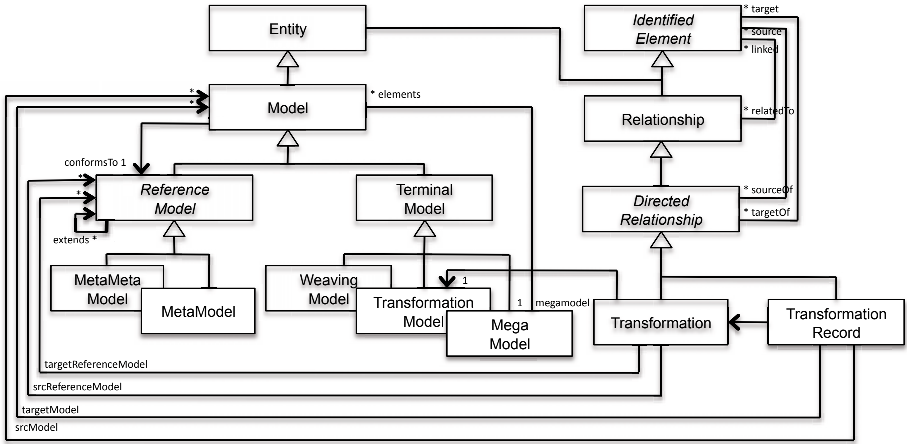


Fig. 2. Part of the core metamodel for megamodels

The basic MDE artefacts supported by the megamodel are: ***MetaMetaModel*s (M3)**, which represent models conforming to themselves; ***Metamodel*s (M2)**, which represent models conforming to metametamodels; and ***TerminalModel*s (M1)**, which represent models conforming to metamodels but no other model conforms to them. Examples of *TerminalModels* are ***TransformationModel*s**, ***WeavingModel*s** and ***Megamodel*s** themselves. Relationships between artefacts (MDE and non-MDE) are represented by the ***Relationship*** concept. For instance, a ***Transformation*** is a directed relationship between a *TransformationModel* and one or more ***ReferenceModel*s** (metamodels or metametamodels). The *Transformation-Model* is the representation of the source code of the transformation while the *ReferenceModels* restrict the type of input and output models the transformation may be applied on. A ***TransformationRecord*** is another kind of directed relationship. A *TransformationRecord* associates a *Transformation* with a set of input and output models. As we will see later, it is useful for rerunning transformations without giving any additional input.

Since MDE artefacts may be bridged to models (e.g. XMI files), from this point forward we are going to call them just *models*.

– **Metadata Engine:** Provides services to MoScript for retrieving models, executing tools services and (un)register models (from) into the megamodel. The Metadata Engine exposes a homogeneous interface, which provides location and technology transparency of models and transformation tools. It also protects models from unauthorized access and modifications.

The metadata engine uses the megamodel for run-time type checking. For instance, the metadata engine can check if the transformations are being applied to the right models. In a previous work [24], we demonstrate the viability of this type checking.

– **Model repositories:** Contain models stored in different formats, e.g. XMI, XML, RDBMS, etc. Model repositories may reside in different physical locations, such as a local filesystem, a remote WebDAV server, the cloud, etc.
– **Transformation Tools:** Model-to-model (M2M), model-to-text (M2T) or text-to-model (T2M) transformation tools provide transformation services. They implement a generic interface, thus all transformation tools services can be invoked the same way regardless the technology behind. Transformation tools may include QVT [1], ATL [14], Kermeta,[1] EMF Compare,[2] Xpand,[3] etc. In general, any tool that produces a new view of a modelling artefact (e.g. documentation generators, compilers, file comparison tools, etc.) is considered a transformation tool. If any transformation tool does not fit the generic interface it may extend it along with the metamodel of the megamodel, for adding new services and concepts.
– **DSLs, Editors and Discoverers:** These tools create models outside the MoScript context and need to contribute them to the megamodel. They can (un)register models (from) into the megamodel through MoScript, and they can query the megamodel as well.

3.2 Architecture Information Flow

The information flow that takes place between the architecture components when performing models manipulations with MoScript, is denoted by the numbers in Fig. 1. (1) Users write and run a MoScript script. (2) MoScript queries the megamodel to retrieve the model elements (metadata) describing the models and transformations involved in the process. Then, it (3) asks the Metadata Engine to apply the selected transformations on the selected models. (4) The metadata engine retrieves[4] from the repositories the models and transformation definitions (using the information stored in the megamodel elements, such as location, protocol, access restrictions etc). (5) Then it executes the transformations with the models and (6) registers the resulting models in the megamodel if necessary. Finally, the metadata engine returns to MoScript the model elements of the megamodel resulting from the program execution, for further processing.

[1] http://www.kermeta.org/
[2] http://www.eclipse.org/modeling/emf/?project=compare#compare
[3] http://www.eclipse.org/modeling/m2t/?project=xpand
[4] Retrieving the model means that an interface (model handler) is exposed for accessing the model. It does not necessarily means that the whole model traverses the network.

4 The MoScript Language

MoScript is a megamodel-based scripting DSL for modeling tasks and workflow automation that uses OCL [2] as query language.

A megamodel is a regular model and thus can be navigated with standard OCL, however the result of executing an OCL query on it, is merely informative. For instance, consider the following query:

```
Model::allInstances()->select(m | m.conformsTo.kind = 'Java')
```

The query selects from a megamodel, all the models that conform to a specific kind of metamodel. The result is a collection of elements of type *Model* that cannot be used directly in OCL to access or manipulate (check, match, transform etc.) the physical artefacts they represent. This issue is due to the fact that OCL does not handle models as a bootstrapped concept and does not have multi-model support either.

The MoScript language intends to fill this gap with three main contributions: **(1) Model dereferencing**, to retrieve models represented by metadata in a megamodel; **(2) Extensive library of generic operations** to perform common model manipulation tasks with dereferenced models; **(3) Modelling tasks operation composition** combined with OCL for manipulating dereferenced models with powerful expressiveness.

Model dereferencing is applicable to all the megamodel elements that have a separated physical representation in the system and may be accessed through a locator (e.g., an URI). As a result of the dereferencing, an interface of the model is loaded in memory and exposed for being used through an OCL *ModelElement* type. Since OCL works on top of the megamodel, the OCL *ModelElement* type always corresponds to an element type of the megamodel (*TerminalModel*, *Metamodel*, *Transformation* etc.).

Furthermore, a set of operations are associated to those model element types for being invoked from OCL and which in turn may be composed as any other OCL expression, to perform more complex operations.

Next, we will explain in detail MoScript abstract and concrete syntax, as well as its native library of operations and statements.

4.1 MoScript Abstract and Concrete Syntax

The MoScript DSL has a semantic model [11] and an abstract and concrete syntax [22].

The MoScript's **semantic model** is the megamodel. It is the place where the domain concepts are stored and is independent from the language constructs. The core concepts of the megamodel have been covered in section 1.

The **abstract syntax** as shown in figure 3, is divided in two packages. The OCL package and the MoScript package. Since MoScript uses OCL as query language, the complete OCL abstract syntax (not showed) is included as part of the language.

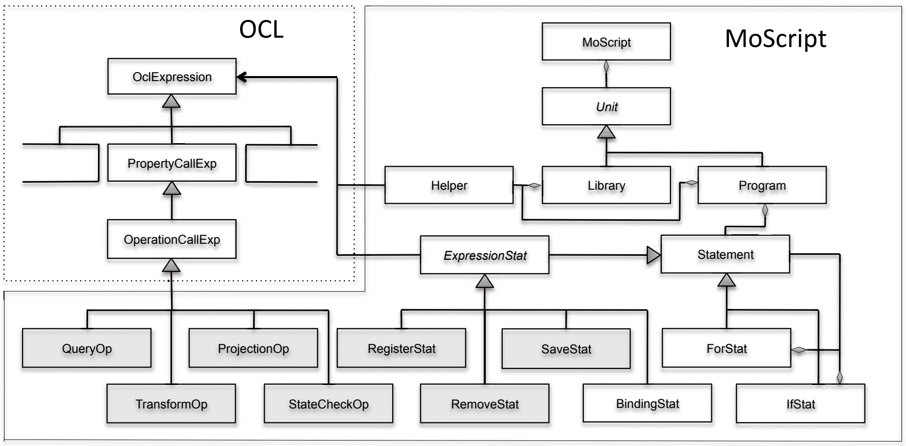


Fig. 3. MoScript abstract syntax main concepts

The `OperationCallExp` from the OCL package has been extended with a set of operations we call *operations without side effects*. These operations are used to perform several modelling tasks that **do not modify the model repository or the megamodel**.

Operations without side effects are divided in four categories: query operations (`QueryOp`), operations for transformations between same technical spaces (`TransformOp`), operations for transformations between different technical spaces (`ProjectionOp`) and operations for checking the models state (`StateCheckOp`). For each category MoScript provide several concrete operations, which will be explained in the next subsection.

The MoScript package also provides a set of *statements with side effects* (`SaveStat, RemoveStat and RegisterStat`). **These statements allow the modification of the repository or the megamodel.** Side effects statements may embed OCL expression and therefore side effects free operations. This is why `ExpressionStat` is related to `OCLExpression`. This relation allows to carry out complex models manipulations before persisting them in the repository and the megamodel. However, the opposite (embed side effects statements within OCL expressions) is not permitted. OCL expressions do not know side effects statements, thus respecting the OCL side effects free philosophy.

MoScript also provides a statement for variable declaration and value binding (`BindingStat`) and `for` (`ForStat`) and `if` (`IfStat`) statements for control flow.

MoScript has two kinds of modules: **libraries** and **programs**. A library contains **helpers**, which are used to modularise complex OCL expressions. Libraries may be in turn imported by programs or by other libraries.

The **concrete syntax** of MoScript is summarised in the following listing:

```
program program_name

uses library
...
[using  {
    variable : type = OclExpr; ...
}]

do {
  variable <- OclExpr;

  save(...); ...
  remove(OclExpr); ...
  register(...); ...

  if...
  for..
}

helper context OclAny def: helper_name(params) : return_type; ...
```

A program has two sections, the `using` and `do` sections. The `using` section is optional, and is used for declaring variables and assigning their initial value. The `do` section is mandatory and is the core of the program. In it, operations without side effects and side effects statements are used in combination with control flow statements and OCL queries to perform modelling artefacts manipulations.

The complete definition of the concrete syntax is expressed in the TCS language [15], and can be found at:
`http://www.emn.fr/z-info/atlanmod/index.php/Moscript`.

In the following subsections, we will discuss in detail the operations without side effects and the statements with side effects provided by MoScript and summarised in table 1.

Table 1. MoScript operations and statements summary

Operations without Side Effects
Model :: allContents() : Collection(OclAny)
Model :: allContentsRoots() : Collection(OclAny)
Model :: allContentsInstancesOf(type_name : String) : Collection(OclAny)
Model :: allContentsInstancesOf(type : OclAny) : Collection(OclAny)
Transformation :: applyTo(inputModels : Sequence(Model)) : TransformationRecord
Transformation :: applyTo(inputModels : Map(String, Model)) : TransformationRecord
TransformationRecord :: run() : TransformationRecord
Model :: inject() : Model
Model :: extract() : Model
Model :: available() : Boolean
Model :: isDirty() : Boolean

Statements with Side Effects
save(m : Model, mm : Megamodel, id : String, locator : String)
remove(m : Model, mm : Megamodel)
register(mm : Megamodel, id : String, locator : String)

4.2 Operations without Side Effects

This subsection describes in detail the operations without side effects provided by MoScript. As mentioned before, operations without side effects are classified in four categories, queries, transformations of models in a same technical space, transformations of models between different technical spaces and model state checkers.

Query Operations: The query operations provided by MoScript are `allContents`, `allContentsRoots` and `allContentsInstancesOf`. These operations dereference and load the physical model represented by the *Model* element. Then, they query the model and return a collection of OCL elements. The elements of the resulting collection are used as entry points to the model, from where the rest of the elements may be reached. Subsequent queries to the model are made with standard OCL expressions. The following example illustrates how this operations may be used in general:

```
Model::allInstances()->any(m | m.indentifier = 'SimpsonFamily')
    ->allContents()->collect(c | c.name))
```

In the example, we select a model with the "SimpsonFamily" id from the repository, and invoke the `allContents` operation on it. The operation dereferences de model and returns an OCL collection with all the elements contained in the model. Next, we iterate on the results, collecting all the element names. The resulting collection should look like {`'Bart', 'Homer', 'Lisa', 'Maggie', 'Marge'`}.

Note that the `allContents` operation hides complexity from the user. There is no need to specify the metamodel of the model as this information is retrieved from the megamodel.

When working with big models the operation `allContents` may be expensive in terms of memory consumption and processing. So, MoScript includes other operations like `allContentsRoots` and `allContentsInstancesOf` for extracting the models elements with more precision and therefore better performance.

Model to Model Transformations: The M2M transformations operations provided by MoScript are the `applyTo` and the `run` operations.

The `applyTo` operations work in the context of the *Transformation* megamodel element. They input models may be provided as a Map or as a Sequence and the output models are returned as part of a *TransformationRecord*. When provided as a Map, models are differentiated by their key and when provided as a Sequence, models are differentiated by their order in the Sequence.

The `applyTo` operations are especially useful if we consider transformations that are somehow generic (e.g., a transformation which transforms a Java source code model to a .Net source code model), i.e. there may exist lots of different models that may be transformed with the same transformation. In this case it is very convenient to have a way for varying the input models for each transformation execution. The following example illustrates how these operations may be used:

```
let j2dNet : Transformation =
    Transformation::allInstances()->any(t | t.identifier = 'j2dNet')
in
    TerminalModel::allInstances()
```

```
        ->select(m | m.conformsTo.kind = 'Java'))
        ->collect(jModel | j2dNet.applyTo(jModel))
```

In the example we first retrieve the transformation "Java to .Net" from the repository and store it as `j2dNet`. Then we apply `j2dNet` to all the Java models found in the repository. Note that behind the scenes, the metadata engine makes several checks before running the transformation. First, it checks if the model is a transformation model, and thus may be executed. Then, it checks if the input models conform to the metamodels the transformation supports. Finally, it determines which is the right transformation engine[5] for running the transformation. To do this, the metadata engine queries the megamodel.

The `run` operation works in the context of the *TransformationRecord* megamodel element. The `run` operation executes a transformation based on the information stored in the *TransformationRecord*. Since it stores the last transformation execution parameters, it is useful to rerun transformations without specifying the input models. The operation returns the newly produced models within another *TransformationRecord*.

The following example shows how it is possible to rerun all the transformations of a model repository:

```
TransformationRecord::allInstances()->collect(tr | tr.run())
```

Projectors: As we are working with heterogeneous model repositories, we rely on *technical projectors* for non-XMI modelling artefacts (e.g. grammar-based text). There are two kinds of projectors: injectors and extractors. Injectors translate from other technical spaces (e.g. grammarware[17], xmlware, etc) to the modelware technical space and extractors do exactly the opposite. MoScript provides the `inject` operation for injecting models and the `extract` operation for extracting models.

The `inject` operation represents the T2M transformations. It works in the context of the *Model* element, which represents a non-XMI artefact that depends on a specific grammar. The inject operation applies the transformation to the model and produces an XMI model. The following example shows how is possible to inject the source code of Java programs into Java XMI models:

```
Model::allInstances()->select(m | m.conformsTo.kind = 'JavaGrammar'))
    ->collect(jCode | jCode.inject())
```

In the example, we select all the Java models which conform to the Java grammar and inject them into models conforming to Java metamodels. The result is a collection of Java XMI models. Behind the scenes, the Metadata Engine retrieves from the megamodel the corresponding parser[6] of the grammar and the tool that uses it, to produce the XMI model.

The `extract` operation represents the M2T transformations and uses the same mechanism as the `inject` operation, but in the opposite direction.

For both operations we follow an approach similar to the one described in [25].

[5] Required relations not showed in Fig.2

[6] Required relations not showed in Fig.2

Models State Checkers: A set of consistency check utility operations have been included in the language. The `available` operation, which verifies if the modelling artefact is available in the repository (e.g., it could have been removed by an external tool, or its physical location is unreachable), and the `isDirty` operation, which checks if the model has been modified outside MoScript. This is useful to know if it is necessary to re-execute the transformations in which the model participates.

4.3 Statements with Side Effects

This subsection describes in detail the statements with side effects provided by MoScript. As said before, these statements allow the modification of the models in the repository and the megamodel. These statements are usually combined with OCL expressions and operations without side-effects.

save. The `save` statement persists an in-memory model into the repository and registers it in the megamodel if it is not already registered. The latter step is important for keeping integrity between the megamodel and the repository. The `save` statement takes as arguments the *Model* to be persisted, the *megamodel* in which the model should be stored[7], an identifier and a locator. The locator argument is the physical location path where the model should be stored (e.g. a filesystem path or URI).

Suppose we want to store the .Net models derived from Java models showed in a previous example. The following example shows how the `save` statement can be used for this purpose:

```
... for(dNetModel in dNetModels) {
      save(dNetModel, this, dNetModel.getIdentifier(),
           dNetModel.location + '.xmi');
    } ...

helper context Model def: getIdentifier(): ...;
```

In the example, we iterate over the collection of .Net models and persist them in the repository. We use a helper to produce the identifiers of the models. The `this` keyword means that the model will be stored in the root megamodel.

Register. The `register` statement allows the registration of models in the megamodel when the model is already stored in the repository. It takes as arguments the *megamodel*, the model identifier, its physical location and creates the corresponding megamodel element.

The `register` statement is the statement other tools (e.g. editors, discoverers, DSLs, etc.) use to register the artefacts created outside the MoScript context. For instance, manually created models, discovered models, etc. The following example shows how it is possible with MoScript to register a new metametamodel:

```
register(this, 'Ecore', 'http://www.eclipse.org/emf/2002/Ecore');
```

[7] Remember a megamodel may contain other megamodels.

The metadata of a model already registered in the megamodel can be updated by re-invoking the `register` statement. For instance, when another tool changes the location of a model.

Remove. The `remove` statement allows the removal of models from the repository. It also removes the model element from the megamodel in order to maintain consistency between both. It receives as argument the *megamodel* and the *Model* to be eliminated.

5 Putting All Together

In this section we provide examples of complete MoScript scripts which demonstrate the power of the language.

5.1 Change Propagation

Roughly speaking, Model Driven Development (MDD) consists in transforming models from higher levels to lower levels of abstraction until the generation of code, in order to produce runnable systems.

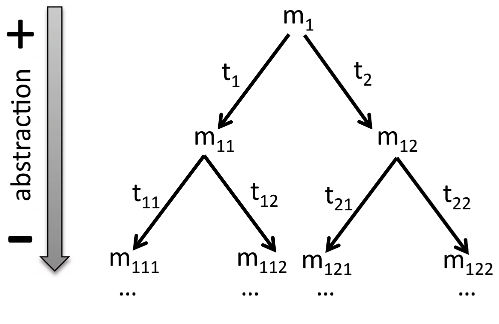


Fig. 4. An MDD system transformation chains

Now, suppose we have an MDD based system, which has a binary tree like arrangement of models and transformations, as shown in Fig. 4. In the figure, models are denoted by the m nodes and transformations by the t directed edges. Now, if model m_1 changes we will have to re-execute all the transformations that are directly or indirectly affected by the change in the model, in order to reflect the change in the models of the lowest level of abstraction (the code). The MoScript program in listing 4 shows how to do it.

Listing 1.1. Change propagation

```
program PropagateChanges

do {

    im : Model = Model::allInstances()->any(m | m.identifier = 'm1');

```

```
 7     for(tr : getTransformations(m)) {
 8         om : Model = tr.run().targetModel->first();
 9         save(om, this, om.identifier, om.locator);
10     }
11 }
12
13 helper def: getTransformations(m :Model) :Sequence(
       ↪TransformationRecord) =
14     trs : Sequence(TransformationRecord) = TransformationRecord::
           ↪allInstances()->select(tr | tr.srcModel->first().identifier
           ↪ = m.identifier) in
15     if trs->isEmpty() then
16         Sequence{}
17     else
18         trs->union(trs->collect(tr | getTranformations(tr.targetModel
               ↪->first()))->flatten())
19     endif;
```

The explanation of the code is the following:

I We select from the repository the modified model by its id (line 5).
II We call the helper `getTransformations` (line 7) to return a collection of *TransformationRecord*s in the order they must be executed.
III The `getTransformations` helper selects all the *TransformationRecord*s that use model `m` as input of its transformation (line 14).
IV For each *TransformationRecord*, the output model of its transformation is selected, and a recursive call is made to the `getTransformations` helper, in order to go through the tree in depth, getting the rest of the *TransformationRecord*s (line 18).
V Finally, for each *TransformationRecord* its transformation is executed (line 8), its resulting model saved in the repository and updated in the megamodel (line 9)

Note that the example is an intentional over simplification of real case models and transformations arrangements, in order to keep the code simple. We assumed the transformations have only one input and output model and no cycles between them.

5.2 Inspecting and Combining Models Information

In this example we show how we can combine information from several models and make computations for obtaining measurements from the model repository. We will compute a measure for determining the transformations *naive completeness*[8] of all the transformations in the repository.

A transformation t_1 is *naively complete* if all the elements of its source metamodel mm_1 and its target metamodel mm_2 are matched (used) by at least one rule of the transformation.

To illustrate our definition of naive completeness, suppose we have a transformation and its models $t_1(m_1) = m_2$ where m_1 is the input model and m_2

[8] Checking whether a transformation is actually complete or not is much more complex.

is the output model. m_1 and m_2 conform to the metamodels mm_1 and mm_2 respectively. A transformation is a finite set of rules $t_1 = (r_1, r_2, ..., r_n)$. Each rule has 1 or none input element or pattern and has at least one output element $r() = (op_1, op_2, ..., op_n)$ or $r(ip) = (op_1, op_2, ..., op_n)$. The input and output patterns correspond to elements of the metamodels.

To determine which elements of mm_1 and mm_2 are used in the transformation, we will inspect its transformation rules. For each rule of t_1 we will verify the number of elements from mm_1 (e_{mm1}) present as input pattern (ip_{t1}) in at least one rule of the transformation. Number of elements from mm_2 (e_{mm2}) present as output patterns (op_{t1}) in at least one rule of the transformation.

The result of the measurement is calculated for mm_1 as
$Cr_{mm1} = \sum(ip_{t11}, ip_{t12}, \ldots, ip_{t1n}) / \sum(e_{mm11}, e_{mm12}, \ldots, e_{mm1n})$ and the same for mm_2 but with the output patterns and the output metamodel. The transformation is considered naively complete if $Cr_{mm1} = 1$ and $Cr_{mm2} = 1$.

Listing 1.2 shows the OCL query for obtaining the described measures. For the sake of simplicity we inspect only ATL matched rules, which are fully declarative and always have an input element. We also assume that all the metamodels conform to Ecore.

Listing 1.2. Transformation completeness query

```
program TransformationCompleteness

do {
    res : Sequence(OclAny) = getNaiveCompleteness();
    -- Do something with the result
}

helper def getNaiveCompleteness(): Sequence(OclAny) =
    Transformation.allInstances()->collect(tr |
        let trName : String = tr.transformationModel.name in
        let mmIn : Set(String) = tr.srcReferenceModel
            ->collect(e | e.referenceModel.allContentInstancesOf('
                ↪EClass'))
            ->flatten()->collect(e | e.name).asSet() in
        let mmOut : Set(String) = tr.targetReferenceModel
            ->collect(e | e.referenceModel.allContentInstancesOf('
                ↪EClass'))
            ->flatten()->collect(e | e.name).asSet() in
        let trIn : Set(String) = tr.transformationModel.inject().
            ↪allContentInstancesOf('MatchedRule')
            ->collect(x | x.inPattern.elements
            ->collect(y | y.type.name))->flatten().asSet() in
        let trOut : Set(String) = tr.transformationModel.inject().
            ↪allContentInstancesOf('MatchedRule')
            ->collect(x | x.outPattern.elements
            ->collect(y | y.type.name))->flatten().asSet() in
        let inRt : Real = trIn->size() / mmIn->size() in
        let outRt : Real = trOut->size() / mmOut->size() in
        Sequence(trName, inRt, outRt)
    );
```

Due to space limitations we do not explain the code in detail, but note that doing these kind of computations without MoScript will demand a lot of work with existing scripting techniques or adhoc codifications.

6 Implementation

In this section, we describe our implementation of MoScript. Figure 5 shows how we made the instantiation of the architecture presented in section 3.

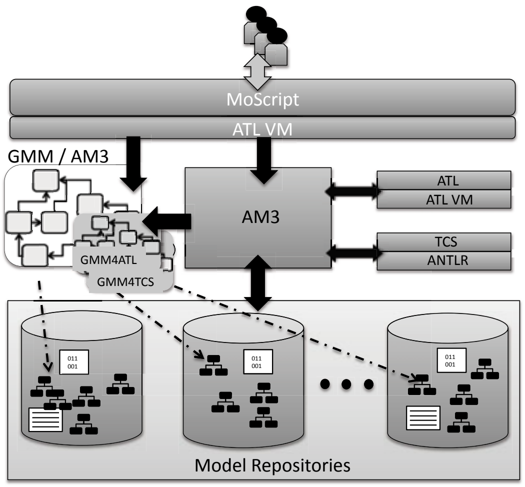


Fig. 5. MoScript architecture implementation

As concrete implementation, we use our previous implementation of the megamodel included in the AM3 tool[3]. AM3 follows the megamodel definition as shown in Fig. 2, plus two extensions that support M2M and M2T-T2M transformation in ATL and TCS respectively. The megamodel extension for ATL is called GMM4ATL and the extension for TCS is called GMM4TCS. As Metadata Engine, we use the AM3 tool metadata layer. As transformation engines we use ATL and TCS. TCS performs T2M transformations by generating an ANTLR[9] grammar and performs M2T using Java-based extractors or ATL OCL queries.

MoScript has been implemented on top of the Eclipse Modeling Platform. We use TCS as well, for defining its abstract and concrete syntax. TCS is in charge of parsing and lexing MoScript to populate an abstract syntax tree (AST) model ready for compilation. We built the MoScript compiler with ACG[10], which is the ATL VM Code Generator. It translates the AST model (generated by TCS) into ATL VM assembly code for its execution.

Note that ATL and the ATL VM are two different concepts. ATL is a DSL for transformations which is compiled in ATL VM code. Other DSLs may run on top of the ATL VM as is the case of MoScript.

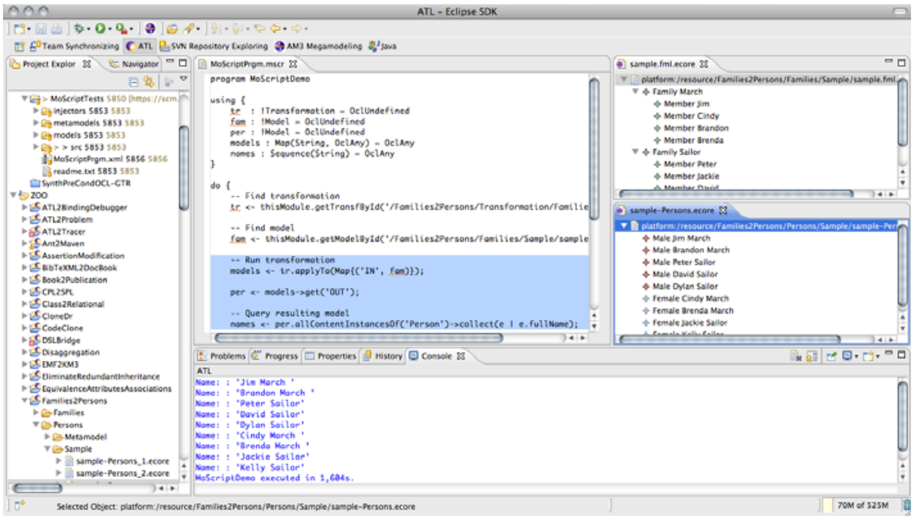
The concrete architecture uses two instances of the ATL virtual machine. One instance for MoScript and another one for ATL. This guarantees that MoScript operates independently of ATL and other transformation tools.

[9] http://www.antlr.org/

[10] http://wiki.eclipse.org/ACG

We tested MoScript with the ATL Transformations Zoo[11]. A model repository of ATL transformation projects developed by the Eclipse community. It holds so far 205 metamodels, 275 models, 219 transformations, and more than 400 other artefacts including textual syntaxes, binary code, source code, libraries, etc. We also tested MoScript with a WebML [8] repository, where models are stored in XML.

In fig. 6 we show a screen shot of a running MoScript script in Eclipse. The current implementation of MoScript can be downloaded from `http://www.emn.fr/z-info/atlanmod/index.php/Moscript_downloads`.


```
program MoScriptDemo

using {
    tr  : !Transformation = OclUndefined
    fam : !Model = OclUndefined
    per : !Model = OclUndefined
    models : Map(String, OclAny) = OclAny
    names : Sequence(String) = OclAny
}

do {
    -- Find transformation
    tr <- thisModule.getTransfById('/Families2Persons/Transformation/Familie

    -- Find model
    fam <- thisModule.getModelById('/Families2Persons/Families/Sample/sample

    -- Run transformation
    models <- tr.applyTo(Map{('IN', fam)});

    per <- models->get('OUT');

    -- Query resulting model
    names <- per.allContentInstancesOf('Person')->collect(e | e.fullName);
```

```
Name: : 'Jim March '
Name: : 'Brandon March '
Name: : 'Peter Sailor'
Name: : 'David Sailor'
Name: : 'Dylan Sailor'
Name: : 'Cindy March '
Name: : 'Brenda March '
Name: : 'Jackie Sailor'
Name: : 'Kelly Sailor'
MoScriptDemo executed in 1,604s.
```

Fig. 6. Running MoScript script

7 Related Work

The concept of a megamodel was proposed in [5] and in [10]. In [5] the megamodel is proposed a solution to Global Model Management (GMM) while in [10] it is presented as a metamodel for describing MDE formalized with set theory. Several recently works report use of megamodels or megamodeling techniques. For instance, in [12] a megamodel is used for the representation of all the artefacts and their relationships involved in the model-driven support for the evolution of software architectures. In [20], Megaf an infrastructure for the creation of architecture frameworks formalizes its underlying infrastructure through a megamodel for checking consistency among architectural elements. The 101company[12] project is an effort to create a conceptual framework based on megamodeling techniques for understanding analogies between heterogeneous technologies. In general, all the

[11] `http://www.eclipse.org/m2m/atl/atlTransformations/`
[12] http://101companies.uni-koblenz.de/index.php/Main_Page

mentioned works focus on representing high number of different artefacts and technical spaces involved in non trivial software systems and development processes. MoScript uses the megamodel with the same general intend.

As far as we know, there are not many DSLs or approaches for GMM, i.e. they do not use a megamodel as a global view for orchestrating and verifying MDE development activities against it. However, we find similarities with approaches such as Rondo [21], Maudeling[13], Model Bus [6] and Moose [18]. Rondo, Maudeling and Moose translate models to their own internal formats, whereas Model Bus and MoScript work directly on the models via a metadata engine. Rondo represents models as directed labeled graphs. Maudeling represents models in the Maude language [9], which is based on rewriting logic. Moose represents models in CDIF or XMI exachange formats conforming to the FAMIX metamodel using third party parsers. Rondo translates between different model representations of the same information, and operates on a lower level than MoScript: it directly manipulates the model artefacts, whereas MoScript relies on the invocations of transformation engines. Maudeling provides advanced querying services on modelling artefacts, and as such, could be an invokable service for MoScript. Moose offers services for navigating and manipulating multiple model versions and uses Pharo [14] (Smaltalk) as scripting language. Model Bus provides a modelling artefact broker service, where registered tools can be applied to registered models. Model Bus does not provide a megamodel concept to look up model and tool metadata. MoScript uses a reflective approach, and queries the megamodel to check if specific modelling artefacts may be used in combination.

Model search engines such as those presented in [7] and [19], are also related to GMM in that they can perform large-scale model queries, based on model contents. They differ from our approach in that the results obtained from a model search cannot be directly used in further modelling operations. The results are usually shown as a list of model names or model fragments which at most can be downloaded.

MoScript is intended to implement MDE workflows. The main difference with other MDE workflow approaches is that MoScript relies on a query language that works on the rich contents of a megamodel and orchestrations are based on its queries results. Other MDE workflow approaches are UniTI [23], TraCo [13], the Modeling Workflow Engine (MWE)[15], and MDA Control Center [16]. UniTI composes transformation processes via typed input and output parameters. Compositions are validated based on model type information and any additional constraints that can be specified on the models. TraCo uses a component metamodel, with components and ports, where each workflow component is wired to other components via its input and output ports. Ports are typed in order to validate the compositions. MWE is a model-driven version of Ant[16], with several builtin tasks for model querying and transformation. MWE

[13] Maudeling: http://atenea.lcc.uma.es/index.php/Main_Page/Resources/Maudeling

[14] http://www.pharo-project.org/home

[15] http://www.eclipse.org/modeling/emft/?project=mwe

[16] http://ant.apache.org

does not perform any validation of the workflow composition. MoScript does not perform a static type check on its workflow compositions either, but checks the validity of the composition at run-time.

8 Conclusions and Future Work

In this paper, we presented MoScript: a scripting DSL and platform for Global Model Management (GMM), based on the notion of a megamodel.The MoScript architecture provides uniform access to modelling artefacts, such as models, metamodels, and transformations, regardless of their storage format or their physical location. It also provides bindings to several model manipulation tools, such as transformation engines and querying tools, and allows invocation of those tools.

The MoScript is an OCL-based scripting language for model-based task and workflow automation, based on the metadata contained in a megamodel. It allows querying a megamodel and use the results of such queries to load and store modelling artefacts, and perform model manipulations, such as the invocation of a model transformation engine. MoScript can use the rich metadata in the megamodel to validate model manipulations, e.g. to check if a model transformation is applied to a model that conforms to the right metamodel. MoScript is able to perform this validation at run-time, when the model manipulation is invoked.

MoScript has been implemented on top of the Eclipse Modeling Platform, using TCS, ACG tools and AM3 metadata engine. It provides a textual and uses the ATL virtual machine and debugger as its run-time environment. MoScript uses ATL as M2M and TCS as M2T-T2M transformation engines. MoScript implementation has been tested against models from the ATL examples repository and a WebML repository.

As further work we plan to extend the list of repositories and tools our language can interact with, and increase the number of predefined operations and statements of the language. This may include a querying tool, such as Maudeling, that allows us to validate modelling workflows written in MoScript.

References

1. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, version 1.0, formal/08-04-03 (Apr 2008), http://www.omg.org/spec/QVT/1.0/PDF/
2. OCL 2.2 Specification, version 2.2, formal/2010-02-01 (February 2010), http://www.omg.org/spec/OCL/2.2/PDF
3. Allilaire, F., Bezivin, J., Bruneliere, H., Jouault, F.: Global model management in eclipse gmt/am3. In: Proc. of the Eclipse Technology eXchange Workshop (eTX) at ECOOP 2006 (2006)
4. Barbero, M., Jouault, F., Bézivin, J.: Model driven management of complex systems: Implementing the macroscope's vision. In: Proc. of ECBS 2008. IEEE Computer Society Press (2008)

5. Bezivin, J., Jouault, F., Valduriez, P.: On the need for megamodels. In: Proc. of Workshop on Best Practices for Model-Driven Software Development at the 19th Annual ACM Conference on OOPSLA (August 2004)
6. Blanc, X., Gervais, M.-P., Sriplakich, P.: Model Bus: Towards the Interoperability of Modelling Tools. In: Aßmann, U., Aksit, M., Rensink, A. (eds.) MDAFA 2003. LNCS, vol. 3599, pp. 17–32. Springer, Heidelberg (2005)
7. Bozzon, A., Brambilla, M., Fraternali, P.: Searching Repositories of Web Application Models. In: Benatallah, B., Casati, F., Kappel, G., Rossi, G. (eds.) ICWE 2010. LNCS, vol. 6189, pp. 1–15. Springer, Heidelberg (2010)
8. Ceri, S., Brambilla, M., Fraternali, P.: The History of WebML Lessons Learned from 10 Years of Model-Driven Development of Web Applications. In: Borgida, A.T., Chaudhri, V.K., Giorgini, P., Yu, E.S. (eds.) Conceptual Modeling: Foundations and Applications. LNCS, vol. 5600, pp. 273–292. Springer, Heidelberg (2009)
9. 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)
10. Favre, J.M.: Towards a basic theory to model model driven engineering. In: 3rd UML Workshop in Software Model Engineering (WISME 2004) Joint Event with UML 2004 (October 2004)
11. Fowler, M.: Domain-Specific Languages, 1st edn. Addison-Wesley Professional (October 2010)
12. Graaf, B.: Model-driven evolution of software architectures. In: European Conference on Software Maintenance and Reengineering (CSMR 2007), pp. 357–360 (2007)
13. Heidenreich, F., Kopcsek, J., Aßmann, U.: Safe Composition of Transformations. In: Tratt, L., Gogolla, M. (eds.) ICMT 2010. LNCS, vol. 6142, pp. 108–122. Springer, Heidelberg (2010)
14. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: Atl: A model transformation tool. Science of Computer Programming 72(1-2), 31–39 (2008)
15. Jouault, F., Bézivin, J., Kurtev, I.: Tcs: a dsl for the specification of textual concrete syntaxes in model engineering. In: GPCE 2006: Proc. of the 5th Int. Conf. on Generative Programming and Component Engineering, pp. 249–254. ACM (2006)
16. Kleppe, A.: MCC: A Model Transformation Environment. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 173–187. Springer, Heidelberg (2006)
17. Klint, P., Lämmel, R., Verhoef, C.: Toward an engineering discipline for grammarware. ACM Trans. Softw. Eng. Methodol. 14, 331–380 (2005)
18. Laval, J., Denier, S., Ducasse, S., Falleri, J.R.: Supporting Simultaneous Versions for Software Evolution Assessment. Journal of Science of Computer Programming (December 2010)
19. Lucrédio, D., de M. Fortes, R.P., Whittle, J.: MOOGLE: A Model Search Engine. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 296–310. Springer, Heidelberg (2008)
20. Malavolta, I.: A model-driven approach for managing software architectures with multiple evolving concerns. In: Proc. of European Conference on Software Architecture: Companion Volume, ECSA 2010, pp. 4–8. ACM (2010)

21. Melnik, S., Rahm, E., Bernstein, P.A.: Rondo: a programming platform for generic model management. In: Proc. of SIGMOD 2003, pp. 193–204. ACM (2003)
22. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. 37(4), 316–344 (2005)
23. Vanhooff, B., Ayed, D., Van Baelen, S., Joosen, W., Berbers, Y.: UniTI: A Unified Transformation Infrastructure. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MoDELS 2007. LNCS, vol. 4735, pp. 31–45. Springer, Heidelberg (2007)
24. Vignaga, A., Jouault, F., Bastarrica, M.C., Brunelière, H.: Typing in Model Management. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 197–212. Springer, Heidelberg (2009)
25. Wimmer, M., Kramler, G.: Bridging Grammarware and Modelware. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 159–168. Springer, Heidelberg (2006)

Reconstructing Complex Metamodel Evolution

Sander D. Vermolen, Guido Wachsmuth, and Eelco Visser

Software Engineering Research Group, Delft University of Technology, The Netherlands
{s.d.vermolen,g.h.wachsmuth,e.visser}@tudelft.nl

Abstract. Metamodel evolution requires model migration. To correctly migrate models, evolution needs to be made explicit. Manually describing evolution is error-prone and redundant. Metamodel matching offers a solution by automatically detecting evolution, but is only capable of detecting primitive evolution steps. In practice, primitive evolution steps are jointly applied to form a complex evolution step, which has the same effect on a metamodel as the sum of its parts, yet generally has a different effect in migration. Detection of complex evolution is therefore needed. In this paper, we present an approach to reconstruct complex evolution between two metamodel versions, using a matching result as input. It supports operator dependencies and mixed, overlapping, and incorrectly ordered complex operator components. It also supports interference between operators, where the effect of one operator is partially or completely hidden from the target metamodel by other operators.

1 Introduction

Changing requirements and technological progress require metamodels to evolve [8]. Preventing metamodel evolution by downwards-compatible changes is often insufficient, as it reduces the quality of the metamodel [2]. Metamodel evolution may break conformance of existing models and thus requires model migration [22]. To correctly migrate models, the evolution – implicitly applied by developers – needs to become explicit. Metamodel evolution can be specified manually by developers, yet this is error-prone, redundant, and hard in larger projects. Instead, evolution needs to be detected automatically from the original and evolved metamodel versions.

The most-used solution for detecting evolution is matching [24]. Metamodel matching attempts to link elements from the original metamodel to elements from the target metamodel based on similarity. The result is a set of atomic differences highlighting what was created, what was deleted and what was changed. In practice, groups of atomic differences may be applied together to form complex evolution steps such as pulling features up an inheritance chain or extracting super classes [13]. In model migration, a complex operator is different from its atomic changes. For example, pulling up a feature preserves information, whereas deleting and recreating it loses information. To correctly describe evolution, we therefore need to detect complex evolution steps. There are three major problems in reconstructing complex evolution steps:

Dependency. While metamodel changes are unordered, evolution steps are generally applied sequentially and may depend on one another [4]. These dependencies need to be respected by a mapping from metamodel changes to evolution steps.

Detection. To detect a complex evolution step, we must find several steps which make up this complex step. But these steps are likely to be separated, incorrectly ordered, and mixed with parts of other complex evolution steps.

Interference. An evolution step can hide, change, or partially undo the effect of another step. Multiple steps can completely mask a step. As such, some or all steps forming a more complex step may be missing, which impedes its detection.

Example. The upper part of Figure 1 shows two metamodel versions for a tag-based issue tracker. In the original metamodel on the left-hand side, each issue has a reporter, a title, and some descriptive text. Projects are formed by a group of users and have a name and a set of issues. Users can comment on issues and tag issues. Additions and removals of tags are recorded, such that they can be reverted.

While evolving the issue tracker, tagging became the primary approach for organization. As such, it became apparent, that not only issues, but also projects should be taggable. Additionally, the metamodel structure had to be improved to allow users to more easily subscribe to events, as to send them email updates. The resulting metamodel is shown at the upper right of Figure 1. An Event entity was introduced, which comprises comments as well as tag events (tag additions and removals). Furthermore, projects obtained room for storing tags and events on these tags.

Matching the original and evolved metamodel yields the difference model presented in the middle part of Figure 1. Two classes and seven features were added to the evolved metamodel (left column), eight features were subtracted (middle column), and three classes have an additional super type in the evolved metamodel (right column). We will use this difference model as a starting point to detect the complex evolution steps involved in the evolution of the original metamodel.

The evolution of the metamodel can also be captured in an evolution trace as shown in the bottom part of Figure 1. At the metamodel level, the trace specifies the creation of five new features, the renaming of two other features, and the extraction of two new classes. At the model level, it specifies a corresponding migration. From the properties of the involved operators, we can conclude that the evolution is constructive and that we can safely migrate existing models without losing information.

In detecting the example evolution trace from the difference model, we face all three major problems in trace reconstruction several times. For example, the second step depends on the first step as it can only be applied if `TagRemoval` has a `timestamp`. Furthermore, the second step comprises several of the presented differences. And finally, the first step interferes with the second, since its effect is completely hidden from the difference model. The step needs to be reconstructed during detection.

Contribution. In this paper, we provide an approach to reconstruct complex evolution traces from difference models automatically. It is based on the formalization of the core concepts involved, namely metamodels, difference models, and evolution traces (Section 2). First, we provide a mapping from changes in a difference model to primitive operators in an evolution trace. We solve the dependency problem by defining preconditions for all primitive operators. Based on these preconditions, we define a dependency relation between operators which allows us to order operators on dependency and to

```
class Issue {
   title :: String
   description :: Text
   reporter -> User
   project -> Project opposite issues
   tags <> Tag (0..*)
}

class Project {
   name :: String
   issues -> Issue (1..*)
      opposite project
   members -> User (1..*)
}

class Tag {
   name :: String
}

class TagAddition {
   issue -> Issue
   tag -> Tag
   timestamp :: DateTime
}

class TagRemoval {
   issue -> Issue
   tag -> Tag
}

class Comment {
   issue -> Issue
   timestamp :: DateTime
   content :: Text
   author -> User
}

class User {...}
```

```
class Issue {
   title :: String
   description :: Text
   reporter -> User
   project -> Project opposite issues
   log <> Event (0..*) opposite issue
   tags <> Tag (0..*)
}

class Project {
   name :: String
   issues -> Issue (1..*)
      opposite project
   members -> User (1..*)
   log <> TagEvent (0..*)
   tags <> Tag (0..*)
}

class Tag {
   name :: String
}

class TagAddition : TagEvent {}
class TagRemoval : TagEvent {}

class Event {
   issue -> Issue opposite log
   time :: DateTime
   actor -> User
}

class TagEvent : Event {
   tag -> Tag
}

class Comment : Event {
   content :: Text
}

class User {...}
```

$\perp \rightarrow \langle\texttt{Issue.log}\rangle$
$\perp \rightarrow \langle\texttt{Project.log}\rangle$
$\perp \rightarrow \langle\texttt{Project.tags}\rangle$
$\perp \rightarrow \langle\texttt{Event}\rangle$
$\perp \rightarrow \langle\texttt{Event.issue}\rangle$
$\perp \rightarrow \langle\texttt{Event.time}\rangle$
$\perp \rightarrow \langle\texttt{Event.actor}\rangle$
$\perp \rightarrow \langle\texttt{TagEvent}\rangle$
$\perp \rightarrow \langle\texttt{TagEvent.tag}\rangle$

$\langle\texttt{TagAddition.tag}\rangle \rightarrow \perp$
$\langle\texttt{TagAddition.timestamp}\rangle \rightarrow \perp$
$\langle\texttt{TagAddition.issue}\rangle \rightarrow \perp$
$\langle\texttt{TagRemoval.issue}\rangle \rightarrow \perp$
$\langle\texttt{TagRemoval.tag}\rangle \rightarrow \perp$
$\langle\texttt{Comment.author}\rangle \rightarrow \perp$
$\langle\texttt{Comment.issue}\rangle \rightarrow \perp$
$\langle\texttt{Comment.timestamp}\rangle \rightarrow \perp$

$\langle\texttt{TagAddition}\rangle \xrightarrow[\langle\texttt{TagEvent}\rangle]{+superTypes} \langle\texttt{TagAddition}\rangle$

$\langle\texttt{TagRemoval}\rangle \xrightarrow[\langle\texttt{TagEvent}\rangle]{+superTypes} \langle\texttt{TagRemoval}\rangle$

$\langle\texttt{Comment}\rangle \xrightarrow[\langle\texttt{Event}\rangle]{+superTypes} \langle\texttt{Comment}\rangle$

```
create feature TagRemoval.timestamp :: DateTime
extract super class TagEvent {issue, timestamp, tag} from TagAddition, TagRemoval

rename Comment.author to actor
create feature TagEvent.actor -> User
extract super class Event {issue, timestamp, actor} from Comment, TagEvent
rename Event.timestamp to time

create feature Issue.log <> Event (0..*) opposite issue
create feature Project.log <> TagEvent (0..*)
create feature Project.tags <> Tag (0..*)
```

Fig. 1. Original and evolved metamodel, difference model, and evolution trace

construct valid primitive evolution traces from a difference model (Section 3). Second, we show how to reorder primitive traces without breaking their validity and provide patterns for mapping sequences of primitive operators to complex operators. We solve the detection problem by reordering primitive traces to different normal forms in which the patterns can be detected easily (Section 4). Finally, we extend our method to detect also partial patterns in order to solve the interference problem (Section 5).

2 Modeling Metamodel Evolution

Metamodeling Formalism. Metamodels can be expressed in various metamodeling formalisms. In this paper, we focus only on the core metamodeling constructs that are interesting for coupled evolution of metamodels and models. We leave out packages, enumerations, annotations, derived features, and operations.

Figure 2 gives a textual definition of the metamodeling formalism used in this paper. A metamodel defines a number of classes which consist of a number of features. Classes can have super types to inherit features and might be abstract. A feature has a multiplicity (lower and upper bound) and is either an attribute or a reference. An attribute is a feature with a primitive type, whereas a reference is a feature with a class type. We only support predefined primitive types like Boolean, Integer, and String. An attribute can serve as an identifier for objects of a class. A reference may be composite and two references can be combined to form a bidirectional association by making them opposite of each other. In the textual notation, features are represented by their name followed by a separator, their type, and an optional multiplicity. The separator indicates the kind of a feature. We use `::` for attributes, `->` for ordinary references, and `<>` for composite references.

If we want to reason about properties of metamodels and their evolution, a textual representation is often not sufficient. Thus, we provide in Figure 3 a more formal representation of metamodels in terms of sets, functions, and predicates. In the upper left, we define instance sets for the metaclasses from Figure 2. In the upper right, we formalize most metafeatures from Figure 2 in terms of functions and predicates. Since super types and features of a class c form subsets of instance sets, we formalize them accordingly.

```
class MetaModel {
  classes     <> Class (0..*)
}

abstract class NamedElement {
  name        :: String (1..1)
}

abstract class Type : NamedElement {}

class Class : Type {
  isAbstract  :: Boolean (1..1)
  superTypes  -> Class (0..*)
  features    <> Feature (0..*)
}
```

```
class DataType : Type {}

abstract class Feature : NamedElement {
  lowerBound  :: Integer (1..1)
  upperBound  :: Integer (1..1)
  type        -> Type (1..1)
}

class Attribute : Feature {
  isId        :: Boolean (1..1)
}

class Reference : Feature {
  isComposite :: Boolean (1..1)
  opposite    -> Reference
}
```

Fig. 2. Metamodeling formalism providing core metamodeling concepts

Instance sets

$N := T \cup F$	(*named elements*)
$T := T_d \cup T_c$	(*types*)
T_d	(*data types*)
T_c	(*classes*)
$F := F_a \cup F_r$	(*features*)
F_a	(*attributes*)
F_r	(*references*)

Functions and predicates

$name : N \rightarrow String$	(*names*)
$lower : F \rightarrow Integer$	(*lower bounds*)
$upper : F \rightarrow Integer$	(*upper bounds*)
$type : F \rightarrow T$	(*types*)
$opposite : F_r \rightarrow F_r$	(*opposite references*)
$abstract : T_c$	(*abstract classes*)
$id : F_a$	(*identifying attributes*)
$composite : F_r$	(*composite references*)

Instance subsets

$C_p(c)$	(*parents*)
$C_c(c) := \{\, c' \in T_c \mid c \in C_p(c') \}$	(*children*)
$C_a(c) := C_p(c) \cup \bigcup_{c' \in C_p(c)} C_a(c')$	(*ancestors*)
$C_d(c) := C_c(c) \cup \bigcup_{c' \in C_c(c)} C_d(c')$	(*descendants*)
$C_h(c) := C_a(c) \cup C_d(c) \cup \{c\}$	(*type hierarchy*)
$F(c)$	(*defined features*)
$F_i(c) := F(c) \cup \bigcup_{c' \in C_a(c)} F(c')$	(*defined and inherited features*)
$F_a(c) := F_a \cap F(c)$	(*attributes*)
$F_r(c) := F_r \cap F(c)$	(*references*)

Lookup functions

$$\langle cn \rangle := \begin{cases} c & \text{if } c \in T_c \wedge name(c) = cn \\ \bot & \text{else} \end{cases} \qquad \langle cn.fn \rangle := \begin{cases} f & \text{if } f \in F(\langle cn \rangle) \wedge name(f) = fn \\ \bot & \text{else} \end{cases}$$

Fig. 3. Formal representation of metamodels in terms of sets, functions, and predicates

In terms of these subsets, we define other interesting subsets, e.g., children, ancestors and descendants of c in the middle part. Typically, we refer to a class c by its name cn and to a feature f of class c by $cn.fn$ where cn and fn are the names of c and f, respectively. To access classes and features referred by name, we define lookup functions in the last box. The formalization so far also captures invalid metamodels, such as metamodels with duplicate class names, or cycles in an inheritance hierarchy. Therefore, we define metamodel validity by a number of invariants in Figure 4.

Difference Models. Difference-based approaches to coupled evolution use a declarative evolution specification, generally referred to as the difference model [3,9]. This difference model can be mapped automatically onto a model migration. With an automated detection of the difference model, the process can be completely automated. Matching algorithms provide such a detection [17,7,5,15,30,1].

In this paper, we do not rely on a particular matching algorithm and abstract over concrete representations of difference models. We model the difference between an original metamodel m_o and an evolved version m_e as a set $\Delta(m_o, m_e)$. The elements

Metamodel validity $\vdash m$

$\forall c, c' \in T_c : name(c) = name(c') \Rightarrow c = c'$	(*unique class names*)
$\forall c \in T_c : \forall f, f' \in F_i(c) : name(f) = name(f') \Rightarrow f = f'$	(*unique feature names*)
$\forall c \in T_c : c \notin C_a(c)$	(*non-cyclic inheritance*)
$\forall f \in F : lower(f) \leq_b upper(f) \wedge upper(f) >_b 0$	(*correct bounds*)
$\forall f \in F_a : type(f) \in T_d$	(*well-typed attributes*)
$\forall f \in F_r : type(f) \in T_c$	(*well-typed references*)
$\forall f, f' \in F_r : opposite(f) = f' \Leftrightarrow opposite(f') = f$	(*inverse reflectivity*)

Difference model validity $\vdash \Delta(m_o, m_e)$

$\vdash m_o \wedge \vdash m_e$	(*source and target validity*)
$\forall \delta, \delta' \in \Delta(m_o, m_e) : t(\delta) = t(\delta') \neq \perp \Rightarrow s(\delta) = s(\delta')$	(*unique sources*)
$\forall \delta, \delta' \in \Delta(m_o, m_e) : s(\delta) = s(\delta') \neq \perp \Rightarrow t(\delta) = t(\delta')$	(*unique targets*)
$\forall \delta, \delta' \in \Delta(m_o, m_e) : s(\delta) \in F(s(\delta')) \wedge t(\delta) \neq \perp \Rightarrow t(\delta) \in F(t(\delta'))$	(*non-moving features*)

Evolution trace validity $m_o, m_e \vdash O_1 \dots O_n$

$\vdash m_o$	(*source validity*)
$\forall i \in 1, \dots, n : \vdash O_1 \circ \dots \circ O_i(m_o)$	(*valid applications*)
$O_1 \circ \dots \circ O_n(m_o) = m_e$	(*target validity*)

Fig. 4. Validity of metamodels, difference models, and evolution traces

of this set are three different kinds of changes [26,3]: *Additive changes* $\perp\rightarrow e$, where the evolved metamodel contains an element e which was not present in the original metamodel. *Subtractive changes* $e \rightarrow\perp$, where the evolved metamodel misses an element e which was present in the original metamodel. *Updative changes*, where the evolved metamodel contains an element e' which corresponds to an element e in the original metamodel and the value of a metafeature of e' is different from the value in e. We distinguish three kinds of updates: *Additions* $e \xrightarrow[v]{+mf} e'$, where the multi-valued metafeature mf of e' has an additional value v which was not present in e. *Removals* $e \xrightarrow[v]{-mf} e'$, where the multi-valued metafeature mf of e' is missing a value v which was present in e. *Substitutions* $e \xrightarrow{mf} e'$, where the single-valued metafeature mf of e' has a new value which is different from the value in e. A complete list of possible metamodel changes with respect to our metamodeling formalism is given in the left column of Figure 5.

For validity of difference models, we have three requirements: First, the original and evolved metamodel need to be valid. Second, two changes should not link the same source element with different target elements or the same target element with different source elements. Element merges and splits are represented as separate additions and removals and will be reconstructed during detection. Third, we expect changing features not to move between classes, i.e., the class containing a changed feature should be the same or a changed version of the class containing the original feature. We define these requirements formally in Figure 4. Note that $s(\delta)$ yields the source element of a change (left-hand side of an arrow) while $t(\delta)$ gives the target element (right-hand side).

Evolution Traces. Operator-based approaches to coupled evolution provide a rich set of coupled operators which work at the metamodel level as well as at the model level [29,11]. At the metamodel level, a *coupled operator* defines a metamodel transformation capturing a common evolution step. At the model level, it defines a model transformation capturing the corresponding migration. Following the terminology from [13], we differentiate between primitive and complex operators. *Primitive operators* perform an atomic metamodel evolution step that can not be further subdivided. A list of primitive operators which is complete with respect to our metamodeling formalism is given in the left column of Figure 7. *Complex operators* can be decomposed into a sequence of primitive operators which has the same effect at the metamodel level but typically not at the model level. For example, a feature pull-up can be decomposed into feature deletions in the subclasses followed by a feature creation in the parent class. At the model level, the feature deletions cause the deletion of values in instances of the subclasses while the feature creation requires the introduction of default values in instances of the parent class. Thus, values for the feature in instances of the subclasses are replaced by default values. This is not an appropriate migration for a feature pull-up which instead requires the preservation of values in instances of the subclasses. We will define only a few complex operators in this paper. For an extensive catalog of operators, see [13].

Each operator has a number of formal parameters like class and feature names. Instantiating these parameters with actual arguments results in an *operator instance* O. This notation hides the actual arguments but is sufficient for this paper. We can now model the evolution of a metamodel as a sequence of such operator instances $O_1 \dots O_n$. We call this sequence an *evolution trace*. We distinguish *primitive traces* of only primitive operator instances from *complex traces*. There are three requirements for the validity of an evolution trace with respect to the original and the evolved metamodel. First, we require the original metamodel to be valid. Second, each operator instance should be applicable to the result of its predecessors and should yield a valid metamodel. Third, applying the complete trace should result in the evolved metamodel. Again, we capture these requirements formally in Figure 4.

3 Reconstructing Primitive Evolution Traces

This section shows how to reconstruct a correctly ordered, valid evolution trace from a difference model. First, we provide a mapping from metamodel changes to sequences of primitive operator instances. Second, we define a dependency relation between operator instances based on preconditions of these instances. This allows us to order primitive evolution traces on dependency resulting in valid primitive evolution traces.

Mapping. The mapping of changes onto sequences of operator instances is presented in Figure 5. The left column shows the different metamodel changes. The right column shows the corresponding operator instances. The middle column shows conditions to select the right mapping and to instantiate parameters correctly. Note that we omit conditions of the form $xn = name(x)$. We assume such conditions implicitly whenever there is a pair of variables x and xn. This way, cn refers to the name of a class c, fn to the name of a feature f, and tn to the name of a type t. Figure 6 (left) shows the result of the mapping applied to the example difference model from Figure 1.

Dependencies between Operator Instances. Despite the atomicity of primitive operators, not all primitive evolution traces can be completely executed. Reconsider the left trace in Figure 6. Step 5 creates a reference to `TagEvent` at a point where no class `TagEvent` exists. Similarly, step 8 references a non-existent class `Tag` and step 24 attempts to create an inheritance chain with duplicate feature names. Operator instances cannot be applied to all metamodels: Features can only be created in classes that exist, classes can only be created if no equivalently named class is present and a class can only be dropped if it is not in use anywhere else. These restrictions either come directly from the meta-metamodel or from the invariants for valid metamodels. We can translate these restrictions into preconditions. An operator precondition $O_{pre}(m)$ ensures that an operator instance O can be applied to a metamodel m and that the application on a valid m yields again a valid metamodel. Figures 7 and 8 give a complete overview of the preconditions for primitive operators.

One condition for the validity of a trace of operators is the validity of each intermediate metamodel. Since succeeding operator preconditions ensure this validity, we can redefine trace validity in terms of preconditions:

Evolution trace validity $m_o, m_e \vdash O_1 \dots O_n$

$$O_{1,pre}(m_o) \wedge \forall_{i \in 2..n} : O_{i,pre}((O_1 \circ \dots \circ O_{i-1})(m)) \qquad (\textit{valid applications})$$

Applying operator instances enables or disables other operator instances. For example, the creation of a class c can enable the creation of a feature $c.f$. The class creation operator validates parts of the precondition of the feature creation operator. To model the effect of an operator instance on conditions, we use a backward transformation description as introduced by Kniesel and Koch [14]. A backward description O_{bd} is a function that, given a condition C to be checked after applying an operator instance O, computes a semantically equivalent condition that can be checked before applying O: $O_{bd}(C)(m) \Leftrightarrow C(O(m))$. We define backward description functions for the primitive operators based on the postconditions specified in Figures 7 and 8: A backward description rewrites any clause in a condition C with $true$, when it is implied by the operator postcondition. Using these backward description functions, we can define enabling and disabling operator instances as dependencies: Operator instance O_2 depends on operator instance O_1, if the backward description of operator O_1 changes the precondition of O_2. Typically, operator instances are dependent if they affect or target the same metamodel element. Examples are creation and deletion of the same class, creation of a class and addition of a feature to this class, and creation of a class and of a reference to this class.

Dependency Ordering. To ensure trace validity, we need to ensure that the preconditions of all operator instances are enabled and thus all dependencies are satisfied. The dependency relation between operator instances is a partial order on these instances. To establish validity, we apply the partial dependency order to the trace and make the ordering complete by arbitrarily ordering independent operator instances. Figure 6 (right) shows the dependency-ordered trace of primitive operators for the running example.

Metamodel Difference	Conditions	Primitive Operator Instances
$\perp \rightarrow c$	$c \in T_c$ $abstract(c)$ $C_p(c) = \{sc_1, \ldots, sc_k\}$	**create class** cn [**make** cn **abstract**] [**add super** scn_1 **to** cn $\vdots$ **add super** scn_k **to** cn]
$c \rightarrow \perp$	$c \in T_c$	**drop class** cn
$e \xrightarrow{name} e'$	$e \in T_c$	**rename** en **to** en'
	$e \in F(c)$	**rename** $cn.en$ **to** en'
$c \xrightarrow{isAbstract} c'$	$\neg abstract(c)$	**make** cn **abstract**
	$abstract(c)$	**drop** cn **abstract**
$c \xrightarrow[sc]{+superTypes} c'$		**add super** scn **to** cn
$c \xrightarrow[sc]{-superTypes} c'$		**drop super** scn **from** cn
$\perp \rightarrow f$	$f \in F_a(c) \wedge t = type(f)$ $l = lower(f) \wedge l >_b 0$ $u = upper(f) \wedge u >_b 1$ $id(f)$	**create feature** $cn.fn$ `::` tn [**specialize lower** $cn.fn$ **to** l] [**generalize upper** $cn.fn$ **to** u] [**make** $cn.fn$ **identifier**]
	$f \in F_r(c) \wedge t = type(f)$ $l = lower(f) \wedge l >_b 0$ $u = upper(f) \wedge u >_b 1$ $composite(f)$ $f' = opposite(f)$	**create feature** $cn.fn$ `->` tn [**specialize lower** $cn.fn$ **to** l] [**generalize upper** $cn.fn$ **to** u] [**make** $cn.fn$ **composite**] [**make** $cn.fn$ **inverse** fn']
$f \rightarrow \perp$	$f \in F(c)$	**drop feature** $cn.fn$
$f \xrightarrow{lowerBound} f'$	$l = lower(f') \wedge l <_b lower(f)$	**generalize lower** $cn.fn$ **to** l
	$l = lower(f') \wedge l >_b lower(f)$	**specialize lower** $cn.fn$ **to** l
$f \xrightarrow{upperBound} f'$	$u = upper(f') \wedge u >_b upper(f)$	**generalize upper** $cn.fn$ **to** u
	$u = upper(f') \wedge u <_b upper(f)$	**specialize upper** $cn.fn$ **to** u
$f \xrightarrow{type} f'$	$f \in F(c)$ $f' \in F_a(c') \wedge t = type(f')$ $l = lower(f') \wedge l >_b 0$ $u = upper(f') \wedge u >_b 1$ $id(f')$	**drop feature** $cn.fn$ **create feature** $cn'.fn'$ `::` tn [**specialize lower** $cn'.fn'$ **to** l] [**generalize upper** $cn'.fn'$ **to** u] [**make** $cn'.fn'$ **identifier**]
	$f \in F(c)$ $f' \in F_r(c') \wedge t = type(f')$ $l = lower(f') \wedge l >_b 0$ $u = upper(f') \wedge u >_b 1$ $composite(f')$ $f'' = opposite(f')$	**drop feature** $cn.fn$ **create feature** $cn'.fn'$ `->` tn [**specialize lower** $cn'.fn'$ **to** l] [**generalize upper** $cn'.fn'$ **to** u] [**make** $cn'.fn'$ **composite**] [**make** $cn'.fn'$ **inverse** fn'']
$f \xrightarrow{isId} f'$	$\neg id(f)$	**make** $cn.fn$ **identifier**
	$id(f)$	**drop** $cn.fn$ **identifier**
$f \xrightarrow{isComposite} f'$	$\neg composite(f)$	**make** $cn.fn$ **composite**
	$composite(f)$	**drop** $cn.fn$ **composite**
$f \xrightarrow{opposite} f'$	$f' \in F_r(c) \wedge f'' = opposite(f') \neq \perp$	**make** $cn.fn'$ **inverse** fn''
	$f' \in F_r(c) \wedge opposite(f') = \perp$	**drop** $cn.fn'$ **inverse**

Fig. 5. Possible metamodel changes and corresponding sequences of primitive operator instances

```
create feature Issue.log <> Event
generalize upper Issue.log to -1
make Issue.log composite
make Issue.log inverse Event.issue
create feature Project.log
   <> TagEvent
generalize upper Project.log to -1
make Project.log composite
create feature Project.tags <> Tag
generalize upper Project.tags to -1
make Project.tags composite
add super TagEvent to TagAddition
drop feature TagAddition.issue
drop feature TagAddition.tag
drop feature TagAddition.timestamp
add super TagEvent to TagRemoval
drop feature TagRemoval.issue
drop feature TagRemoval.tag
create class Event
create feature Event.issue -> Issue
create feature Event.time
   :: DateTime
create feature Event.actor -> User
create class TagEvent : Event
create feature TagEvent.tag -> Tag
add super Event to Comment
drop feature Comment.issue
drop feature Comment.timestamp
drop feature Comment.author
```

```
create feature Project.tags <> Tag
generalize upper Project.tags to -1
make Project.tags composite
drop feature TagAddition.issue
drop feature TagAddition.tag
drop feature TagAddition.timestamp
drop feature TagRemoval.issue
drop feature TagRemoval.tag
create class Event
create feature Issue.log <> Event
generalize upper Issue.log to -1
make Issue.log composite
create feature Event.issue -> Issue
make Issue.log inverse Event.issue
create feature Event.time
   :: DateTime
create feature Event.actor -> User
create class TagEvent : Event
create feature Project.log
   <> TagEvent
generalize upper Project.log to -1
make Project.log composite
add super TagEvent to TagAddition
add super TagEvent to TagRemoval
create feature TagEvent.tag -> Tag
drop feature Comment.issue
drop feature Comment.timestamp
drop feature Comment.author
add super Event to Comment
```

Fig. 6. Unordered and dependency-ordered primitives mapped from the difference model

Primitive Operator	Preconditions	Postconditions
create class *cn*	$\langle cn\rangle = \bot$	$\langle cn\rangle \neq \bot \wedge F(\langle cn\rangle) = \emptyset$ $\neg targeted(\langle cn\rangle) \wedge \neg abstract(\langle cn\rangle)$
drop class *cn*	$\langle cn\rangle \neq \bot \wedge F(\langle cn\rangle) = \emptyset$ $\neg targeted(\langle cn\rangle)$	$\langle cn\rangle \neq \bot$
create feature *cn.fn* :: *tn*	$\langle cn\rangle \neq \bot$ $\forall c' \in C_h(\langle cn\rangle) : \forall f' \in F(c') :$ $name(f') \neq fn$	$\langle cn.fn\rangle \neq \bot$ $\langle cn.fn\rangle \in F_a$
create feature *cn.fn* -> *tn*	$\langle cn\rangle, \langle tn\rangle \neq \bot$ $\forall c' \in C_h(\langle cn\rangle) : \forall f' \in F(c') :$ $name(f') \neq fn$	$\langle cn.fn\rangle \neq \bot$ $\langle cn.fn\rangle \in F_r \wedge type(\langle cn.fn\rangle) = \langle tn\rangle$ $\nexists f' : opposite(\langle cn.fn\rangle) = f')$ $\neg composite(\langle cn.fn\rangle) \wedge \neg id(\langle cn.fn\rangle)$
drop feature *cn.fn*	$\langle cn.fn\rangle \neq \bot$	$\langle cn.fn\rangle = \bot$

Fig. 7. Pre- and postconditions for structural primitive operators

Primitive Operator	Preconditions	Postconditions
rename class cn **to** cn'	$\langle cn \rangle \neq \perp \wedge \langle cn' \rangle = \perp$	$\langle cn \rangle = \perp \wedge \langle cn' \rangle \neq \perp$
rename feature $cn.fn$ **to** fn'	$\langle cn.fn \rangle \neq \perp$ $\forall c' \in C_h(\langle cn \rangle) : \forall f' \in F(c') :$ $name(f') \neq fn'$	$\langle cn.fn \rangle = \perp$ $\langle cn.fn' \rangle \neq \perp$
make cn **abstract**	$\langle cn \rangle \neq \perp \wedge \neg abstract(\langle cn \rangle)$	$abstract(\langle cn \rangle)$
drop cn **abstract**	$\langle cn \rangle \neq \perp \wedge abstract(\langle cn \rangle)$	$\neg abstract(\langle cn \rangle)$
add super cn_{sup} **to** cn_{sub}	$\langle cn_{sup} \rangle, \langle cn_{sub} \rangle \neq \perp$ $\langle cn_{sup} \rangle \notin C_h(\langle cn_{sub} \rangle)$ $\forall c \in C_h(\langle cn_{sub} \rangle) : \forall f \in F(c) :$ $\langle cn_{sup}.name(f) \rangle = \perp$	$\langle cn_{sup} \rangle \in C_p(\langle cn_{sub} \rangle)$
drop super cn_{sup} **from** cn_{sub}	$\langle cn_{sub} \rangle, \langle cn_{sup} \rangle \neq \perp$ $\langle cn_{sup} \rangle \in C_p(\langle cn_{sub} \rangle)$	$\langle cn_{sup} \rangle \notin C_p(\langle cn_{sub} \rangle)$
generalize type $cn.fn$ **to** cn'	$\langle cn.fn \rangle \neq \perp \wedge \langle cn' \rangle \neq \perp$ $\langle cn' \rangle \in C_a(\langle cn \rangle)$	$type(\langle cn.fn \rangle) = \langle cn' \rangle.$
specialize type $cn.fn$ **to** cn'	$\langle cn.fn \rangle \neq \perp \wedge \langle cn' \rangle \neq \perp$ $\langle cn' \rangle \in C_d(\langle cn \rangle)$	$type(\langle cn.fn \rangle) = \langle cn' \rangle.$
generalize upper $cn.fn$ **to** u	$\langle cn.fn \rangle \neq \perp$ $u >_B upper(\langle cn.fn \rangle)$	$upper(\langle cn.fn \rangle) = u$
generalize lower $cn.fn$ **to** l	$\langle cn.fn \rangle \neq \perp$ $l < lower(\langle cn.fn \rangle)$	$lower(\langle cn.fn \rangle) = l$
specialize upper $cn.fn$ **to** u	$\langle cn.fn \rangle \neq \perp$ $u <_B upper(\langle cn.fn \rangle)$ $u \geq_B lower(\langle cn.fn \rangle)$	$upper(\langle cn.fn \rangle) = u$
specialize lower $cn.fn$ **to** l	$\langle cn.fn \rangle \neq \perp$ $l > lower(\langle cn.fn \rangle)$ $l \leq upper(\langle cn.fn \rangle)$	$lower(\langle cn.fn \rangle) = l$
make $cn.fn$ **inverse** $cn'.fn'$	$\langle cn.fn \rangle, \langle cn'.fn' \rangle \neq \perp$ $\nexists f : opposite(\langle cn.fn \rangle) = f$ $\vee opposite(\langle cn'.fn' \rangle) = f$	$opposite(\langle cn.fn \rangle) = \langle cn'.fn' \rangle$
drop $cn.fn$ **inverse**	$\langle cn.fn \rangle \neq \perp$ $\exists f' : opposite(\langle cn.fn \rangle) = f'$	$\nexists f' : opposite(\langle cn.fn \rangle) = f'$
make $cn.fn$ **identifier**	$\langle cn.fn \rangle \neq \perp \wedge \neg id(\langle cn.fn \rangle)$	$id(\langle cn.fn \rangle)$
drop $cn.fn$ **identifier**	$\langle cn.fn \rangle \neq \perp \wedge id(\langle cn.fn \rangle)$	$\neg id(\langle cn.fn \rangle)$
make $cn.fn$ **composite**	$\langle cn.fn \rangle \neq \perp$ $\neg composite(\langle cn.fn \rangle)$	$composite(\langle cn.fn \rangle)$
drop $cn.fn$ **composite**	$\langle cn.fn \rangle \neq \perp$ $composite(\langle cn.fn \rangle)$	$\neg composite(\langle cn.fn \rangle)$

Fig. 8. Pre- and postconditions for non-structural primitive operators

Complex Operator	Conditions	Equivalent Trace
`pull up feature` $cn.fn$	$\boxed{\begin{array}{l} C_c(\langle cn\rangle) = \{c_1, \ldots, c_k\} \\ \langle cn_1.fn\rangle \equiv_F \cdots \equiv_F \langle cn_k.fn\rangle \end{array}}$ $t = type(\langle cn_1.fn\rangle) \wedge t \in T_d$ $l = lower(\langle cn_1.fn\rangle) \wedge l >_b 0$ $u = upper(\langle cn_1.fn\rangle) \wedge u >_b 1$ $id(\langle cn_1.fn\rangle)$	**`drop feature`** $cn_1.fn$ $\ldots$ **`drop feature`** $cn_k.fn$ **`create feature`** $cn.fn$ `::` tn [**`specialize lower`** $cn.fn$ **`to`** l] [**`generalize upper`** $cn.fn$ **`to`** u] [**`make`** $cn.fn$ **`identifier`**]
	$\boxed{\begin{array}{l} C_c(\langle cn\rangle) = \{c_1, \ldots, c_k\} \\ \langle cn_1.fn\rangle \equiv_F \cdots \equiv_F \langle cn_k.fn\rangle \end{array}}$ $t = type(\langle cn_1.fn\rangle) \wedge t \in T_c$ $l = lower(\langle cn_1.fn\rangle) \wedge l >_b 0$ $u = upper(\langle cn_1.fn\rangle) \wedge u >_b 1$ $composite(\langle cn_1.fn\rangle)$ $f' = opposite(\langle cn_1.fn\rangle)$	**`drop feature`** $cn_1.fn$ $\ldots$ **`drop feature`** $cn_k.fn$ **`create feature`** $cn.fn$ `->` tn [**`specialize lower`** $cn.fn$ **`to`** l] [**`generalize upper`** $cn.fn$ **`to`** u] [**`make`** $cn.fn$ **`composite`**] [**`make`** $cn.fn$ **`inverse`** fn']
`extract super class` cn $\{fn_1, \ldots, fn_j\}$ **`from`** $cn_1, \ldots, cn_k$	$\boxed{true}$	**`create class`** cn **`add super`** cn **`to`** cn_1 $\ldots$ **`add super`** cn **`to`** cn_k **`pull up feature`** $cn.fn_1$ $\ldots$ **`pull up feature`** $cn.fn_j$
`fold super class` cn **`from`** cn'	$\boxed{\begin{array}{l} F_i(\langle cn\rangle) = \{f_1, \ldots, f_k\} \\ \forall i = 1 \ldots k : \langle cn'.fn_i\rangle \equiv_F f_i \end{array}}$	**`drop feature`** $cn'.fn_1$ $\ldots$ **`drop feature`** $cn'.fn_k$ **`add super`** cn **`to`** cn'

Fig. 9. (De-)Composition patterns for complex operators

4 Reconstructing Complex Evolution Traces

This section shows how to reconstruct valid complex evolution traces from valid primitive traces. First, we provide patterns for mapping sequences of primitive operator instances to complex operator instances. Second, we discuss how to reorder evolution traces without breaking their validity. This allows us to reorder traces into different normal forms in which the patterns can be detected easily and be replaced by complex operator instances.

Patterns. A complex operator instance comprises a sequence of (less-complex) operator instances. We can use patterns on these sequences to detect complex operator instances. Figure 9 lists the decompositions and conditions for two complex operators working across inheritance. When read from left to right, it shows how to decompose a complex operator instance, when read from right to left, it defines its detection pattern. Given a source metamodel m, we can recursively decompose an operator instance O into a sequence of primitive operator instances $\lfloor O \rfloor_m = P_1 \ldots P_n$. As a precondition, a complex operator instance needs to fulfill the backward descriptions of the preconditions of these primitives. But typically this is not enough and an operator instance requires additional preconditions.We highlight these additional preconditions with a box in Figure 9.

```
1 create class Event
2 create feature Event.timestamp
                      :: DateTime
3 create feature Event.actor -> User
4 drop feature Comment.timestamp
5 drop feature Comment.actor
6 add super Event to Comment
```

```
1 create class Event
6 add super Event to Comment
5 drop feature Comment.actor
3 create feature Event.actor -> User
4 drop feature Comment.timestamp
2 create feature Event.timestamp
                      :: DateTime
```

Fig. 10. Excerpt of dependency-ordered operators in Figure 6

Reordering traces. Figure 10 shows an excerpt of Figure 6. It displays the extraction of super class `Event` from class `Comment`. Operator ordering is still determined by the dependency ordering from the previous section. To simplify the example, we changed the operator on `Comment.author` to work on `Comment.actor`. We will look at `author` and the complete trace in the next section. Consider applying the patterns from Figure 9. There is no consecutive sequence of operator instances satisfying any of the patterns. We could detect pulling up feature `timestamp` in instances 2 and 4, yet where do we put the detected complex operator: at position 2 or at position 4?

Detection patterns typically cannot be applied directly. Instead, traces need to be reordered to find consecutive instances of a pattern. Dependency ordering is partial and therefore leaves room for swapping independent operators. In the example, we can swap 2 and 3 as they work on different features; 2 and 4 as they work on different types; 2 and 5 which also work on different types; 4 and 5 which work on different features; 3 and 5, which work on different types; and finally, we can repeatedly swap 6 to follow operator 1, as all features that are created are dropped from the inheritance chain first. The reordered trace is shown at the right of Figure 10. We can now apply the patterns for pulling up `timestamp` and `actor`. Subsequently, we see the pattern for class extraction emerge, which yields a super class extraction of `Event {timestamp, actor}` from `Comment` and `TagEvent`.

Normal forms. In the example, we carefully swapped operators. Not only did we avoid swapping dependent operators (as to preserve trace validity), we also chose swaps, which gave us a detectable pattern. In particular, we focused on obtaining a consecutive feature creation and drop, of features that only differ in position in the inheritance chain. A set of swap rules can bring an evolution trace into a format most suitable for detecting a pattern. In general, these rules obey the dependency relation. However, some dependent instances can still be swapped by adjusting their parameters. For example, **`rename class`** `A` **`to`** `B` and **`create feature`** `B.f...` can be swapped to: **`create feature`** `A.f...` and **`rename class`** `A` **`to`** `B`.

Repeated application of a set of swap rules will result in a normal form defined by this set. Each normal form targets to bring potential components of a pattern together and to satisfy the operator precondition. For example, to detect a feature pull up, we rely on feature similarity: Class creations and super additions get precedence over other operators. Feature creations, changes, and drops are sorted on feature name, type, and modifiers. Class drops and destructive updates on the inheritance chain go last. Different patterns need different trace characteristics and thus different normal forms. But operators with similar kinds of patterns can share normal forms.

5 Reconstructing Masked Operator Instances

In this section, we extend the detection to deal not only with complete but also partial patterns. First, we revisit the problem of operator interference and study its effects on detection. Second, we show how to complete partial patterns by the additions of operator instances in a validity preserving fashion. This allows us to detect operator instances which patterns are partially or even completely hidden by other instances.

Masked Operators. We reconsider the running example from Figure 1. During evolution, several features of the classes `TagAddition` and `TagRemoval` were extracted into a new super class `TagEvent`. In order to extract the feature `timestamp` it needs to be present in both `TagAddition` and `TagRemoval`. Yet, it is not. As a human, we deduce that `timestamp` must have been added in the process of extracting `TagEvent`. There is, however, no explicit record of such feature creation. Detection will therefore fail. Later in the evolution, when extracting the class `Event`, we seek to pull up a feature `actor`. The class `Comment`, which we are extracting from, only offers a feature `author`. Again as a human, we assume that `author` must have been renamed to `actor` (like we did in the previous section), yet this operation is not present in the original evolution trace. Similarly, we have to create the feature `actor` in `TagEvent` before extracting `Event` and rename the feature `timestamp` to `time` after extracting `Event` to yield the target metamodel. Each of these operations has no record in the difference set obtained from the matching algorithm.

When evolutions become more complex, individual evolution steps no longer need to have an explicit effect on the target metamodel and are therefore not explicit in the matching result. An operator instance can hide or even undo parts of the effect of another instance. This is a strong variant of dependency, which we call masking. A primitive operator P_1 masks another primitive operator P_2 when composition of the two can be captured in a third primitive operator P_3. More generally, we define masking for arbitrary operator instances as the presence of a mask in decompositions:

$$P_1\ masks_m\ P_2 \Leftrightarrow \exists P_3 : (P_1 \circ P_2)(m) = P_3(m)$$
$$O_1\ masks_m\ O_2 \Leftrightarrow \exists P_1 \in \lfloor O_1 \rfloor_m : \exists P_2 \in \lfloor O_2 \rfloor_m : P_1\ masks\ P_2$$

Most operators can be masked by renaming. All operators are masked by their inverses, in which case P_3 is the identity operator. Extraction of class `TagEvent` in the running example masks extraction of class `Event`. Note that a trace obtained from a valid difference model will only contain masks that involve complex operators.

Masked Detection Rules. Detection of masked operator instances follows a trace rewriting approach similar to the original detection of complex operator instances: We try to rewrite a sequence of operator instances into another sequence which has the same effect on the metamodel. Instead of checking the operator precondition in a pattern, like we did in the previous section, we now *ensure* the precondition by deducing a suitable sequence to rewrite to. We now discuss how to derive a detection rule for a masked complex operator instance, e.g., for pulling up an attribute $cn_{sup}.fn$. Its decomposition is the following:

```
drop feature $cn_{sub}1.fn$
...                                 [specialize lower $cn_{sup}.fn$ to l]
drop feature $cn_{sub}i.fn$         [generalize upper $cn_{sup}.fn$ to u]
create feature $cn_{sup}.fn$        [make $cn.fn$ identifier]
```

From the decomposition we choose a trigger, which tells us that there may have been a feature pull up. We choose one of the feature drops (number x). We use the trigger as a pattern on the left-hand side of a rewrite rule and assume on the right-hand side that there must have been a feature pull up:

```
drop feature $cn_{sub}x.fn$     ->      ... pull up feature $cn_{sup}.fn$ ...
```

When the dots are left blank, application of the left-hand side to a metamodel does not have an equivalent effect as application of the right-hand side. Instead, we fill the dots, to establish equivalence. The left set of dots ensures that the pull up feature operator can be applied, i.e., its precondition is satisfied. The right set of dots ensures that application of the trace is equivalent to application of the left-hand side of the rewrite rule. Both sets of dots are filled in using inverses of the operators found in the pattern. The left set of dots is replaced by inverses of each of the primitive operators whose precondition is not already satisfied. For pull up feature, we create features in all sibling classes if they do not exist yet and remove the target feature if it already exists. The right set of dots is replaced by inverses that neutralize the effect of the complex operator and bring the metamodel back to its original state. For pull up feature, we need to create all sibling features, which were present beforehand, as these were deleted during pull up and we need to drop the target feature if it was not present beforehand. The rewrite rule for detecting a masked feature pull up is (leaving out the operations on feature modifiers, for simplicity):

```
                                        create feature $cn_{sib}n1.fn$
                                        ...
                                        create feature $cn_{sib}nj.fn$
                                        [drop feature $cn_{sup}.fn$]
drop feature $cn_{sub}x.fn$     ->      pull up feature $cn_{sup}.fn$
                                        create feature $cn_{sib}e1.fn$
                                        ...
                                        create feature $cn_{sib}ek.fn$
                                        [drop feature $cn_{sup}.fn$]
```

In which cn_{sup} is chosen arbitrarily from $C_p(\langle cn_{sub}x\rangle)$, $cn_{sib}n$ is the set of all sibling classes which do not have a feature named *fn* and thus need to obtain the feature to pull it up. $cn_{sib}e$ is the set of all sibling classes which do have a feature named *fn* and thus need to be reequipped with *fn* to neutralize the effect of pulling it up. The feature drops are conditional. The first drop should be present if $\langle cn_{sup}.fn\rangle \neq \perp$ and the latter should be present if $\langle cn_{sup}.fn\rangle = \perp$. In addition to the pattern on the left-hand side of a rewrite rule for a masked complex operator O, a rewrite rule is also conditioned by the operator's precondition O_{cpre}. It is checked in addition to the trigger. For feature pull up, the operator precondition O_{cpre} ensures presence of an inheritance chain between cn_{sub} and cn_{sup}. The metamodel invariants ensure feature names uniqueness across inheritance. The precondition of the trigger ensures *fn* exists in cn_{sub}. Therefore, *fn* cannot exist in cn_{sup}. The rewrite rule for feature pull up can thus be simplified by removing the top drop feature and always using the bottom drop feature.

Using the presented approach, we can derive masked detection rules for any complex operator. By definition, such rules expand the trace. To find a suitable evolution, we

```
1  drop feature TagAddition.issue      ->  pull up feature TagEvent.issue
                                           drop feature TagEvent.issue
                                           create feature TagRemoval.issue

2  create feature TagRemoval.issue     ->  identity
   drop feature TagRemoval.issue

3  create feature TagEvent.tag -> Tag  ->  pull up feature TagEvent.tag
   drop feature TagAddition.tag            create feature TagRemoval.timestamp
   drop feature TagAddition.timestamp      pull up feature TagEvent.timestamp
   drop feature TagRemoval.tag             drop feature TagEvent.timestamp

4  pull up feature TagEvent.issue      ->  drop class TagEvent
                                           drop super TagEvent from TagAddition
                                           drop super TagEvent from TagRemoval
                                           extract super TagEvent
                                              {issue, tag, timestamp}
                                              from {TagAddition, TagRemoval}
                                           push down feature TagEvent.tag
                                           push down feature TagEvent.timestamp
```

Fig. 11. Masked detection applied to running example

need to compact the trace again. First, we can rewrite any pair of inverse operators to the identity function, as their effect on the metamodel is canceled out and they are unlikely to have been part of the original evolution. Second, we combine a creation and deletion of two features, which only differ by name into a feature rename. This allows us to detect complex operators, which are masked by a rename, such as a pull up of feature f, followed by a rename of f to f'. Combining rules for inverses requires a normal form grouping on operator category and the renaming rule requires a normal form on feature similarity.

Applying Masked Detection Rules. We apply masked detection rules to the running example. Figure 11 shows the intermediate steps. Step 1 applies feature pull up detection to `TagAddition.issue`. After normalizing the trace, we apply an inverse pattern to creation and drop of `TagRemoval.issue` and reduce the trace (step 2). `TagEvent.issue` is not reduced yet. It will be used later as a component of extracting class `Event`. Next, we repeat steps 1 and 2 by pulling up `tag` and `timestamp` (step 3). Subsequently, the pull of `TagEvent.issue` triggers detection of super class extraction of `TagEvent` in step 4. The drop class, both super drops, and both feature push downs are subsequently neutralized by a class creation, super additions, and pull ups respectively. We then repeat detection of super class extraction for `Event`, using the rename pattern to neutralize create and drops of `timestamp` and `time` as well as `author` and `actor`. Finally, we get the result shown in Figure 1 (bottom).

All regular rewrite rules, which we defined in the previous section, reduced the number of operators in the trace. Furthermore, we did not consider overlapping (interfering) complex operators. These two assumptions enabled fast detection. The rules for detecting masked operators, on the other hand, can increase the size of the trace. For example, the feature pull up pattern increases the trace by the number of occurrences of this feature in sibling classes plus one (for dropping the pulled up feature). Furthermore, for each trace, several rules may be applicable at different positions in the trace. To find a solution, we therefore use a backtracking approach. Each backtracking step tries to apply each of the rules to a trace, yielding zero or more new traces, to which rule application is applied recursively.

6 Related Work

Research on difference detection is found in differencing textual documents, matching structured artifacts, and detection of complex evolution. Text differencing is ignorant of structure or semantics. We discuss related work on matching and complex detection.

Matching. A matching algorithm detects evolution between two artifacts by linking elements of one artifact to elements of the other. Links are either established based on similarity, or using an origin tracking technique such as persistent identifiers. Links are concerned with one element in each artifact. Consequently, matching approaches detect atomic changes. They do not offer support for detecting complex changes. Nevertheless, we discuss them as potential input to our approach. Matching has received attention in the domains of UML, source code reorganization, database schemas and metamodels.

In the domain of UML, Ohst et al. first proposed a solution to compare two UML documents [19]. They compare XML files and use persistent ids for matching. Later work by Xing and Stroulia presents UMLDiff, a matching tool set using similarity metrics instead of persistent ids to establish links [30]. Lin et al. propose a generalization of the work of Xing and Stroulia, which is not restricted to UML models, but uses domain specific models as input instead [16].

In the domain of source code reorganization, Demeyer et al. proposes to find refactorings using change metrics [6]. Later work by Tu and Godfrey uses statistical data and metrics to match evolved software architectures, a process referred to as origin analysis [25].The work on evolving architectures is extended by Godfrey and Zou, by adding detection of merged and split source code entities [10]. In schema matching, a body of work exists, which generally offers a basis for the other works presented in this section. Rahm et al. and later Shvaiko et al. present surveys on schema matching [20,21]. Sun and Rose present a study of schema matching techniques [24].

Lopes et al. consider schema matching applied in the context of model-driven engineering, but propose a new matching algorithm for models [17]. Instead, Falleri et al. take the existing similarity flooding algorithm from the field of schema matching and apply it to metamodels [7]. Work by DelFabro et al. [5] and by Kolovos et al. [15] propose new matching algorithms to the modeling domain. Finally, EMFCompare offers metamodel independent model comparison in the Eclipse Modeling Framework [1]. It relies on heuristic-based matching and differencing, which are both pluggable.

Complex Detection. Detection of complex operators has received significantly less attention in research than matching. Cicchetti et al. discuss an approach for model migration along complex metamodel evolution [3]. They obtain the complex evolution from an arbitrary matching algorithm, but do not offer such an algorithm on their own. Instead, they emphasize the need for a matching algorithm able to detect complex evolution. Our approach fulfills this need. Later work of Cicchetti addresses the problem of dependencies between evolution steps [4]. Since their work focuses only on dependency ordering but not on complex operator detection, they specify operator dependency only statically in terms of the metamodeling formalism. This is too restrictive for the detection of complex operators since it limits possible reorderings dramatically. By defining dependency only in the context of an actual metamodel, our approach enables reordering into various normal forms which allow for the detection of complex operators.

Garcès et al. present an approach to automatically derive a model migration from metamodel differences [9]. The difference computation uses heuristics to detect also complex changes. Each heuristic refines the matching model, and is implemented by a model transformation in ATL. The transformation rules for detecting complex changes are similar to the patterns presented in Section 4. Yet, the approach does not cover operator dependencies, was not able to detect complex changes in a Java case study, and does not address operator masking.

7 Discussion

Metamodeling Formalism. In this paper, we focus only on core metamodeling constructs that are most interesting for coupled evolution of metamodels and models. Concrete metamodeling formalisms like Ecore [23] or MOF [18] provide additional metamodeling constructs like packages, interfaces, operations, derived features, volatile features, or annotations. Since our approach allows for extension, we can add support for these constructs. Therefore, we need to provide additional primitive operators, define their preconditions, extend existing preconditions with respect to new invariants, derive additional complex operators, and define detection patterns for them.

Implementation. We implemented our approach prototypically in *Acoda*[1], a data model evolution tool for WebDSL [28], which is a DSL for web applications. Acoda offers an Eclipse plugin to seamlessly integrate into regular development. The plugin provides editor support for evolution traces (such as syntax highlighting, instant error marking and content completion), generation of SQL migration code, application of migrations to a database, and the evolution detection presented in this paper. The implementation uses an existing data model matching algorithm. We relied on rewrite rules in Stratego [27] to specify each step of the reconstruction algorithm, i.e., mapping data model changes to primitive operators, dependency ordering, normal form rewriting, complex operator detection, and masked operator detection. Acoda presents different evolution traces to the user, who can select and potentially modify the best match.

Trace Selection. Involving the user in the selection process prevents complete automation, but with a rich set of supported coupled operators, detection is likely to yield several suitable traces. Only the user can decide which migration is correct. We can assist this decision by presenting migrations of example models. Conversely, the user can assist the detection by giving examples for original and migrated models. The detection can then drop all traces which cannot reproduce the examples. Additionally, the user may choose to only consider information-preserving traces, thereby narrowing down the set of suitable traces.

[1] http://swerl.tudelft.nl/bin/view/Acoda

Completeness. The set of primitive operators guarantees completeness at the metamodel level as it allows us to evolve any source metamodel to any target metamodel. Completeness at the model level is not feasible since it would imply that we can detect any model transformation between the instances of two arbitrary metamodels. Though, we can add more complex coupled operators to our detection. This increases the search space for both the user and for the detection. As for the user, we have a tradeoff between completeness and usability. There will be many similar operators with minor differences in their migration. Understanding and distinguishing operators becomes harder. In a number of real-life case studies, we identified the most common operators [13]. We propose to support only the detection of these operators and to leave rare cases to the user. As for the detection, supporting more complex operators increases the search space and we have a tradeoff between completeness and performance.

Performance. Besides the number of supported complex operators, detection performance is influenced by evolution size and mask depth, but not by metamodel size, which only affects the matching process. The GMF case study [12] showed us that a larger distance between original and evolved metamodel reduces the matching algorithm precision, making it more unlikely to still detect a good evolution trace. On the other hand, we found that an evolution between two commits to the repository could mostly be captured by 20 evolution steps. A preliminary case study of Acoda on part of the evolution of Researchr[2], a publication management system, showed the applicability of detection. Traces in Researchr between subsequent repository commits are short, hence we applied the detection to steps of ten subsequent commits, which yields traces up to 52 steps in length. A detection run generally takes several seconds and is significantly shortened when reducing the number of commits considered in a single detection run.

Acknowledgments. This research was supported by NWO/JACQUARD project 638.001.610, *MoDSE: Model-Driven Software Evolution*.

References

1. Brun, C., Pierantonio, A.: Model differences in the eclipse modelling framework. UPGRADE, The European Journal for the Informatics Professional (2008)
2. Casais, E.: Managing class evolution in object-oriented systems, ch. 8, pp. 201–244. Prentice Hall International (UK) Ltd. (1995)
3. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: Automating co-evolution in model-driven engineering. In: Enterprise Distributed Object Computing Conference, EDOC. IEEE (2008)
4. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: Managing Dependent Changes in Coupled Evolution. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 35–51. Springer, Heidelberg (2009)
5. Del Fabro, M.D., Valduriez, P.: Semi-automatic model integration using matching transformations and weaving models. In: Proceedings of the 2007 ACM Symposium on Applied Computing, SAC 2007, pp. 963–970. ACM (2007)

[2] http://researchr.org

6. Demeyer, S., Ducasse, S., Nierstrasz, O.: Finding refactorings via change metrics. In: Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2000, pp. 166–177. ACM (2000)
7. Falleri, J.-R., Huchard, M., Lafourcade, M., Nebut, C.: Metamodel Matching for Automatic Model Transformation Generation. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 326–340. Springer, Heidelberg (2008)
8. Favre, J.-M.: Languages evolve too! changing the software time scale. In: IWPSE 2005: Eighth International Workshop on Principles of Software Evolution, pp. 33–42. IEEE (2005)
9. 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)
10. Godfrey, M.W., Zou, L.: Using origin analysis to detect merging and splitting of source code entities. IEEE Transactions on Software Engineering, 166–181 (2005)
11. Herrmannsdoerfer, M., Benz, S., Juergens, E.: COPE - Automating Coupled Evolution of Metamodels and Models. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 52–76. Springer, Heidelberg (2009)
12. Herrmannsdoerfer, M., Ratiu, D., Wachsmuth, G.: Language Evolution in Practice: The History of GMF. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 3–22. Springer, Heidelberg (2010)
13. Herrmannsdoerfer, M., Vermolen, S.D., Wachsmuth, G.: An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models. In: Malloy, B., Staab, S., van den Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 163–182. Springer, Heidelberg (2011)
14. Kniesel, G., Koch, H.: Static composition of refactorings. SCP 52(1-3), 9–51 (2004)
15. Kolovos, D., Di Ruscio, D., Pierantonio, A., Paige, R.: Different models for model matching: An analysis of approaches to support model differencing. In: ICSE Workshop on Comparison and Versioning of Software Models, CVSM 2009, pp. 1–6 (May 2009)
16. Lin, Y., Gray, J., Jouault, F.: DSMDiff: a differentiation tool for domain-specific models. European Journal of Information Systems 16(4), 349–361 (2007)
17. Lopes, D., Hammoudi, S., Abdelouahab, Z.: Schema matching in the context of model driven engineering: From theory to practice. In: Advances in Systems, Computing Sciences and Software Engineering, pp. 219–227. Springer (2006)
18. Object Management Group. Meta Object Facility (MOF) core specification version 2.0 (2006), http://www.omg.org/spec/MOF/2.0/
19. Ohst, D., Welle, M., Kelter, U.: Differences between versions of uml diagrams. In: Proc. of the 9th European Software Engineering Conference, ESEC/FSE, pp. 227–236. ACM (2003)
20. Rahm, E., Bernstein, P.A.: A survey of approaches to automatic schema matching. The VLDB Journal 10(4), 334–350 (2001)
21. Shvaiko, P., Euzenat, J.: A Survey of Schema-Based Matching Approaches. In: Spaccapietra, S. (ed.) Journal on Data Semantics IV. LNCS, vol. 3730, pp. 146–171. Springer, Heidelberg (2005)
22. Sprinkle, J.M.: Metamodel driven model migration. PhD thesis, Vanderbilt University (2003)
23. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0. Addison-Wesley (2009)
24. Sun, X.L., Rose, E.: Automated schema matching techniques: An exploratory study. Research Letters in the Information and Mathematical Science 4, 113–136 (2003)
25. Tu, Q., Godfrey, M.: An integrated approach for studying architectural evolution. In: 10th International Workshop on Program Comprehension, pp. 127–136 (2002)
26. Vermolen, S.D., Visser, E.: Heterogeneous Coupled Evolution of Software Languages. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 630–644. Springer, Heidelberg (2008)

27. Visser, E.: Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9. In: Lengauer, C., Batory, D., Blum, A., Vetta, A. (eds.) Domain-Specific Program Generation. LNCS, vol. 3016, pp. 216–238. Springer, Heidelberg (2004)
28. Visser, E.: WebDSL: A Case Study in Domain-Specific Language Engineering. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007. LNCS, vol. 5235, pp. 291–373. Springer, Heidelberg (2008)
29. Wachsmuth, G.: Metamodel Adaptation and Model Co-adaptation. In: Bateni, M. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 600–624. Springer, Heidelberg (2007)
30. Xing, Z., Stroulia, E.: Umldiff: an algorithm for object-oriented design differencing. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE 2005, pp. 54–65. ACM (2005)

Designing Variability Modeling Languages

Krzysztof Czarnecki

University of Waterloo

Abstract. The essence of software product line engineering (SPLE) is the process of factoring out commonalities and systematizing variabilities, that is, differences, among the products in a SPL. A key discipline in SPLE is variability modeling. It focuses on abstracting the variability realized in the many development artifacts of an SPL, such as code, models, and documents.

This talk will explore the design space of languages that abstract variability, from feature modeling and decision modeling to highly expressive domain-specific languages. This design space embodies a progression of structural complexity, from lists and trees to graphs, correlating with the increasing closeness to implementation. I will also identify a set of basic variability realization mechanisms. I will illustrate the variability abstraction and realization concepts using Clafer, a modeling language designed to support these concepts using a minimal number of constructs. I will also report on the progress towards a Common Variability Language, the Object Management Groups effort to standardize variability modeling, which embodies many of these concepts. I will close with an outlook on the future research challenges in variability modeling.

Formalizing a Domain Specific Language Using SOS: An Industrial Case Study

Frank P.M. Stappers[1], Sven Weber[2], Michel A. Reniers[3], Suzana Andova[1], and Istvan Nagy[1,2]

[1] Dept. of Mathematics and Computer Science, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands
[2] Dept. for Architecture and Platform, ASML, P.O. Box 324, NL-5500 AH Veldhoven, The Netherlands
[3] Dept. of Mechanical Engineering, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands

Abstract. This paper describes the process of formalizing an existing, industrial domain specific language (DSL) that is based on the task-resource paradigm. Initially, the semantics of this DSL is defined informally and implicitly through an interpreter. The formalization starts by projecting the existing concrete syntax onto a formal abstract syntax that defines the language operators and process terms. Next, we define the dynamic operational semantics at the level of individual syntactical notions, using structural operational semantics (SOS) as a formal meta-language. Here, the impact of the formalization process on the DSL is considered in terms of disambiguation of underlying (semantic) language design decisions.

1 Introduction

Modern manufacturing systems coordinate concurrent components to successfully manufacture products. As a result, the governing control software has to support complex execution, coordination and optimize scenarios.

To cope with the complexity of control software, model-driven engineering (MDE) techniques have been widely adopted over the last decade. MDE treats models as first class entities and aims at reducing development effort and lead-time when compared to more traditional software engineering techniques. Nowadays, models are increasingly created using domain specific languages (DSLs). DSLs are languages that define the jargon [19] of a particular class of problem domains or set of domain aspects. Executable DSLs hide software implementation details by generating executable models or code from concrete domain models.

DSL design starts with the definition of an *abstract syntax* (grammar) relating domain notions such as verb, noun and adverb. In MDE this typically results in a meta-model. For an abstract syntax, a textual or graphical *concrete syntax* can be defined to create syntactically correct terms such as the sentence 'The cow flies red.'. Clearly, syntax (form) needs semantics (meaning) to obtain a meaningful specification. *Static semantics* define aspects like well-formedness, typing and structure of concrete models. *Dynamic semantics* define the model of computation for the (composed) execution behavior of a DSL's syntactical notions. In practice, the semantics of a DSL is often defined implicitly and informally through (i) translations to a (formal) language and/or executable language, or (ii) an engine/interpreter that processes concrete domain models. Having well defined, processable semantics should enable automated reasoning on the execution behavior of concrete models. However, even with explicit and formal semantics, limited or no tool support exists for semantic mappings and automated formal reasoning.

We propose a two-step approach that facilitates both semantic mappings and automated formal reasoning on concrete domain models. First, we define the semantics for the DSL's individual syntactical components in an intermediate formal meta-language. Second, we transform the DSL's formal semantics, along with a syntactic interpretation, to facilitate automated analysis [26]. Since both steps are non-trivial, this paper focuses on the first step and illustrates that it can uncover sub-optimal design decisions and ambiguities. Also, we illustrate how to minimize the impact of formalization while retaining backward compatibility. During formalization both the relational structure of domain notions and their behavioral effect(s) are considered. We use structural operational semantics (SOS) [23] to define the effect of language terms from an operational perspective. SOS is widely applicable given its mathematical expressiveness, e.g. support for higher-order functions and compositionality. Also, SOS has been successfully used to define the formal semantics of a wide variety of languages [2,13,27,28].

We consider the formalization of an existing, industrial DSL called Task Resource Control System (TRECS) [29]. TRECS supports the definition of predictive and reactive rules to optimally allocate manufacturing activities (tasks) to mechatronic subsystems (resources) over time while subjected to dynamic constraints. TRECS domain models are specified using, among others, a graphical syntax that resembles UML activity diagrams, where tasks correspond to activities. Here, tasks are executed concurrently taking conditions, predecessors and availability of resources into account. Also, tasks are annotated with resource usage including resource state pre- and postconditions (not discussed in this paper). Since its conception, the DSL has matured in an industrial setting where many - non-disclosed - reactive concepts like (dynamic) priorities and exceptions have been added. Although we formalized the entire DSL, this paper discloses only a small subset. To illustrate, we use this entire subset to model an unconventional example: a simple recipe to create the Italian dessert Tiramisu. We refer to [12,29] for examples on the use of TRECS to control wafer scanners.

The remainder of this paper is structured as follows. Section 2 discusses related work. In Section 3 we introduce the (concrete syntax of) disclosed domain notions using our running example: making Tiramisu. Also, we project the concrete syntax onto a formal abstract syntax that contains the operators and process variables used to define the formal semantics in SOS. Section 4 defines the dynamic semantics, which are validated with domain experts and language engineers. We evaluate our approach and results in Section 5. The paper is concluded in Section 6.

2 Related Work

The DSL as formalized in this paper, is loosely based on the UML [24] modelling format. As such, the formalization of UML Statechart Models [9,18], UML State Machines [20,22], UML Sequence Diagrams [1], and UML Activity Diagrams [5] can be considered as a starting point. Here, design constraints can be captured in the Object Constraint Language (OCL) [7]. As DSLs add domain specific notions, the usability of these existing formal definitions can be significantly reduced depending on the complexity and nature of the changes. In our case, the non-disclosed changes are quite extensive and include scheduling, dispatching logic, exception handling and more. When considering TRECS as a separate language, rather than one specialized from UML, we find many frameworks and methods that *transform* DSLs and/or their concrete models in such a way that formal syntax and semantics are assigned [21].

The framework of [11] restricts modeling languages in a way that only descriptions of possible domain configurations are mappable. First, domain (ontological) semantics are assigned to language constructs. Second, the ontological assumptions are identified by administering the elements of the domain and their relationships. Third, the ontological assumptions are transferred and become the rules that restrict the use of the language constructs and limit the statements to the specific domain. Finally, they construct the meta-model from these rules.

A similar approach is taken by [17] where the meta-model is formalized bottom-up. They start from a simple core that defines the syntax for a class of DSLs. Next, a relating class, i.e. transformation, is defined to relate syntactic elements of one domain with elements of another. Then, a special element is introduced that can generate all the domains for a particular class, i.e. the meta-model. Finally, formal Horn logic [14] is added to preserve and formulate various properties over different domains using the FORMULA [16] theorem prover. In the work of [11,17], a meta-model is created that describes the constructs that specify the commonalities and/or differences between DSLs. Meta-models are expressed using OCL and class diagrams that define relationships [6]. Our route is similar since we take basic notions and create a syntactical meta-model for them. Rather than constraining class diagrams, we provide an actual model of computation through SOS. This allows us to specify the behavior *mathematically* for each syntax element in isolation and provide a compositional language.

The work of [8] shows a pragmatic and instrumented approach towards providing operational semantics for DSLs. First, they sketch assigning semantics in an axiomatic, operational, or denotational (translational) manner, based on the DSL's taxonomy. Based on the selected adequate target language, a mapping is provided that preserves the semantic relation. The proof for preservation is also considered. Our approach is similar but we use operational semantics instead. Operational semantics are preferred when considering the semantics of complex, composed language terms. In [31] operational semantics in MDE are explored for a small academic language.

Since we demonstrate feasibility of the operational approach for a large, industrial language, other aspects (like backward compatibility) need to be considered. Our work supersedes this scope as it fits and complements both approaches.

Finally, in [10,25] different approaches are taken to assign dynamic semantics of DSLs in the context of MDE. Here, dynamics are assigned through Abstract State Machines (ASMs) [4], with extensions to Prolog [30] and Scheme [15]. As the underlying semantics of ASMs can be formally defined in SOS [27], we demonstrate that any intermediate semantic definitions can be omitted.

3 Formalizing Domain Notions

To illustrate and discuss all *disclosed features* of TRECS, we consider a concrete model to create a simple Tiramisu. This is explained in detail in Section 3.1. In Section 3.2 we project our example's concrete syntax onto a formal abstract syntax. We validate our formal syntax in Section 3.3.

3.1 Running Example

To create Tiramisu, we require ingredients, kitchen utensils and appliances, a recipe and a way of working as specified in Fig. 1. In the DSL, ingredients, kitchen utensils and appliances are called "resources". The recipe and its sub-recipes are called "subplans" and the individual activities in a subplan are called "tasks".

ingredient	cream topping	coffee syrup	assembling
milk	500 ml		
white sugar	150 gram	75 gram	
flour	35 gram		
egg yolks	6 pcs.		
dark rum	60 ml	60 ml	
vanilla extract	2 teasp.		
mascarpone cheese	225 gram		
ladyfingers			32 pcs.
espresso		360 ml	
butter	60 gr		

A. Make Tiramisu

1. **Make Cream Topping** - see **B**
2. **add mascarpone** - using a wooden spoon, beat 225 gram mascarpone cheese in a bowl until it is soft and smooth. Then gently whisk the mascarpone into the result from B.6 until the custard mixture is smooth.
3. **mix coffee syrup** - in a large shallow bowl combine 360 ml espresso, 75 grams sugar and 60 ml dark rum.
4. **line loaf pan** - before making layers, take a loaf pan and line it with plastic wrap and making sure that the plastic wrap extends outside the loaf pan to allow wrapping.
5. **Make Layers** - see **C**
6. **cool down cake** - once all layers are made, cover the Tiramisu with plastic wrap and place it in a refrigerator and have it cool for at least 6 hours.
7. **present and serve** - once the cake is cooled, remove the plastic wrap from the top and gently invert the Tiramisu from the loaf pan to a serving plate. Remove remaining plastic wrap and serve the dessert.

– continued on next page –

B. Make Cream Topping

1. **boil sweet milk** - in a saucepan heat 430 ml milk and 100 grams sugar right up to the boiling point.
2. **egg yolk mixture** - meanwhile whisk 70 ml milk, 50 grams sugar, 35 grams flour and 6 egg yolks in a heatproof bowl.
3. **whisk milk & egg yolk** - once the sweet milk has just come to a boil gradually whisk it into the egg yolk mixture. Transfer this mixture into a large saucepan.
4. **reduce mixture** - next slowly cook the result from B.3 while stirring constantly until it comes to a boil. Once it boils, continue to whisk the mixture constantly for another minute and let it reduce a bit.
5. **enrich mixture** - take the result from B.4 of the heat and strain into a large bowl. Whisk in 60 ml dark rum, 2 teaspoons vanilla extract, and 60 grams butter. Cover the bowl with plastic wrap to prevent crust-forming.
6. **cool down topping** - place the bowl in the refrigerator and let it cool down for approximately two hours.

C. Make Layers

1. **8 fingers?** - ensure that we have 8 ladyfingers to create a layer.
2. **dip fingers** - one ladyfinger at a time, dip 8 ladyfingers into the coffee syrup from step A.3 and place them side by side in the loaf pan.
3. **cover with cream** - spoon 1/4 of the custard from A.2 and completely cover the 8 ladyfingers.
4. **make Layers** - repeat **Make Layers** until no more ladyfingers are left.

Fig. 1. A recipe for a simple Tiramisu

The DSL's activity diagram-like concrete syntax is illustrated using Fig. 2. Fig. 2d shows resource definition through a hash-like table. An initialization is required to execute a concrete model. We assume we Make Tiramisu only once, using exactly those resources required. This results in an initialization of ingredients, kitchen utensils and appliances as illustrated in Fig. 2e.

3.2 Concrete Syntax Projection

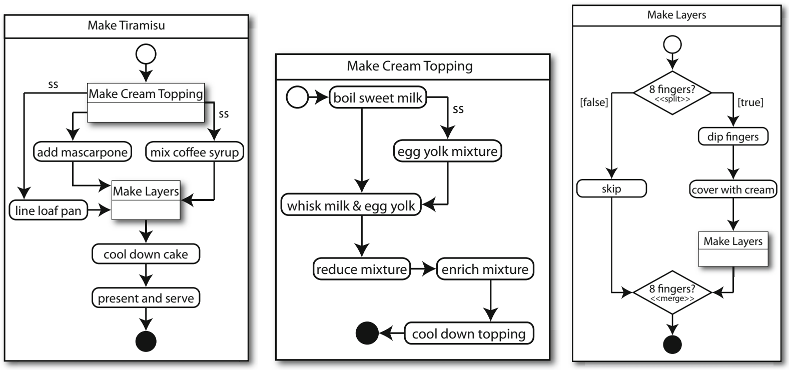
We start by identifying and projecting the most elementary notions in terms of behavior to obtain a compositional formal syntax. If we cannot capture the intended behavior by any of the already introduced notions, we either add new or refine existing notions. Note that we choose a process algebra-like notation like in [2] and reuse process algebraic operators where possible (see Section 4).

Task. A task is the smallest identifiable behavior in the system under control. In Fig. 2a, add mascarpone and mix coffee syrup are tasks. The concrete syntax of a task is a rounded rectangle (node) with a name label.

Decision 1. The execution of a labeled task is atomic and observable. □

Decision 2. In some cases we do not want observable behavior. This is shown in Fig. 2c by a task labeled with the reserved word skip. This is represented by a process term τ. For now, $\mathcal{T}$ denotes the finite set of all tasks including τ. □

Precedence Relation. A finish-start precedence (FS) relation can be used to start behavior if and only if the preceding behavior has terminated successfully.



(a) Tiramisu subplan (b) Cream Topping subplan (c) Layers subplan

task	consumes	produces
add mascarpone	wooden spoon: 1 pcs	wooden spoon: 1 pcs
	mascarpone: 225 gr	bowl : 1 pcs
	cooled topping: 1 pcs	custard: 4 vol.
	bowl: 1 pcs	
⋮	⋮	⋮

(d) Resource usage

resource(s)	init
milk	480 ml
sugar	225 gr
wooden spoon	1pcs
refrigerator	1 pcs
⋮	⋮

(e) Resouce initialization

Fig. 2. Partial Tiramisu recipe in DSL's concrete syntax

This is shown in Fig. 2a as a labelless, directed edge between cool down cake and present and serve. A start-start precedence (SS) relation can be used to start behavior if and only if preceding behavior has started its execution. This is shown in Fig. 2b as a directed edge with label SS between boil sweet milk and egg yolk mixture. In practice, such a relation could be used to the capture queued execution of activities on mechatronic subsystems.

Decision 3. For each type of relation we introduce a dedicated operator. Let p and q be process terms. Then a finish-start relation in the abstract syntax is expressed by

$$p \cdot q$$

The start-start relation in abstract syntax is expressed by

$$p \parallel\!\!\lfloor q$$

□

Decision 4. Supersedes Decision 1 where tasks are considered atomic. Atomicity implies that if p would be a single task t, we could not distinguish the behavior of the $\parallel\!\!\lfloor$ operator from that of the $\cdot$ operator. To make this observable, we introduce an explicit start and finish action for all labeled tasks except τ. Let $\mathcal{T}_\alpha$ be a finite set of elements called starting tasks, i.e. the alphabet of starting tasks. Let $\mathcal{T}_\omega$ be a finite set of elements called finishing tasks, i.e. the alphabet of finishing tasks. We refine a labeled task such that $t_\alpha \in \mathcal{T}_\alpha$ and $t_\omega \in \mathcal{T}_\omega$ denote the start respectively the finish of task t. For now, we assume that for every action t_α performed, t_ω will always follow eventually (see Section 4.2, Decision 18). From this point forward, we refer to this as performing a (labeled) task t and will denote the associated behavior by a single t_α. We denote $\mathcal{T} = \mathcal{T}_\alpha \cup \mathcal{T}_\omega \cup \{\tau\}$, where $\tau \notin \mathcal{T}_\alpha \cup \mathcal{T}_\omega$. □

Choice. The execution of behavior can be conditionally based on some decision, as shown in Fig. 2c where decision 8 fingers? tests if we have sufficient ladyfingers. The concrete syntax of a choice consists of a split diamond that is closed by a merge diamond. The alternative conditional behavior (branches) is specified in-between making every branch in the choice syntactically finite. Splits have a finite number of outcomes making the number of branches n also finite. Here, each outcome corresponds to one edge that is annotated with a squared bracketed outcome label. Note that for the split diamond, the incoming edges have relevance in terms of precedence relations, for the merge diamond this holds for the outgoing edges.

Decision 5. We map a branch i to its corresponding process term p_i, and define an operator across all branches. Now, assuming a decision function d that maps each evaluation outcome to exactly one branch process term, we obtain

$$\bigvee_d \langle p_1, \ldots, p_n \rangle$$

□

Decision 6. The choice operator acts on the state of the system. This means we require a mechanism to store the actual state. Let Λ denote the set of all values and let $\mathcal{V}$ denote the set of all variables. Then $\Sigma = \mathcal{V} \to \Lambda$ denotes the set of all variable valuations. A variable valuation is a total function that captures the values of the variables. Now, $\sigma \in \Sigma$ denotes a variable valuation where σ is the state vector that stores the variable valuation observed by the system's behavior. Now, the evaluated decision function will be of the form $d : \Sigma \to \mathbb{N}$ where each valuation will correspond to exactly one branch process term. □

Concurrent Execution. We would like tasks to execute concurrently where possible to e.g. maximize the output of the system under control. Execution can be forked and merged using multiple ingoing/outgoing precedence relations, possibly of different types. The concrete syntax captures forking [joining] concurrent behavior implicitly, without an explicit sync bar, through multiple outgoing [incoming] edges as shown in Fig. 2b for example boil sweet milk. Note that - unlike in more strictly scoped forks [joins] of activity diagrams - tasks can have predecessors belonging to different forks [joins].

Decision 7. We duplicate task labels to force synchronization. Let p and q be process terms. For duplicate task labels in p and q we force synchronization while for other labels we allow full interleaving by writing

$$p \parallel q$$

Here, only terms that occur on both sides of the operator are performed simultaneously. Terms that are on either one side of the operator can be executed concurrently as long as their precedence relations are respected. The use of this operator and the formal syntax in general are clarified using Fig. 3. □

Decision 8. Extends Decision 4 where task start and finish actions were implicitly considered to be unique. Since task labels are now duplicated, they cannot be distinguished anymore when they are mapped from the concrete to the abstract syntax. We refine the definition of labeled task t by extending its process terms with an unique identifier $i \in \mathbb{N}$ such that t_α becomes t^i_α and t_ω becomes t^i_ω. As diagrams are syntactically finite, the set of unique labels that needs to be assigned is also finite. □

Composition. Tasks can be placed in a named group called a subplan to enable reuse and nesting. A subplan is represented by a square labeled box that contains behavior in te form of labeled tasks. In Fig. 2, Make Tirasmisu, Make Cream Topping and Make Layers are subplans. Subplans can reference subplans (including itself). A reference is represented by a smaller square labeled box that does not contain any modeled behavior.

Decision 9. We introduce process equations to facilitate composition. Let $\mathcal{S}$ denote the set of subplan labels, disjoint from the set of task labels, i.e. $\mathcal{S} \cap \mathcal{T} = \emptyset$. Furthermore, we require that the process equations are orthogonal, that is, every left-hand variable in the process equation may only be defined once. Now, let $A \in \mathcal{S}$ describe the behavior for process term p by the equation

$$A \equiv p$$

□

Decision 10. Extends Decision 8 as process equations may result in behavior inside of a subplan to become potentially indistinguishable. As an example, consider a single task label that is used and instantiated in two different subplans. We assume $P : \mathit{List}(\mathcal{S})$ to be a list of subplan labels at which actions t^i_α and t^i_ω are execute such that during execution we observe $t^{i,P}_\alpha$ and $t^{i,P}_\omega$. Note that considering the left-hand side of the equation as the parent node and the element on the right as its child, we can infer a tree-like structure. Here, it must hold that p does not contain identical subplan labels to obtain unique paths. Also, subplan references must be uniquely distinguishable. □

Decision 11. To mark the initial process we introduce a special keyword *init* that marks the initial process term p. We assume that every specification has exactly one initialization, which is expressed by

$$\mathsf{init}\ p$$

□

Resource. A task consumes a set of resource labels when it starts its execution and produces a set of resource labels when it finishes. Both sets can be empty. All tasks with the same label are of the same (proto)type: they produce and consume the exact same amount of each resource label. In the DSL, these definitions are stored separately as shown for task add mascarpone in Table 2d. Note that some resource labels (such as wooden spoon) are *used* by consuming them at the beginning of a task and returning them at the end.

Decision 12. We assume that a task claims all required resource during its entire execution. So, early resource release is not considered. Let $\mathcal{R}$ denote the finite set of resource labels. Now, $R_Q : \mathcal{T}_\alpha \to (\mathcal{R} \to \mathbb{N})$ denotes the (possibly empty) set of resources required to start execution of a task. Similarly, $R_P : \mathcal{T}_\omega \to (\mathcal{R} \to \mathbb{N})$ denotes the (possibly empty) set of resources that is produced when the execution of a task finishes. The amount of resources that are available is denoted by $R_A : \mathcal{R} \to \mathbb{N}$. The resource usage is encoded in the state vector resulting in R_A as a reserved variable in σ, whereas R_Q and R_P are globally given. □

3.3 Formal Syntax Validation

At this point, domain experts and language engineers are involved to mature the formal abstract syntax. That is, we validate the expected behavior of composed operators based on the informal execution semantics as presented in Section 3. For illustrative purposes we reconsider the subplans from Fig. 2 and write them in the formal abstract syntax as shown in Fig. 3. Using our running example we can illustrate two out of five detected ambiguities. The remaining ambiguities concern non-disclosed parts of the language.

To illustrate the first ambiguity, we reconsider Fig. 2b and replace the sequential composition of reduce mixture (B.4) and enrich mixture (B.5) by a preemptive sequential composition. We keep the sequential composition with Cool down topping (B.6) such that we get $\mathsf{B.4} \mathbin{\underline{|||}} \mathsf{B.5} \cdot \mathsf{B.6}$. The result is shown in Fig. 4a. Next,

$$\begin{aligned}
\textsf{Make Tiramisu} \equiv\ & ((\textsf{Make Cream Topping} \mathbin{\underline{|||}} \textsf{line loaf pan}_\alpha \cdot \textsf{Make Layers}) \perp\!\!\!\perp \\
& (\textsf{Make Cream Topping} \cdot \textsf{add mascarpone}_\alpha \cdot \textsf{Make Layers}) \perp\!\!\!\perp \\
& (\textsf{Make Cream Topping} \mathbin{\underline{|||}} \textsf{mix coffee syrup}_\alpha \cdot \textsf{Make Layers})) \cdot \\
& \textsf{cool down cake}_\alpha \cdot \textsf{present and serve}_\alpha
\end{aligned}$$

$$\begin{aligned}
\textsf{Make Cream Topping} \equiv\ & ((\textsf{boil sweet milk}_\alpha \mathbin{\underline{|||}} \textsf{egg yolk mixture}_\alpha \cdot \\
& \textsf{whisk milk \& egg yolk}_\alpha) \perp\!\!\!\perp (\textsf{boil sweet milk}_\alpha \cdot \\
& \textsf{whisk milk \& egg yolk}_\alpha)) \cdot \textsf{reduce mixture}_\alpha \cdot \\
& \textsf{enrich mixture}_\alpha \cdot \textsf{cool down topping}_\alpha
\end{aligned}$$

$$\textsf{Make Layers} \equiv \bigvee_{\textsf{8 fingers?}} \langle\ \tau,\ \textsf{dip fingers}_\alpha \cdot \textsf{cover with cream}_\alpha \cdot \textsf{Make layers}\ \rangle$$

$$\textsf{init Make Tiramisu}$$

Fig. 3. Subplans and their initialization in DSL's formal syntax. We assume operator priorities as discussed at the end of Section 3.3

we define $\mathsf{E} \equiv \mathsf{B.4} \mathbin{|||} \mathsf{B.5}$ as illustrated by Fig. 4b. Based on syntactic replacement, domain experts expect from both Fig. 4a and Fig. 4b that $\mathsf{B.6}_\alpha$ can occur before $\mathsf{B.4}_\omega$. However, the DSL's legacy implementation performs a mathematical substitution such that brackets are placed around E, which changes the dynamic behavior to where E now needs to successfully terminate before $\mathsf{B.6}_\alpha$ can occur.

Decision 13. Changing the execution semantics of existing operators can adversely affect behaviors from validated and implemented concrete legacy models. So, to preserve backward compatibility for concrete models written in the informal DSL, a new "all finish-start" precedence relation is added. We use a directed edge annotated with an FS^ label. This precedence relation can only be used in conjunction with subplan references. Let p be a reference and q be any process term then the all finish-start behavior is obtained by $(p) \cdot q$. □

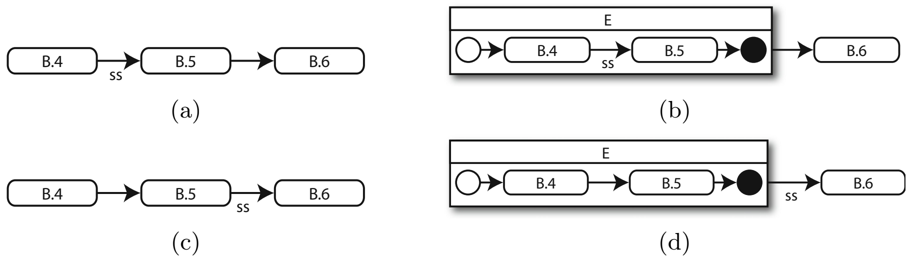


Fig. 4. Disambiguation on finish–start and start–start relations

To illustrate the second ambiguity, we reconsider the same fragment as before but assume $\mathsf{B.4} \cdot \mathsf{B.5} \mathbin{|||} \mathsf{B.6}$ instead as shown in Fig. 4c. Here, domain experts expected $\mathsf{B.6}_\alpha$ can occur as soon as $\mathsf{B.5}_\alpha$ occurs and $\mathsf{B.4}_\omega$ to occur before $\mathsf{B.5}_\alpha$. Next, we define $\mathsf{E} \equiv \mathsf{B.4} \cdot \mathsf{B.5}$ and write $\mathsf{E} \mathbin{|||} \mathsf{B.6}$ as shown in Fig. 4d. Considering syntactic replacement, we expect that $\mathsf{B.6}_\alpha$ can occur no earlier than after performing $\mathsf{B.5}_\alpha$. Domain engineers, however, expect that $\mathsf{B.6}_\alpha$ may occur after performing $\mathsf{B.4}_\alpha$. The intuition behind, is that we consider a composite term 'in progress' as soon as some start action is observed and *not* when all start actions have been observed.

Decision 14. To preserve backward compatibility for concrete models written in the informal DSL, a new "any start-start" precedence relation is added. We use a directed edge annotated with an SS\$ label. This precedence relation can only be used in conjunction with subplan references. Let p be a reference and q be any process term then the any start-start behavior is obtained by

$$p \mathbin{\|\!\lfloor} q$$

□

In all, for each concrete instance of the legacy precedence relations in conjunction with a subplan reference, domain experts will have to decide to use the new or the the old operator. The resulting, validated formal syntax and taxonomy for TRECS are given in Table 1. Here, $p_{\text{semantics}}$ is a placeholder term (see Section 4.3). The descending order of operators is defined as

$$``\cdot", \{``\lfloor\!\lfloor", ``\perp\!\!\!\perp", ``\lfloor\!\lfloor\!\lfloor"\}, ``\bigvee_d"$$

Of these operators "$\perp\!\!\!\perp$", "$\lfloor\!\lfloor\!\lfloor$", and "$\cdot$" associate to the right. Priorities can be overruled by using parentheses "(" and ")".

Table 1. Formal abstract syntax and taxonomy for TRECS

process term	operator name	variable	description
$p ::= \tau$	skip		
$t_{\alpha}^{i,P}$	start of a task t	$i : \mathbb{N}$ $P : List(\mathcal{S})$	finite identifier list of process definition labels
$t_{\omega}^{i,P}$	finish of a task t	$i : \mathbb{N}$ $P : List(\mathcal{S})$	finite identifier list of process definition labels
$p \cdot p$	sequential composition		
$p \lfloor\!\lfloor\!\lfloor p$	preemptive sequential composition		
$p \lfloor\!\lfloor p$	left merge composition		
$\bigvee_d \langle p_1, \ldots, p_n \rangle$	conditional choice	$d : \Sigma \to \mathbb{N}$ $n : \mathbb{N}$	decision function finite branch number
$p \perp\!\!\!\perp p$	synchronized parallel composition		
$A \equiv p$	process definition	$A \in \mathcal{S}$	
$p_{\text{semantics}}$	grammar for defining semantics		

symbol	description
Λ	set of all values
$\mathcal{V}$	set of all variables
$\Sigma = \mathcal{V} \to \Lambda$	set of all variable valuations
$\sigma : \Sigma$	state vector with active variable valuations
$\mathcal{T}_{\alpha}$	finite set of task starts
$\mathcal{T}_{\omega}$	finite set of task finishes
$\mathcal{T}$	finite set of tasks, where $\mathcal{T}_{\alpha} \cup \mathcal{T}_{\omega} \cup \{\tau\}$ and $\tau \notin \mathcal{T}_{\alpha} \cup \mathcal{T}_{\omega}$
$R_A : \mathcal{R} \to \mathbb{N}$	reserved variable in σ denoting available resources
$R_Q : \mathcal{T}_{\alpha} \to (\mathcal{R} \to \mathbb{N})$	reserved constant in σ denoting the required resources to start execution of a task
$R_P : \mathcal{T}_{\omega} \to (\mathcal{R} \to \mathbb{N})$	reserved constant in σ denoting the produced resources at the finish of a task
$\mathcal{S}$	finite set of process equations where $\mathcal{S} \cap \mathcal{T} = \varnothing$
I	initial subplan label where $I \in \mathcal{S}$

4 Formalizing Dynamic Semantics

We use SOS to assign dynamic operational semantics to DSL process terms (abstract notions). SOS associates a labeled transition system to terms, where action transitions describe the discrete behavior. First, we explain SOS and its semantic notions. Then we assign semantics for the TRECS' individual process terms.

4.1 Semantic Preliminaries

Process. A process is a tuple $\langle p, \sigma \rangle$, where p denotes a process term for an element of an activity diagram, and $\sigma \in \Sigma$ denotes a variable valuation.

Transition. A transition between process terms describes a state change, there by observing a possible action that is represented by a label.

Decision 15. We limit ourselves to observing the executed task and its associated resources, we choose to reveal at most this information on each transition. As such, a label consists of two elements: (i) the label of the executed task and (ii) the associated set of resources. A transition dictates either continuative behavior or successful termination. □

Continuative action transitions : $\_ \xrightarrow{\_} \_ \subseteq (\mathcal{P}\times\Sigma)\times(\mathcal{X}\times(\mathcal{R}\to\mathbb{N}))\times(\mathcal{P}\times\Sigma)$, where $\mathcal{X}$ is (i) $\mathcal{T}$ when an internal action is performed, or (ii) $\mathcal{T}\times\mathbb{N}\times\mathcal{L}(\mathcal{S})$ when the start of a task is performed. The intuition of an action transition $\langle p,\sigma\rangle \xrightarrow{t,R} \langle p',\sigma'\rangle$ is that process $\langle p,\sigma\rangle$ performs the discrete action (t,R), thereby transforming into process $\langle p',\sigma'\rangle$. σ' denotes the corresponding valuation of process p' after performing transition t, associating resources R.

Terminating action transitions : $\_ \xrightarrow{\_} (\checkmark,\_) \subseteq (\mathcal{P}\times\Sigma)\times(\mathcal{X}\times(\mathcal{R}\to\mathbb{N}))\times(\mathcal{P}\times\Sigma)$, where $\mathcal{X}$ is the same as for continuative action transitions. The intuition of a termination transition $\langle p,\sigma\rangle \xrightarrow{t,R} \langle\checkmark,\sigma'\rangle$ is that process $\langle p,\sigma\rangle$ transforms into $\langle\checkmark,\sigma'\rangle$, by performing the discrete action (t,R). $\checkmark$ denotes successful process termination.

Transition System Specification. A transition system specification [3] denotes a set of deduction rules. A deduction rule has the form $\frac{H}{C}$ where H is a set of transition formulae (*premises*) and C is a transition formula (*conclusion*). To derive the conclusion, and perform an action, all premises need to be satisfied.

4.2 Abstract Syntax Projection

Skip. The τ is defined as Table 2 **(skip)**. A τ action is an internal action that cannot be observed nor claim resources.

Decision 16. τ cannot change the state vector σ is not updated. The no resource claim $\forall_{r\in\mathcal{R}}\{R(r)=0\}$, using an auxiliary function R that maps all resource labels to zero, is represented by $\varnothing$. □

Start of a Task. The start of a task is defined as Table 2 **(start-task)**. $t_{\alpha}^{i,P}$ is the action that starts task t. To perform $t_{\alpha}^{i,P}$, the required resources $R_Q(t_\alpha)$ must be available.

Decision 17. The resource availability is expressed by premise $(\sigma(R_A)\geq R_Q(t_\alpha))$. As all functions are total, we assume point-wise evaluation. If $t_{\alpha}^{i,P}$ is performed, we observe $t_{\alpha}^{i,P}\in\mathcal{T}_\alpha$ where i is the unique identifier and P is the subplan hierarchy, thereby claiming resources $R_Q(t_\alpha)$. □

Decision 18. To ensure that $t_{\omega}^{i,P}$ follows after $t_{\alpha}^{i,P}$, we rewrite the term to a term that performs the finish of task t. The number of claimed resources is subtracted from the available resources, reflected by $\sigma[R_A\to\sigma(R_A)-R_Q(t_\alpha)]$. □

Finish of a Task. The finish of a task is defined asTable 2 **(finish-task)**. $t_\omega^{i,P}$ is the action that finishes task t.

Decision 19. Any release of claimed (produced) resources are added to the set of available resources, reflected by $\sigma[R_A \to \sigma(R_A) + R_P(t_\omega)]$. The set of premises is empty, so the finish of a task is performed unconditionally. We observe $t_\omega^{i,P}$ and $R_P(t_\omega)$ on the transition and rewrite $t_\omega^{i,P}$ to $\checkmark$ to terminate. □

Sequential Composition. Table 2 **(FS)** defines the sequential composition.

Decision 20. We follow the standard semantics given in literature, e.g. [13]. Here, $p \cdot q$ behaves as q, if p successfully terminates after performing action (β, ρ), i.e. the upper case of Table 2 **(FS)**. If p, by performing action (β, ρ) becomes p', then the process $p \cdot q$ behaves as $p' \cdot q$, i.e. the lower case of Table 2 **(FS)**. □

Preemptive Sequential Composition. The preemptive sequential composition is defined as Table 2 **(SS1)** and **(SS2)**. Here, we want a right term of the operator to perform actions iff a left term *can* successfully terminate.

Decision 21. **(SS1)** defines the behavior when term p performs a transition. Whenever p successfully terminates, the them continues as q. If p continues as p', the term continues as $p' \mathbin{|\!|\!|} q$. Informally, rule **(SS2)** expresses that q can

Table 2. Operational rules for the basic operators

$$\textbf{(skip)}\ \frac{}{\langle \tau, \sigma\rangle \xrightarrow{\tau,\varnothing} \langle \checkmark, \sigma\rangle}$$

$$\textbf{(start-task)}\ \frac{\sigma(R_A) \geq R_Q(t_\alpha)}{\langle t_\alpha^{i,P}, \sigma\rangle \xrightarrow{t_\alpha^{i,P}, R_Q(t_\alpha)} \langle t_\omega^{i,P}, \sigma[R_A \to \sigma(R_A) - R_Q(t_\alpha)]\rangle}$$

$$\textbf{(finish-task)}\ \frac{}{\langle t_\omega^{i,P}, \sigma\rangle \xrightarrow{t_\omega^{i,P}, R_P(t_\omega)} \langle \checkmark, \sigma[R_A \to \sigma(R_A) + R_P(t_\omega)]\rangle}$$

$$\textbf{(FS)}\ \frac{\langle p, \sigma\rangle \xrightarrow{\beta,\rho} \left\langle \begin{matrix}\checkmark \\ p'\end{matrix}, \sigma' \right\rangle}{\langle p \cdot q, \sigma\rangle \xrightarrow{\beta,\rho} \left\langle \begin{matrix} q \\ p' \cdot q\end{matrix}, \sigma' \right\rangle} \qquad \textbf{(SS1)}\ \frac{\langle p, \sigma\rangle \xrightarrow{\beta,\rho} \left\langle \begin{matrix}\checkmark, \sigma' \\ p', \sigma'\end{matrix} \right\rangle}{\langle p \mathbin{|\!|\!|} q, \sigma\rangle \xrightarrow{\beta,\rho} \left\langle \begin{matrix} q, \sigma' \\ p' \mathbin{|\!|\!|} q, \sigma'\end{matrix} \right\rangle}$$

$$\textbf{(SS2)}\ \frac{\langle p, \sigma\rangle \xrightarrow{\beta,\rho} \langle \checkmark, \sigma'\rangle, \quad \langle q, \sigma\rangle \xrightarrow{\beta',\rho'} \left\langle \begin{matrix}\checkmark \\ q'\end{matrix}, \sigma'' \right\rangle, \quad \langle p, \sigma''\rangle \xrightarrow{\beta,\rho} \langle \checkmark, \sigma'''\rangle}{\langle p \mathbin{|\!|\!|} q, \sigma\rangle \xrightarrow{\beta',\rho'} \left\langle \begin{matrix} p \\ p \mathbin{|\!|\!|} q'\end{matrix}, \sigma'' \right\rangle}$$

$$\textbf{(SS\$)}\ \frac{\langle p, \sigma\rangle \xrightarrow{\beta,\rho} \left\langle \begin{matrix}\checkmark, \sigma' \\ p', \sigma'\end{matrix} \right\rangle}{\langle p \mathbin{\|\!\lfloor} q, \sigma\rangle \xrightarrow{\beta,\rho} \left\langle \begin{matrix} q, \sigma' \\ p' \rfloor \varnothing \lfloor q, \sigma'\end{matrix} \right\rangle} \qquad \textbf{(C)}\ \frac{\langle p_{d(\sigma)}, \sigma\rangle \xrightarrow{\beta,\rho} \left\langle \begin{matrix}\checkmark \\ p'\end{matrix}, \sigma' \right\rangle}{\langle \bigvee_d \langle p_1, \ldots, p_n\rangle, \sigma\rangle \xrightarrow{\beta,\rho} \left\langle \begin{matrix}\checkmark \\ p'\end{matrix}, \sigma' \right\rangle}\ d(\sigma) \in [1, n]$$

$$\textbf{(spc)}\ \frac{\langle p \rfloor sync(p) \cap sync(q) \lfloor q, \sigma\rangle \xrightarrow{\beta,\rho} \left\langle \begin{matrix}\checkmark \\ p'\end{matrix}, \sigma' \right\rangle}{\langle p \perp\!\!\!\perp q, \sigma\rangle \xrightarrow{\beta,\rho} \left\langle \begin{matrix}\checkmark \\ p'\end{matrix}, \sigma' \right\rangle} \quad \textbf{(pe)}\ \frac{\langle p, \sigma\rangle \xrightarrow{\beta,\rho} \left\langle \begin{matrix}\checkmark \\ p'\end{matrix}, \sigma' \right\rangle}{\langle A, \sigma\rangle \xrightarrow{\beta[{}^{P \triangleleft A}/_{P}],\rho} \left\langle \begin{matrix}\checkmark \\ p'[{}^{P \triangleleft A}/_{P}]\end{matrix}, \sigma' \right\rangle}\ A = p \in \mathcal{S}$$

perform an action, iff p *can* terminate but does *not perform* the action yet. $p \mathbin{|\!|\!|} q$ states that q performs action (β', ρ') such that p stays allowed to successfully terminate by performing action (β, ρ). To ensure continuation of p after the action taken by q in **(SS2)**, the premise $\langle p, \sigma'' \rangle \xrightarrow{\beta,\rho} \langle \checkmark, \sigma''' \rangle$ is added. □

Left Merge Composition. Table 2 **(SS\$)** defines the left merge composition. The process on the left of the operator has to perform an action first, after which the remaining processes behave concurrently. Note that the concurrency used here is less restrictive than the $\perp\!\!\!\perp$ operator.

Decision 22. The upper case of **(SS\$)** expresses that if p successfully terminates in $p \mathbin{|\!\lfloor} q$ the process behaves as q (no remainder of p can interleave). If p continues as p', the lower case of **(SS\$)** expresses that the remaining process behaves as $p' \rfloor \varnothing \lfloor q$. To allow reuse, we introduce $p' \rfloor \varnothing \lfloor q$, which takes the tasks that need to synchronize as a parameter. As no tasks need to synchronize the parameter is set to $\varnothing$. This auxiliary operator is explained in detail in Section 4.3. □

Conditional Choice. The conditional choice selects a process term according to the outcome of an evaluation function as defined in Table 2 **(C)**.

Decision 23. Let $d : \Sigma \to \mathbb{N}$ be this surjective function that, provided a state vector σ, will return a value within the domain of the enumeration (which is a subset of $\mathbb{N}$). The outcome of $d(\sigma)$ is forced to be in range by the function. □

Synchronized Parallel Composition. The semantics for synchronized parallel composition is given in Table 2 **(spc)**. If behavior occurs on both sides of the operator *and* it is enabled, then execution is synchronized. If behavior occurs on only one side, it must execute without synchronization.

Decision 24. As terms are rewritten on both sides of the operator, the set of synchronizing actions needs to be calculated prior to executing any action. The set needs to be preserved until the synchronized parallel composition successfully terminates. For this we use an auxiliary concurrency operator, that is the same operator as the preemptive sequential operator, though instantiated differently. The concurrent execution operator initiates the auxiliary concurrency operator $p \rfloor C \lfloor q$, where it computes $C \subseteq \mathcal{T} \times \mathbb{N} \times \mathit{List}(\mathcal{S})$, being the set of synchronizing actions that occur in both p and q. □

Decision 25. To compute C in we introduce function *sync* that computes the intersection of transition labels both occurring p and q by $\mathit{sync}(p) \cap \mathit{sync}(q)$. We interpret a transition label $\beta \equiv t_x^{i,P}$ as a triple $(t_x, i, P) \in \mathcal{T} \times \mathbb{N} \times \mathit{List}(\mathcal{S})$. The *sync* function is defined as

$$\begin{array}{ll}
sync(\tau) & = \varnothing \\
sync(t_\alpha^{i,P}) & = \{(t_\alpha, i, P)\} \cup sync(t_\omega^{i,P}) \\
sync(t_\omega^{i,P}) & = \{(t_\omega, i, P)\} \\
sync(p \cdot q) & = sync(p) \cup sync(q) \\
sync(\bigvee_d \langle p_1, \ldots p_n \rangle) & = \bigcup_{i=1}^{n} sync(p_i) \\
sync(p \mathbin{|||} q) & = sync(p) \cup sync(q) \\
sync(p \mathbin{\lfloor\!\lfloor} q) & = sync(p) \cup sync(q) \\
sync(p \mathbin{\perp\!\!\!\perp} q) & = sync(p) \cup sync(q) \\
sync(A) & = sync(p') \text{ where } A = p \in \mathcal{S} \text{ and } p' \text{ is obtained by} \\
 & \qquad \text{substituting all labels } P \text{ by } P \lhd A \text{ in } p
\end{array}$$

□

Process Definition. Table 2 **(pe)** states the deduction rule. For each task in a process, we generate an unique identifier by taking the list of identifier equations (the scope in which an action is executed) and combine it with the task's identifier. The generation of such an identifier is done at the semantic level by substituting the hierarchical levels in tasks.

Decision 26. We substitute by taking the current hierarchical level P and append the identifier's equation $P \lhd A$, and perform it on the action as well as the remaining process term. As an example, if we evaluate term $D = a^i$ at hierarchy at level c, which claims no resources, and observe transition $a^{i,[c \lhd D]}, \varnothing$. □

4.3 Auxiliary Operational Semantics

Concurrent Execution. Concurrent execution $p \rfloor C \lfloor q$ only synchronizes behavior, if an action β occurs in C and both p and q have the action enabled. Otherwise, if β does not occur in C, enabled actions from both p and q are performed without synchronization.

Consider action $\beta \notin C$ and p or q having action β enabled. Table 4 **(spe5)** and Table 4 **(spe6)** define that if either p or q successfully terminates $p \rfloor C \lfloor q$, respectively continues as $C \lfloor p$ or $C \lfloor q$. Table 4 **(spe7)** and Table 4 **(spe8)** define that if either p or q continues as p' or q' respectively, $p \rfloor C \lfloor q$ continues as $p' \rfloor C \lfloor q$ or $p \rfloor C \lfloor q'$ respectively.

If action $\beta \in C$, then Table 4 **(spe1)** states that if p and q can perform an action β, and both end up in a terminating state, then $p \rfloor C \lfloor q$ ends up in a terminating state after executing β. Table 4 **(spe2)** and Table 4 **(spe3)** state that if either p or q ends up in a terminating stating and both processes perform an action β, they continue as a right synchronized execution $C \lfloor p'$ or $C \lfloor q'$, respectively. Table 4 **(spe4)** states that if p and q both have action β enabled and continue as p' and q', $p \rfloor C \lfloor q$ continues as $p' \rfloor C \lfloor q'$. In all cases C is constant.

Decision 27. Rules **(spe2)** and **(spe3)** dictate encapsulation ($\lfloor$), which is undefined within the current semantics. Therefore we again require an auxiliary operator and semantic rules. □

Table 3. Operational rules for auxiliary operators

$$\textbf{(spe1)}\ \frac{\beta \in C, \langle p,\sigma\rangle \xrightarrow{\beta,\rho} \langle \checkmark,\sigma'\rangle, \langle q,\sigma\rangle \xrightarrow{\beta,\rho} \langle \checkmark,\sigma'\rangle}{\langle p \rfloor C \lfloor q,\sigma\rangle \xrightarrow{\beta,\rho} \langle \checkmark,\sigma'\rangle} \qquad \textbf{(spe2)}\ \frac{\beta \in C, \langle p,\sigma\rangle \xrightarrow{\beta,\rho} \langle p',\sigma'\rangle, \langle q,\sigma\rangle \xrightarrow{\beta,\rho} \langle \checkmark,\sigma'\rangle}{\langle p \rfloor C \lfloor q,\sigma\rangle \xrightarrow{\beta,\rho} \langle C \lfloor p',\sigma'\rangle}$$

$$\textbf{(spe3)}\ \frac{\beta \in C, \langle p,\sigma\rangle \xrightarrow{\beta,\rho} \langle \checkmark,\sigma'\rangle, \langle q,\sigma\rangle \xrightarrow{\beta,\rho} \langle q',\sigma'\rangle}{\langle p \rfloor C \lfloor q,\sigma\rangle \xrightarrow{\beta,\rho} \langle C \lfloor q',\sigma'\rangle} \qquad \textbf{(spe4)}\ \frac{\beta \in C, \langle p,\sigma\rangle \xrightarrow{\beta,\rho} \langle p',\sigma'\rangle, \langle q,\sigma\rangle \xrightarrow{\beta,\rho} \langle q',\sigma'\rangle}{\langle p \rfloor C \lfloor q,\sigma\rangle \xrightarrow{\beta,\rho} \langle p' \rfloor C \lfloor q',\sigma'\rangle}$$

$$\textbf{(spe5)}\ \frac{\beta \notin C, \langle p,\sigma\rangle \xrightarrow{\beta,\rho} \langle \checkmark,\sigma'\rangle}{\langle p \rfloor C \lfloor q,\sigma\rangle \xrightarrow{\beta,\rho} \langle C \lfloor q,\sigma'\rangle} \qquad \textbf{(spe6)}\ \frac{\beta \notin C, \langle q,\sigma\rangle \xrightarrow{\beta,\rho} \langle \checkmark,\sigma'\rangle}{\langle p \rfloor C \lfloor q,\sigma\rangle \xrightarrow{\beta,\rho} \langle C \lfloor p,\sigma'\rangle}$$

$$\textbf{(spe7)}\ \frac{\beta \notin C, \langle p,\sigma\rangle \xrightarrow{\beta,\rho} \langle p',\sigma'\rangle}{\langle p \rfloor C \lfloor q,\sigma\rangle \xrightarrow{\beta,\rho} \langle p' \rfloor C \lfloor q,\sigma'\rangle} \qquad \textbf{(spe8)}\ \frac{\beta \notin C, \langle q,\sigma\rangle \xrightarrow{\beta,\rho} \langle q',\sigma'\rangle}{\langle p \rfloor C \lfloor q,\sigma\rangle \xrightarrow{\beta,\rho} \langle p \rfloor C \lfloor q',\sigma'\rangle}$$

$$\textbf{(encap)}\ \frac{\langle p,\sigma\rangle \xrightarrow{\beta,\rho} \left\langle {\checkmark \atop p'},\sigma'\right\rangle, \beta \notin C}{\langle C \lfloor p,\sigma\rangle \xrightarrow{\beta,\rho} \left\langle {\checkmark \atop C \lfloor p'},\sigma'\right\rangle}$$

Encapsulation Operator. The encapsulation operator $C \lfloor p$, prohibits the execution of all actions that occur in C. The semantics is provided in Table 4 **(encap)**, where the successfully termination of $C \lfloor p$ is denoted in the upper case, and the continuation of $C \lfloor p$ as $C \lfloor p'$ in the lower case. In all, we extend our formal abstract syntax as shown in Table 4.

Table 4. Definition of process term $p_{\text{semantics}}$

process term	operator name	variable	description
$p_{\text{semantics}} ::= p \rfloor C \lfloor p$	concurrent execution	C	set of actions
$C \lfloor p$	encapsulation operator	C	set of actions

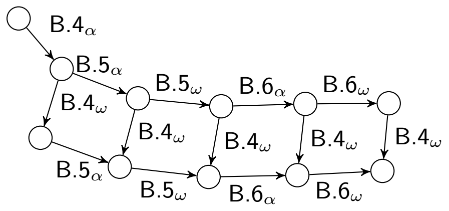
Decision 28. Extends Decision 25 to accommodate the introduction auxiliary operational semantics by adding

$$\begin{aligned} sync(C \lfloor p) &= C \cup sync(p) \\ sync(p \rfloor C \lfloor q) &= C \cup sync(p) \cup sync(q) \end{aligned}$$

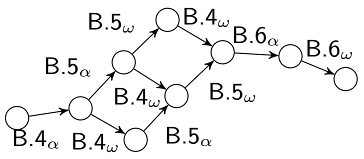
□

4.4 Formal Semantic Validation

To validate DSL's formal semantics, we have proposed a framework [26] that transforms SOS deduction rules, along with the syntactic interpretation, into a specification suitable for formal behavioral analysis. Using an implementation of this framework in the mCRL2 tool-set [13], we can automatically generate state spaces for TRECS models. The generated state spaces can be visualized and used to validate the possible execution behavior for Figure 4a and Figure 4b. The concrete state spaces as generated by the tool-set are visualized using Figure 5a and Figure 5b. The total state space for our Tiramisu example can be generated and visualized in a similar fashion.



(a) Intended behavior



(b) Observed behavior

5 Evaluation

This paper illustrates a structured approach to the formalization of (dynamic) semantics for an industrial DSL using SOS.

We started from an existing DSL with informal and implicit semantics. First, we have identified the concrete notions used in the concrete models. This results in a structuring of the concrete syntax. Furthermore, it facilitates the generalization of concrete syntax elements and syntax variation points. These observations enable multiple concrete syntax projections in the near future.

Once identified, concrete notions are then projected onto abstract notions where the concrete syntax is mimicked as closely as possible. By starting with the most elementary notions first, we try to reuse abstract notations where possible. If reuse is not possible we try to refine existing abstract notions. We introduce new abstract notions when refinement is not possible. This approach helps to create a compositional language. However, it also results in (many) orthogonal annotations, such as the start for a task, process scopes, and task identifiers. These annotations are required to obtain observable and uniquely distinguishable actions for the formal semantics.

Currently, the defined formal abstract syntax for TRECS contains well over thirty abstract notions. Next, we involved engineers to further mature this formal abstract syntax. While maturing, we identified a number of notions where the engineer's *intended* behavior differed from that *implemented* in the interpreter. Including the non-disclosed parts of the DSL, we have identified five semantic gaps, two of which are discussed and addressed in this paper. Most gaps were introduced when formalizing the operators to represent subplans. To close the cognitive gap between intended and implemented semantics, we needed to introduce additional complementary operators.

Next, we defined the formal dynamic semantics for TRECS. That is, for each abstract notion we created one or more SOS deduction rules. Because SOS is a compositional formalism, it facilitates an incremental approach where the behavior of simple notions can be composed into more complex, compound behavior. As such, we expected that the semantics could be defined by more simple notations. Instead, as the semantics are subjected to numerous design decisions, we had to introduce auxiliary operators to *exactly* capture the semantics.

In the formal semantics, "available resources" (R_A) could replace the state vector (σ), since all evaluations of the example are preformed on R_A. We decided to explicitly define σ since the full DSL contains other constructs that also manipulate σ and influence the decision taking process. Moreover, we choose to define resource claims using total mappings (visible on the transition label) meaning that any and all resource labels need to be known in advance. Finally, we want to stress that TRECS allows to fork and join concurrency in an almost arbitrary manner. In turn, this implied (lots of) refinement of notions to obtain unique task labels and ensure correct synchronization.

Once defined, our new operators have been implemented manually in the DSL's interpreter. For each use of a legacy operator, domain experts have to decide to either retain the legacy operator or to switch to the new operator based on the disambiguated semantics. This approach provides backward compatibility with the (execution behavior) of the informal language. Note that the use of complementary operators reduced the regression and qualification impact significantly while phasing out ambiguous behavior.

6 Conclusions and Future Work

A DSL's syntax is typically defined through some (parsing) grammar or metamodel. In practice, the execution semantics and a DSL's model of computation are mostly implemented implicitly in translations and/or an execution engine or interpreter. We observed that the execution behavior of concrete domain models may exhibit unexpected, critical dependencies on this implicit semantics. Formalizing the syntax and defining the semantics for a DSL is a challenging task, particularly when considering the operational impact of changing the existing execution semantics.

This paper demonstrates the successful formalization of an industrial size DSL. We have defined the DSL's operational semantics though SOS, which is fully compositional. From thereon, we can aggregate and compose terms and study the behavior of these composed terms in isolation. The result is a language definition where the DSL's abstract and concrete syntax are formally related to its static and dynamic semantics. As our approach results in orthogonal concepts, it provides handles to analyze different aspects (e.g. throughput and safety) that are closely related to the execution behavior.

Using our bottom-up approach, we obtain a well-defined behavioral scope, which worked particularly well when validating the formal semantics with domain experts and language engineers. However, this validation can also uncover cognitive gaps between the engineer's intended semantics and the DSL's implemented semantics. To bridge these gaps, we propose to add complementary operators to disambiguate. The use of separate operators limits the regression and qualification impact in an operational context. Also, it facilitates the migration of legacy concrete models to the new formalized DSL. To facilitate automated analysis of models created in our formalized DSL, we have developed a generic mathematical framework [26] that also accepts the work presented in this paper directly as input.

The work conducted in this paper was part of the KWR 09124 project LithoSysSL at ASML. Currently, we are extending our work with formal semantics for non-disclosed reactive concepts and run-time optimization rules. Also, the possibility to define SOS on meta-models in MDE environments is investigated.

References

1. Aredo, D.B.: A Framework for Semantics of UML Sequence Diagrams in PVS. Journal of Universal Computer Science 8(7), 674–697 (2002)
2. Baeten, J.C.M., Basten, T., Reniers, M.A.: Process Algebra: Equational Theories of Communicating Processes (Cambridge Tracts in Theoretical Computer Science). Cambridge University Press (December 2009)
3. Bol, R.N., Groote, J.F.: The Meaning of Negative Premises in Transition System Specifications. J. ACM 43(5), 863–914 (1996)
4. Börger, E.: High Level System Design and Analysis Using Abstract State Machines. In: Hutter, D., Traverso, P. (eds.) FM-Trends 1998. LNCS, vol. 1641, pp. 1–43. Springer, Heidelberg (1999)
5. Börger, E., Cavarra, A., Riccobene, E.: An ASM Semantics for UML Activity Diagrams. In: Rus, T. (ed.) AMAST 2000. LNCS, vol. 1816, pp. 293–308. Springer, Heidelberg (2000)
6. Cabot, J., Clarisó, R., Riera, D.: Verification of UML/OCL Class Diagrams using Constraint Programming. In: ICSTW 2008, pp. 73–80. IEEE Computer Society (2008)
7. Clark, T., Warmer, J. (eds.): Object Modeling with the OCL. LNCS, vol. 2263. Springer, Heidelberg (2002)
8. Combemale, B., Crégut, X., Garoche, P.-L., Thirioux, X.: Essay on Semantics Definition in MDE - An Instrumented Approach for Model Verification. JSW 4(9), 943–958 (2009)
9. David, A., Möller, M.O., Yi, W.: Formal Verification of UML Statecharts with Real-Time Extensions. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 218–232. Springer, Heidelberg (2002)
10. Di Ruscio, D., Jouault, F., Kurtev, I., Bézivin, J., Pierantonio, A.: Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs. Technical Report n. 06.02, Laboratoire d'Informatique de Nantes-Atlantique (April 2006)
11. Evermann, J., Wand, Y.: Toward Formalizing Domain Modeling Semantics in Language Syntax. IEEE Trans. Software Eng. 31(1), 21–37 (2005)
12. Graaf, B., Weber, S., van Deursen, A.: Model-Driven Migration of Supervisory Machine Control Architectures. JSS 81(4), 517–535 (2008)
13. Groote, J.F., Mathijssen, A.J.H., Reniers, M.A., Usenko, Y.S., van Weerdenburg, M.J.: The Formal Specification Language mCRL2. In: MMOSS. Dagstuhl Seminar Proceedings, vol. 06351. IBFI, Schloss Dagstuhl, Germany (2007)
14. Horn, A.: On Sentences Which are True of Direct Unions of Algebras. J. Symb. Log. 16(1), 14–21 (1951)
15. IEEE Standard for the Scheme Programming Language. IEEE Std 1178-1990 (1991)
16. Jackson, E.K., Schulte, W.: Model Generation for Horn Logic with Stratified Negation. In: Suzuki, K., Higashino, T., Yasumoto, K., El-Fakih, K. (eds.) FORTE 2008. LNCS, vol. 5048, pp. 1–20. Springer, Heidelberg (2008)
17. Jackson, E.K., Sztipanovits, J.: Formalizing the structural semantics of domain-specific modeling languages. Software and System Modeling 8(4), 451–478 (2009)

18. Jansamak, S., Surarerks, A.: Formalization of UML statechart models using Concurrent Regular Expressions. In: ACSC 2004, pp. 83–88. Australian Computer Society, Inc., Darlinghurst (2004)
19. Kleppe, A.: Software language engineering. Addisson-Wesley (2009)
20. Kuske, S.: A Formal Semantics of UML State Machines Based on Structured Graph Transformation. In: Gogolla, M., Kobryn, C. (eds.) UML 2001. LNCS, vol. 2185, pp. 241–256. Springer, Heidelberg (2001)
21. Mauw, S., Wiersma, W.T., Willemse, T.J.H.: Language-Driven System Design. IJSEKE 14(6), 625–663 (2004)
22. Lilius, J., Paltor, I.P.: Formalising UML State Machines for Model Checking. In: France, R.B. (ed.) UML 1999. LNCS, vol. 1723, pp. 430–444. Springer, Heidelberg (1999)
23. Plotkin, G.D.: A Structural Approach to Operational Semantics. J. Log. Algebr. Program. 60-61, 17–139 (2004)
24. Rumbaugh, J., Jacobson, I., Booch, G.: Unified Modeling Language Reference Manual, 2nd edn. Pearson Higher Education (2004)
25. Sadilek, D.A., Wachsmuth, G.: Using Grammarware Languages to Define Operational Semantics of Modelled Languages. In: Oriol, M., Meyer, B. (eds.) TOOLS EUROPE 2009. LNBIP, vol. 33, pp. 348–356. Springer, Heidelberg (2009)
26. Stappers, F.P.M., Reniers, M.A., Weber, S.: Transforming SOS Specifications to Linear Processes. In: Salaün, G., Schätz, B. (eds.) FMICS 2011. LNCS, vol. 6959, pp. 196–211. Springer, Heidelberg (2011)
27. Tonino, H.: A Sound and Complete SOS-Semantics for Non-Distributed Deterministic Abstract State Machines. In: Workshop on Abstract State Machines, pp. 91–110 (1998)
28. van Beek, D.A., Reniers, M.A., Schiffelers, R.R.H., Rooda, J.E.: Foundations of a Compositional Interchange Format for Hybrid Systems. In: Bemporad, A., Bicchi, A., Buttazzo, G. (eds.) HSCC 2007. LNCS, vol. 4416, pp. 587–600. Springer, Heidelberg (2007)
29. van den Nieuwelaar, N.J.M.: Supervisory Machine Control by Predictive-reactive Scheduling. PhD thesis, Technische University Eindhoven (2004)
30. Wielemaker, J.: An Overview of the SWI-Prolog Programming Environment. In: WLPE. Report, vol. CW371, pp. 1–16. Katholieke Universiteit Leuven (2003)
31. Wolterink, T.J.L.: Operational Semantics Applied to Model Driven Engineering. Master's thesis, University of Twente (2009)

Semantics First!
Rethinking the Language Design Process[⋆]

Martin Erwig and Eric Walkingshaw

School of EECS
Oregon State University

Abstract. The design of languages is still more of an art than an engineering discipline. Although recently tools have been put forward to support the language design process, such as language workbenches, these have mostly focused on a syntactic view of languages. While these tools are quite helpful for the development of parsers and editors, they provide little support for the underlying design of the languages. In this paper we illustrate how to support the design of languages by focusing on their semantics first. Specifically, we will show that powerful and general language operators can be employed to adapt and grow sophisticated languages out of simple semantics concepts. We use Haskell as a metalanguage and will associate generic language concepts, such as semantics domains, with Haskell-specific ones, such as data types. We do this in a way that clearly distinguishes our approach to language design from the traditional syntax-oriented one. This will reveal some unexpected correlations, such as viewing type classes as language multipliers. We illustrate the viability of our approach with several real-world examples.

1 Introduction

How do we go about designing a new language? This seems to still be an open question. To quote Martin Fowler from his latest book [6, p. 42]:

> *When people reviewed this book, they often asked for tips on creating a good design for the language. ... I'd love to have a [sic] good advice to share, but I confess I don't have a clear idea in my mind.*

This quote underlines that even though the development of software languages is supported by quite a few tools, the design process itself is still far from being an engineering discipline. Many concepts remain poorly defined or are interpreted differently depending on the approach taken.

In this paper we will address this problem and present a systematic approach to designing a software language. This approach is based on two key ideas or insights.

First, language development should be *semantics driven*, that is, we start with a semantics model of the language core and then work backwards to define a more and more complete language syntax. This is a rather unorthodox, maybe even heretical, position given the current dogma of programming language specification. For example, the very first sentence in Felleisen et al.'s latest book [5] states rather categorically:

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

The specification of a programming language starts with its syntax.

We challenge this view and argue that, even though the "syntax first" approach has a long and well established tradition, it is actually impeding the design of languages.

Second, language development should be *compositional*, that is, bigger languages should be composed of smaller ones using well-defined language *composition operators*. The notion of compositionality itself is widely embraced and praised as a mark of quality, particularly in the area of denotational semantics [15], and compositionality within individual languages is generally valued since it supports expressiveness with few language constructs. We will illustrate that a semantics-driven language design will itself be compositional in nature and will lead more naturally to compositional languages, in particular, when compared to syntax-driven language design. One reason might be that thinking about a language's syntax is often tied to its concrete syntax, which is problematic since the widely used LL or LR parsing frameworks are *not* compositional in general and thus impose limits on the composition of languages [11].

In order to motivate and explain our approach we will consider in Section 3 the (evolving) design of a small language for a calendar application. This example will help us establish a set of basic concepts and the corresponding terminology. This example also helps to point out some of the challenges that a language developer is faced with.

To discuss the involved technical aspects we have to express the language design in a concrete (meta)language. We use Haskell for this purpose since (1) Haskell has been successfully employed in the development of many DSLs, and (2) many of Haskell's concepts have a direct and clear interpretation in terms of language design. We will briefly summarize how Haskell abstractions map to language concepts in Section 2.

Using Haskell as a metalanguage (or DSL) for language design also allows us to identify new concepts in language design. One example is the notion of *language schema*. Language schemas and their cousins *language families* will be discussed in Section 4 where we specifically point out how polymorphism in the language description formalism (that is, the metalanguage) can be exploited for making language design more systematic and amenable to reuse.

A critical aspect of semantics-driven language design is the systematic, incremental extension of a base language (or schema) to a more complex language. This process is supported by language operators that are discussed in Section 5. As explained in Section 2, the semantics-driven approach leads to a distribution of language descriptions across different concepts of the metalanguage. This suggests a distinction of language operators into different categories, and the description follows these categories.

In Section 6 we will demonstrate the proposed semantics-driven language design approach on several examples to illustrate its power and simplicity. Finally, after a discussion of related work in Section 7 we present some conclusions in Section 8.

2 Haskell as a Language Design DSL

Haskell [16] has a long tradition as a metalanguage and has been used quite extensively to define all kinds of domain-specific languages. A few standard idioms of how to represent languages in Haskell have developed that are part of the Haskell folklore. Even though these idioms may not have been documented comprehensively in one place,

Language Domain		*Metalanguage (Haskell)*	
Language	L	Data type	`data L =` $\texttt{C}^n$
L. Schema	-	Type constructor	`data S a =` $\texttt{C}^n$
Program	$p \in L$	Expression	`e :: L`
Operation	$N_1 \ldots N_k \to L$	Constructor	`C ::` $\texttt{N}^k$ `-> L`
Semantics domain	D	(Data) type	`data D`
Semantics	$[\![\cdot]\!] : L \to D$	Function	`sem :: L -> D`

Fig. 1. Syntax-directed view of language concepts and their representation

Tim Sheard's paper [18] is a good place to start. There has also been some work on how to make language design modular and extensible [8, 13, 18, 19].

Most of this work is focused on syntax, taking the view that a language is defined by defining its (abstract) syntax plus a mapping to some kind of semantics domain. Under this view of language, sentences expanded from a nonterminal `L` are represented in Haskell by terms that are built using constructors of a data type `L`. Each constructor represents a grammar production for `L`. The argument types of the constructor represent the nonterminals that occur on the right-hand side of the production. Constants represent terminal symbols, and constructors having basic type arguments (such as `Int`) form the link to the lexical syntax. This view is briefly summarized in Figure 1.

The two-level view that results from the syntax-directed approach to language design is not the only way in which language can be represented, however. Alternatively, we can start the design of a language with a decision about the semantics domain that best captures the essence of the domain the language is describing. In Haskell this domain will also be represented as a (data) type, say `D`. However, its constructors will be taken immediately as language operators. For example, in a language for representing dates (see also Figure 3) we may have a data type `Month` that includes constructors such as `Jan`. This constructor would not just be considered a semantics value, but would also be used as an operation of the date language.

Of course, the language will need more operations than just the constructors of `D`. Instead of introducing an explicit representation through additional data types or constructors, as in the syntax-directed approach, the semantics-directed approach will simply define functions that take the appropriate arguments and construct elements of the semantics domain `D` directly. This idea is also behind the combinator library approach to the design of domain-specific embedded languages (DSELs), see, for example, [8].

While a semantics function is required in the syntactic approach to map explicit syntax to semantics values, in the semantics-directed approach this semantics function is effectively distributed over many individual function definitions. By forgoing an explicit syntax representation, the phase distinction between syntax and semantics disappears in the semantics-directed approach.

The semantics-directed view of language development and its implication on the metalanguage representation are briefly summarized in Figure 2.

The basic idea of semantics-driven language development is to start with a small language that represents the essence of the language to be developed, then to extend this core systematically. The advantages of this approach are: (1) The compositional

Language Domain		*Metalanguage (Haskell)*	
Language	L	Data type & functions	`data D =` $\texttt{C}^n; (\texttt{f = e})^m$
L. Schema	-	Type constructor & functions	`data S a =` $\texttt{C}^n; (\texttt{f = e})^m$
Operation	$N_1 \dots N_k \to L$	Constructor or function	`C/f ::` $\texttt{N}^k$ `-> L`
Semantics domain	D	Given by `D`, the data type part of L	
Semantics	$[\![\cdot]\!] : L \to D$	Given by $(\texttt{f = e})^m$, the function part of L	

Fig. 2. Semantics-directed view of language concepts and their representation

design clearly represents the different components of the language and how they are connected, (2) the language development is less ad hoc, and (3) being compositional, the language design can be better maintained.

These advantages are particularly relevant for language prototyping. Once a designer is happy with a semantics-driven language design, she can always "freeze" it by converting its syntax from implicit into explicit representations and adding a semantics function. This enables the abstract syntax of programs in the language to be manipulated directly (for analysis or transformation), regaining any advantages of syntax-directed approaches. Therefore the semantics-driven approach should not be seen as an exclusive alternative to the syntax-directed approach, but rather as a companion that can in some cases replace it and in other cases precede it and pave the way for a more syntax-focused design.

3 Semantics-Driven Language Development

In this section we will illustrate with an example how a focus on semantics can go a long way in building a language. Specifically, we will consider compositional language extensions, that is, extensions that do not require changes to existing languages, in Section 3.1. We will also discuss the problem and necessity of non-compositional language extensions in Section 3.2.

3.1 Compositional Language Extensions

Consider a calendar tool for storing appointment information. That this is by no means a trivial application domain can be seen by the fact that many different calendar applications and tools exist with different sets of features, and that some calendar functionality is commonly performed by separate, external tools (consider, for example, the planning of schedules with Doodle).

In order to build a calendar application, we must identify the operations that such an application must perform. However, instead of doing this directly, which would lead to a flat, monolithic set of operations, we instead approach the problem by focusing first on the core language elements. We will begin by identifying the individual components of the domain, and how these can be represented by different DSLs. We will then incrementally compose and extend these smaller DSLs to form the desired application. This approach has the additional advantage that it can produce a library of small and medium-sized DSLs that can be reused in the development of many language projects. We also refer to these small, reusable DSLs sometimes as *Micro DSLs*.

```
data Month = Jan | Feb | ... | Dec
type Day = Int
data Date = D Month Day

[jan,...,dec] = map D [Jan,...,Dec]
```

```
type Hour = Int
type Minute = Int
data Time = T Hour Minute

hours h = T h 0
am h = hours h
pm h = hours (h+12)
before t t' =  t'-t
```

Fig. 3. Micro DSLs for date and time. The function `hours` constructs hour values, `am` and `pm` denote morning and afternoon hours, and the function `before` subtracts two time values.

The most important element of semantics-driven language design is the idea to start the design process by identifying the absolutely most essential concepts of the language to be developed. Compared with a denotational semantics approach, this basically amounts to identifying the semantics domain, which also means that the semantics function for this initial core language will then be the identity function.

At its core, a calendar application offers the ability to define appointments at particular times. We recognize two separate components in this description, "times" and "appointments", that we can try to define through individual DSLs. These two languages are linked to form a calendar, and without needing to know any details about times and appointments, we can see that it is this language combination that captures the essence of calendars. Specifically, times are *mapped* to appointments. We can represent this using a generic language combinator for maps that is parameterized by types for its domain and range. This straightforward `Map` data type can be defined as follows.[1]

```
data Map a b = a :-> b | Map a b :&: Map a b
```

Whereas a data type represents a language, a parameterized data type represents what we call a *language schema*, that is, a whole class of languages. A language can be obtained from a language schema through instantiation, that is, by substituting languages for the type parameters.

To make our example more concrete we start with a simple version of a calendar that maps days, given by month and day, to values of some arbitrary type. To this end we make use of the definitions for dates and times shown in Figure 3.[2] Note that even these tiny languages are not completely defined by data types alone. For example, functions, such as `dec` or `pm` are providing syntactic sugar. In addition, the function `before` extends the `Time` language by a new operation.

Based on the languages `Date` and `Time` we can define a language for calendars that associates appointment information with dates.

```
type CalD a = Map Date a

week52 :: CalD String
week52 = dec 30 :-> "Work" :&: dec 31 :-> "Party"
```

[1] Here and in the following we will sometimes omit details, such as `Show` instance definitions and infix declarations. Their effect will become clear from their use.

[2] Again, we simplify the definitions a bit and omit some definitions, such as the `Show` instances or the `Num` instance for `Time`.

Strictly speaking, `CalD` is still a language schema since the appointment information has not yet been fixed, but `week52` is a program in the language `CalD String`. It is obvious that we can also define a calendar language that maps `Time` to appointments. In the following example we define a simple *calendar pattern* to encode a habit to exercise before a party. With that, we can then define two typical daily schedules.

```
type CalT a = Map Time a

partyAt :: Hour -> CalT String
partyAt h = hours 2 ‘before‘ h :-> "Exercise" :&: h :-> "Party"

work, party :: CalT String
work = am 8 :-> "Work" :&: pm 6 :-> "Dinner"
party = work :&: partyAt 9
```

The use of calendar patterns supports a very high-level and compositional description of calendars without changing the underlying language representation. For example, a `party` day expands to a time calendar value as follows.[3]

```
08:00 -> "Work" & 18:00 -> "Dinner" & 19:00 -> "Exercise" & 21:00 -> "Party"
```

We argue that this form of low-cost extensibility is, at least in part, a direct consequence of choosing the most appropriate semantics domain, in this case a mapping. Therefore, the initial focus on language semantics pays off since it simplifies the later language design by dramatically lowering the language maintenance effort. Since in our view a language is given by the core representation plus additional functions, we can also view function definitions, such as `partyAt`, as user-defined extensions of the DSL.

From a language engineering perspective, we can observe that the function definition capability of the metalanguage helps us to easily and flexibly extend the core representation by new features, such as patterns for dependent appointments.

We can easily combine these two types of calendars by instantiating the date calendar with the time calendar.

```
type Cal a = CalD (CalT a)

week52 :: Cal String
week52 = dec 30 :-> work :&: dec 31 :-> party
```

We have illustrated how to extend the calendar language by refining the *domain* of the mapping structure that forms its semantics basis. In the same way we can create more expressive calendar languages by extending the *range*. Using product types we can combine appointment information with information about participants, how long an appointment takes, dependencies between appointments or other relationships, etc. This is all quite straightforward, and the result can be as general as the requirements of a particular application need it to be. This extensibility is a consequence of finding the right semantics domain for calendars at the beginning of the language design process.

[3] Note that when pretty printed, the data constructors `:&:` and `:->` are rendered as `&` and `->`.

Of course, there are situations when the initial design decision is not general enough to support a specific language extension. In these cases, we have to resort to non-compositional changes to the language. This is what we look into next.

3.2 Non-compositional Language Extensions

Suppose we want to extend the calendar application by allowing the distinction between publicly visible and private (parts of) appointments (we might, for example, want to hide the fact that we have two parties on two consecutive days). This idea can be easily extended to more sophisticated forms of visibility or visibility in particular contexts. From a language perspective we are faced with the need to selectively annotate parts of an abstract syntax tree. Since this situation is quite common, the approach to take from a language composition perspective is to define a generic annotation language (that is, a language schema) and integrate this in some way with the language schema `Cal`. We begin by defining a simple language schema for marking terms as private. This could be easily generalized to a more general annotation language by additionally parameterizing over the annotation language, but we will pursue this less-general approach for clarity.

```
data Privacy k a = Hidden k a | Public a
```

A simple extension of the calendar language with this privacy language is obtained by *composing* the language schemas `Map` and `Privacy k` (for some language of keys `k`) when instantiating `Cal`. Since language schemas are represented by type constructors in the metalanguage, language composition is realized by type instantiation.

```
type Key = String
type Private a = Privacy Key a
type CalP a = Map (Private Date) (Private a)
```

We also add some special syntax for the map constructor for different combinations of hidden and visible information. We use `*` and `.` in the smart constructors to indicate the position of the hidden and publicly visible information, respectively. (We omit the definition of `*->*` since we don't need it for our examples.)

```
(*->.) :: (Key,Date) -> a -> CalP a
(k,d) *->. i = Hidden k d :-> Public i

(.->*) :: Date -> (Key,a) -> CalP a
d .->* (k,i) = Public d :-> Hidden k i

(.->.) :: Date -> a -> CalP a
d .->. i = Public d :-> Public i
```

We can now hide data and/or appointment information in calendars (for example, to hide our birthday on New Year's Eve or that we have a party on December 30th).

```
week52 = ("pwd",dec 30) *->. "Party" :&: dec 31 .->* ("pwd","Birthday")
```

When we inspect a partially hidden calendar, the pretty printer definition for `Privacy` ensures that hidden parts will be blocked out.

```
*** -> "Party" & Dec-31 -> ***
```

So far the privacy extension of calendars was compositional. However the extension is limited. While the shown definition enables us to selectively hide information about particular appointments, it does *not* allow us to hide whole sub-calendars. This could be important because we might not want to expose the number of appointments of some part of our calendar to an outside party, but with the current definition we can only hide the leaves of the syntax tree, and the number of entries remains visible.

Note that simply wrapping `Private` around `CalP` doesn't solve this problem, because the `:&:` operation expects arguments of type `Map` and thus can't be used to compose private calendars. One could envision the definition of a smart constructor for `Map`, a function that inspects the calendar arguments and then propagates the privacy status to the combined calendar, but this approach will inherently lose the privacy information of subcalendars and thus doesn't solve the problem.

A solution to this problem is to generalize the definition of `Map` to allow for an additional language schema as a parameter, which is then used to wrap the result of recursive occurrences of `Map` in `:&:` and the arguments of `:->`. Such a generalization of `Map` itself is not compositional, but after the generalization we have regained compositionality, which allows us to continue to keep the privacy and other micro DSLs separated.

There are different ways to realize this idea. The most obvious approach is to directly apply the type constructor representing the language schema to every occurrence of `Map`.

```
data Map w a b = w a :-> w b | w (Map w a b) :&: w (Map w a b)
```

However, this representation might cause a lot of unnecessary overhead, in particular, in cases when local calendar annotations are only sparingly used. Moreover, from a more general language maintenance perspective, this approach is often more involved since one has to change *all* recursive occurrences. This might cause more work in more complicated data types, which also complicates the adaptation of values to the new types. A less intrusive approach is to add an additional constructor to `Map` which wraps just one recursive occurrence of `Map`. This constructor can then be used on demand and thus introduces the wrapping overhead only when needed.

```
data Map w a b = w a :-> w b
               | Map w a b :&: Map w a b
               | Wrap (w (Map w a b))
```

With this definition we can apply the privacy operations not only to dates and infos, but also to whole subcalendars.

```
week1 = Wrap $ Hidden "pwd" (jan 1 .->. "Party" :&: jan 2 .->. "Rest")
```

Evaluating `week52 :&: week1` produces the following output, completely hiding `week1`.

```
*** -> "Party" & Dec-31 -> *** & ***
```

The calendar scenario demonstrates how languages can be developed in small increments, starting from a small initial semantics core. We have seen that ideally language extensions are performed in a compositional way, but that this is not always possible. In the following two sections we will first briefly discuss the notions of language schemas and language families and then analyze language operators that form the basis of our approach to grow and combine languages out of small micro DSLs.

4 Language Schemas and Families

Sets of (related) languages can be characterized by a *language schema*, that is, a parameterized data type. We have seen different forms of calendars represented in this way, and all calendars are elements of the set of languages characterized by the schema `Map`.

Language schemas facilitate the definition of quite general language operators that can work on whole classes of languages. As an example, consider the function `dom` that computes the domain in the form of a list of values for any language captured by the language schema `Map`. In the calendar language `dom` computes the times at which appointments are scheduled, whereas in a scheduling or voting application (such as Doodle), where `Map` may be used to map users to their votes or preferences, `dom` computes users that have (already) voted. We can thus see that different concepts in different languages are realized by the same polymorphic function, which is made possible since the function is tied to a language schema that can be instantiated in many different ways.

Some language schemas will be the result of instantiation from more general language schemas. We have seen several examples of this, such as `CalT`, which is an instance of `Map`, and `Cal` which is a "nested instance" obtained by instantiating `CalD` (which is already an instance) by `CalT`, which is another instance.

Language schemas capture the idea of fully parameterized, or fully polymorphic, languages, represented by parametric polymorphism in data types. The generality of language schemas is a result of the data type polymorphism.

Language families are groups of related languages and are represented by type classes. Languages are related if they have common operations (methods). An important use of type classes in compositional language design is to enforce constraints on the languages that can be used in a language schema. For example, we might say that any language `w` used in the extended `Map` schema must provide an operation `unwrap`.

Type classes fit a bit differently into the "language operator" view, as will be explained below. Type classes reveal an interesting new class of activities in language design, something that could be called *language organization*. For example, creating a type class, say `F`, does *not* create a new language directly, but it provides *new opportunities* for creating new languages. This typically happens when we make a type (that is, language) `L` an instance (that is, member) of the type class (that is, language family) `F`. In that case all the functions that are derived from the type class become automatically available for the new instance. In other words, the instantiation has added new syntax (represented by the derived functions) to the language.

5 Language Operators

In our vision of semantics-driven, compositional language development, languages live in a space in which they are connected by language operators. This structure allows a language designer to start a design with some initial language and then traverse the space by following language operators until a desired language is reached. In this section we discuss the notion of *language operators*, which transform languages into one another. Specifically, we are interested in language operators for expanding languages since the semantics-driven approach to language design builds more complex languages out of simpler ones. Therefore, we will focus on expansion operators and only briefly mention their inverse cousins for language shrinking.

***First-Order Operations**: Adding/Removing …*

In the language domain	*In the metalanguage (Haskell)*	
… (Sub)language	Data type	$\bullet\langle\texttt{data L ps = CS}\rangle$
… Operation	Constructor	$\texttt{data L ps = CS}\bullet\texttt{C}$
… Operation argument	Constructor argument	$\texttt{data L ps = CS}\{\texttt{C TS}\bullet\texttt{T}\}$

Higher-Order Operation

In the language domain	*In the metalanguage (Haskell)*	
Abstraction	Type parameterization	$\texttt{data L ps}\oplus\texttt{a = }[\texttt{a/T}]\texttt{CS}$
Instantiation	Type instantiation	$\oplus\langle\texttt{type L = S T}\rangle$
Inheritance	Type class instantiation	$\oplus\langle\texttt{instance C L where fs}\rangle$

Fig. 4. Semantics language operators and their representation

In the description of language operators we make use of some auxiliary notation to abbreviate different kinds of changes to a language description. Since, in the context of this paper, a language description is a Haskell program, that is, a set of Haskell type and function declarations, we basically need operations to add, remove, and change such declarations. Thus, we use $\oplus\texttt{D}$ and $\ominus\texttt{D}$ to indicate the addition and removal of a declaration `D` from the language description, respectively. We use $\bullet$ to denote either operation. We also use these operations in the context of declarations to add or remove parts. For example, we write $\texttt{data L = CS}\oplus\texttt{C}$ to express the addition of a constructor `C` to the constructors `CS` of the data type `L`. To pick a single element in a list as a context for a transformation we enclose the element in curly brackets following the list. For example, the notation $\texttt{CS}\{\texttt{C TS}\oplus\texttt{T}\}$ says that the list of argument types `TS` of one constructor `C` in the list of constructors `CS` is extended by the type `T`.

We also make use of the traditional substitution notation $[\texttt{N/O}]\texttt{D}$ for substituting the new item `N` for the existing old item `O` everywhere it occurs in the declaration `D`, and we abbreviate $[\texttt{N/O}]\texttt{O}$ by $[\texttt{N/O}]$. Specifically, we use `D` for declarations, `CS` for lists of constructors, and `C` for individual constructors. We also employ indexing to access parts of specific definitions. For example, $\texttt{CS}_\texttt{L}$ yields the constructors of the data type `L`.

We can distinguish between first- and higher-order language operators. A *first-order language operator* takes one or more languages and produces a new language. In contrast, a *higher-order language operator* takes other language operators as inputs or produces them as outputs. Moreover, we can distinguish language maintenance operations according to the language aspect they affect, that is, whether they affect the *semantics* (representation), the *syntax*, or the *organizational structure*. We will consider first- and higher-order operations for these cases separately in the following subsections.

5.1 Semantics Language Operators

The semantics language operations and their representation in the metalanguage are summarized in Figure 4.

An example of a first-order language operator is the addition of a new operation, represented in the metalanguage by the addition of a constructor to the data type representing the language. Similarly, we can extend an existing language operation by adding a new type argument to the constructor that represents that operation. We can also add whole languages by adding new data types. This will often be a preparatory step to combine the language with others into a bigger language. All of these operations have natural inverse operations, that is, removing productions/constructors, restricting operations/constructors, and removing languages/data types.

These six first-order operations form the basis for other language operations. For example, consider the case when we have two languages `L` and `M` with different operations that are nevertheless describing the same domain. We can merge `L` and `M` into one language, say `L`, by substituting all occurrences of type `M` in the constructors of `M` by `L` and then adding those updated constructors to `L`. Since language `M` is not needed anymore after the merge, it can be removed.

$$\texttt{data L = CS}\oplus\langle[\texttt{L/M}]\texttt{CS}_{\texttt{M}}\rangle$$
$$\ominus\langle\texttt{data M = CS}\rangle$$

This is an example of an (ordered) *union* of two languages (ordered, because one language is privileged since its name is kept as a result of the union).

In contrast to first-order language operators that work directly on languages, a *higher-order language operator* takes other language operators as inputs or produces them as outputs. We should note at this point that a language schema is itself a language operator since it can produce, via instantiation, different languages. With this in mind, we can discuss higher-order language operations. One example is *language abstraction* that takes a language or a language schema and produces a language schema by substituting a type (or sublanguage) by a parameter. Similarly, *language instantiation* takes a language schema and substitutes a language (or language schema) for one of its parameters and thus produces a language or a more specific language schema. For example, `CalD` is obtained from `Map` by substituting `Date` for `a`.

As with first-order language operations, we can derive more sophisticated higher-order language operations from abstraction and instantiation. In the following we discuss one such example, namely *language* or *schema composition*. The basic idea behind schema composition is to instantiate one schema with another. Taking the example from Section 1 we can instantiate a new language schema as follows.

```
type CalP a = Map (Private Date) (Private a)
```

We can then use this specialized schema to instantiate further languages (or schemas).

Finally, we can describe the inheritance of operations from existing languages through the instantiation of type classes, which makes type classes a powerful weapon, because in addition to the class members, all functions that are derived from the class will be made also available for the newly instantiated language. The importance of this language operation cannot be overemphasized. It can extend the scope and expressiveness of a language dramatically with very little effort. We will present an example of this later in Section 6.1.

5.2 Syntax Language Operators

The syntax language operations and their representation in the metalanguage are rather straightforward and are summarized in Figure 5. Interestingly, the syntax level offers

First-Order Operations: *Adding/Removing …*

In the language domain	*In the metalanguage (Haskell)*	
… Operation	Function	•⟨`fun f vs = e`⟩
… Operation argument	Function argument	`fun f vs`•`v = `[`v/e'`]`e`
… Specialized syntax	Function instantiation	•⟨`g = f e`⟩

Fig. 5. Syntax language operators and their representation

only first-order language operations. This might be a reason why the semantics-driven approach is so much more powerful, because it offers higher-order language operations.

Extending a language by introducing new syntax works essentially by adding a new function definition. We have shown examples of this in Figure 3. In addition, in DSEL settings, users can extend language syntax on the fly by adding their own function definitions, as was illustrated in Section 3.1 with the function `partyAt`.

By extending an existing function with a new parameter we can extend the scope of existing operations within a language. For example, we could add a new parameter for minutes to the `pm` function shown in Figure 3 and thus extend the time language. Of course, the inverse operations of removing function definitions or removing function arguments are also available. Moreover, we can add or remove specialized syntax by adding instances of functions obtained through application of more generic functions to specific values. The definition of the reusable calendars `work` and `party` are examples of this, again happening on the user level.

5.3 Organizational Language Operators

The adjective "organizational" indicates that the operators in this group are not directly responsible for extending languages. But that does not mean that they are not useful or even powerless. Organizational operations are preparatory in nature; they are akin to an investment that pays dividend later.

For example, the definition of a type class creates a view of a language, called *language family*, that other languages can be associated with. The benefit of making a language a member of a language family (that is, making the data type an instance of the type class that represents the language family) lies in getting immediate access to all the functions that are derived from the class, that is, in language terms, the syntax of the new family member is at once expanded by the whole "family heritage". An example of this is making a language a member of `Monad`, which expands the language's syntax through all the functions available in the vast monad libraries.

The definition of a language family itself amounts to the definition of a "language multiplier" since the syntax provided by the functions derived from the type class can be repeatedly added to arbitrarily many other languages. Multi-parameter type classes, functional dependencies, and associated types do not change this view in any substantial way. Moreover, most of the machinery that is available for defining type classes, such as subclasses or derived classes, are supporting tools for the definition of language multipliers. Finally, adding a class constraint to a schema/function restricts the languages that that schema can be instantiated with. Adding a class constraint might be considered a higher-order operation since it produces a new (constrained) schema.

6 Semantics-Driven Language Design in Action

The semantics-driven approach to language development is born from our experiences designing many languages for a wide range of application domains. In this section, we discuss the design of just three of these languages from the perspective of semantics-driven design. Each of these languages is described in published papers (one with a best paper award), and one is in active use by other people. The first two strongly exhibit semantics-driven traits as published, while the third language is a more traditional syntax-directed design which we have redesigned here in a semantics-driven way.

It is important to emphasize, however, that the goal of this work is not to provide a fool-proof methodology for language engineering. Rather, it is to provide a strategy for *language design* and a toolbox for implementing this strategy. This will be evident in the following discussion, where a semantics-driven approach does not lead inevitably to an objectively best language, but rather informs design decisions and guides the inherently subjective design process.

6.1 Probabilistic Functional Programming

The first language we consider is a Haskell DSEL for probabilistic modeling, called PFP (probabilistic functional programming) [1]. This language is presented with only minor changes from the published version, made to simplify the discussion.

We begin by considering what a probabilistic model represents at a fundamental level. One obvious answer is a distribution of possible outcomes. By limiting the focus in PFP to discrete probability distributions, we can capture this meaning as a mapping from outcomes to the probabilities that those outcomes occur. We thus begin the design of PFP by partially instantiating the `Map` language schema from Section 3, creating a new language schema `Dist` for representing probability distributions.

```
type Dist a = Map a Float
```

Although we have fixed the representation of probabilities to the language of floating point numbers, this is not the only possibility; for example, probabilities might instead be represented as rational numbers.

We can now instantiate the `Dist` schema with different outcome languages to produce different distribution languages. For example, given the following simple language for coin flip outcomes, `Dist Coin` is the language of distributions of a single coin flip.

```
data Coin = H | T
```

Using this we can define distributions modeling both fair and unfair coins.

```
fair, unfair :: Dist Coin
fair   = H :-> 0.5 :&: T :-> 0.5
unfair = H :-> 0.8 :&: T :-> 0.2
```

On top of this tiny semantic core, PFP provides a large suite of syntactic extensions—operations for extending and manipulating distributions, implemented as functions. Probability distributions have several non-syntactic constraints related to probabilistic axioms. For example, probabilities in a distribution must sum to one and each be between zero and one. Operations must therefore be careful to preserve these properties.

Below we demonstrate a simple syntactic extension of the language with an operation for defining uniform distributions.

```
uniform :: [a] -> Dist a
uniform as = foldr1 (:&:) [a :-> (1/n) | a <- as]
             where n = fromIntegral (length as)
```

Using this, we could instead define the fair coin above as `uniform [H,T]`, or define the distribution of a die roll as `uniform [1..6]`. In the definition of `uniform`, we manually ensure that the probabilistic axioms are preserved and this is not too onerous. For more interesting operations that involve the composition of multiple distributions, this becomes more complicated and thus error-prone. Fortunately, organizational language operators provide a more general solution to this problem.

By observing that probability distributions form a monad, we can carefully define one composition operator (monadic bind) that preserves the axioms, along with an operator for building trivial distributions (monadic return), in order to bring distributions into the monad language family. This gives us immediate access to a huge number of monadic operations for composing and manipulating probability distributions that automatically preserve the probabilistic axioms by virtue of being defined in terms of return and bind. Interestingly, as a class of type constructors, monads are actually a family of *language schemas*. We instantiate the monad schema family for `Dist` as follows, where `toList` is a function that transforms a map, `Map a b`, into an association list, `[(a,b)]`.[4]

```
instance Monad Dist where
  return a = a :-> 1
  d >>= f = foldr1 (:&:) [b :-> (p*q) | (a,p) <- toList d
                                      , (b,q) <- toList (f a)]
```

An interesting feature of the monad language family, when using Haskell as a metalanguage, is that instantiating it also extends the concrete syntax of our language by allowing us to use Haskell's do-notation.

Now we can, for example, write the Cartesian product of two distributions by reusing the `liftM2` composition operator from Haskell's standard libraries.

```
prod :: Dist a -> Dist b -> Dist (a,b)
prod = liftM2 (\a b -> (a,b))
```

And we can confirm that the probabilistic axioms are preserved by examining the product distribution of our fair and unfair coins from above.

```
> prod fair unfair
(H,H) -> 0.4 & (H,T) -> 0.1 & (T,H) -> 0.4 & (T,T) -> 0.1
```

This demonstrates the power of language families for enabling language reuse and promoting structured language extension.

[4] Note that the following is not strictly Haskell code since we cannot instantiate a type class with a partially applied type synonym. In fact, `Dist` is a newtype, but wrapping and unwrapping the nested `Map` value uninterestingly obfuscates the code, so we ignore this detail.

However, not all operations on probability distributions can be implemented in terms of bind. One such example is computing conditional probability distributions. Given a distribution `d` and a predicate `p` on outcomes in `d`, a conditional distribution `d'` is the distribution of outcomes in `d` given that `p` is true. In other words, `p` acts as a filter on `d`, and the probabilities are scaled in `d'` to preserve the probabilistic axioms. We extend the syntax of PFP with a filter operation for computing conditional distributions, described by the following type definition.

```
(|||) :: Dist a -> (a -> Bool) -> Dist a
```

To demonstrate the use of this operator, we also define the following simple predicate on tuples, which returns true if either element in the tuple equals the parameter.

```
oneIs :: Eq a => a -> (a,a) -> Bool
oneIs a (x,y) = a == x || a == y
```

Now we can, for example, compute the distribution of two fair coin tosses, given that one of the tosses comes up heads.

```
> prod fair fair ||| oneIs H
(H,H) -> 0.33 & (H,T) -> 0.33 & (T,H) -> 0.33
```

This discussion has barely scratched the surface of PFP. In addition to many more syntactic extensions (operations on distributions), PFP provides semantic extensions for describing sequences of probabilistic state transitions, running probabilistic simulations, and transforming distributions into random (impure) events. The high extensibility of the language, both syntactically and semantically, is a testament to the benefits of semantics-driven design and an emphasis on language composition. This is also demonstrated in the next subsection, where we directly reuse PFP as a sublanguage in a larger language for explaining probabilistic reasoning.

6.2 Explaining Probabilistic Reasoning

The language described in this subsection focuses on *explaining* problems that require probabilistic reasoning [2, 3]. This language has also been simplified from previously published versions, both for presentation purposes, and to better demonstrate the semantics-driven approach.

We motivate this language with the following riddle: "Given that a family with two children has a boy, what is the probability that the other child is a girl?" Many reply that the probability is one-half, but in fact, it is two-thirds. This solution follows directly from the conditional probability example above. If a birth corresponds to a fair coin flip where heads is a boy and tails is a girl, then we see in the resulting conditional distribution that two out of the three of the remaining outcomes have a girl, and their probabilities sum to two-thirds.

Following the semantics-driven approach, the first step in designing an explanation language is to identify just what an explanation is, on a fundamental level. It turns out that this is an active area of research and hotly-debated topic by philosophers [7]. Ultimately, we opted for a simple and pragmatic explanation representation based on a story-telling metaphor, where an explanation is a sequence of steps that guide the reader from some initial state to the explanandum (that is, the thing that is to be explained).

An initial attempt to represent this semantics in Haskell follows, where the sublanguages `s` and `a` represent the current state at a step and the annotation describing that step, respectively.

```
data Step s a = Step s a
type Expl s a = [Step s a]
```

For explaining probabilistic reasoning problems, we can instantiate these schemas with a probability distribution for `s`, and a simple string describing the step for `a`.

```
type ProbExpl b = Expl (Dist b) String
```

The state of each (non-initial) step in an explanation is derived from the previous step. Rather than encode this relationship in each explanation-building operation, we instead reuse the `Step` schema to extend the semantics with a notion of a *story*. A story is a sequence of annotated steps, where each step is a transformation from the state produced by the previous step to a new state. We can then instantiate a story into an explanation by applying it to an initial state.

```
type Story s a = [Step (s -> s) a]
explain :: Story s a -> s -> Expl s a
```

As an example, we can define the story in the above riddle by a sequence of three steps: add the first child to the distribution, add the second child to the distribution, filter the distribution to include only those families with a boy. We can then instantiate this story with the empty distribution to produce an explanation—essentially a derivation of the conditional distribution from the previous subsection.

However, this explanation is somewhat inadequate since it requires the reader to still identify which outcomes in the final distribution are relevant and add up their probabilities. As a solution, we extend the semantics of probabilistic reasoning explanations by wrapping distributions in a construct that controls how they are viewed, allowing us to group together those cases that correspond to the solution of the riddle. The language schema `G` describes optionally grouped distributions. If a distribution is grouped, a partitioning function maps each element into a group number.

```
data G a = Grouped (a -> Int) (Dist a) | Flat (Dist a)
type ProbExplG b = Expl (G b) String
```

This extension is similar to the addition of privacy to the calendar language in Section 3, in that we extend the semantics by wrapping an existing sublanguage in a language schema that gives us additional control over that language. Finally, we add a fourth step to our story that groups results into two cases depending on whether the other child is a girl or boy, so the riddle's solution can be seen directly in the final grouped distribution.

In addition to several syntactic extensions for creating explanations, in [2] we also extend the semantics to include story and explanation branching, for example, to represent decision points. In [3] we provide several operations for automatically transforming explanations into alternative, equivalent explanations (which might then help a reader who does not understand the initial explanation). This extension highlights a strength of the semantics-driven approach. By focusing on a simple, fundamental representation of explanations these transformations were easy to identify, while they would have been much more difficult to extract from the (quite complex) syntax of explanation creation.

6.3 Choice Calculus

The final language we will consider is the *choice calculus*, a DSL for representing variation in software and other structured artifacts [4]. As published, this is a more traditional language, with a clear separation of syntax and semantics connected by a semantics function. Although the initial design was strongly motivated by a consideration of the semantics, we present a significantly re-designed version of the language here, using a more purely semantics-driven approach.

The essence of a variational artifact is once again a mapping. The range of this mapping is the set of plain artifacts encoded in the variational artifact (that is, its variants), and the domain is the set of decisions that produce those variants. We instantiate the `Map` schema with a language for decisions, defined below, to produce a language schema `V` (which stands for "variational") for the semantics of choice calculus expressions.

```
type V a = Map Decision a
```

For the discussion here, we will use the lambda calculus as our artifact language, represented by the following data type.

```
data LC = Var Name | Abs Name LC | App LC LC
type VLC = V LC
```

We read the `V` schema as *variational*, so `VLC` is the *variational lambda calculus*.

The best representation for decisions is not immediately obvious. One option is to employ a "tagging" approach, where each alternative in a choice (a variation point in the artifact) is labeled with a tag. A decision is then just a list of tags, one selected from each choice. This approach is appealingly simple, but turns out to be too unstructured. As a solution, we introduce in [4] locally scoped *dimensions*, which bind and synchronize related choices. For example, a dimension *OS* might include the tags *Linux*, *Mac*, and *Windows*; every choice in the *OS* dimension must then also contain three alternatives, and the selection of alternatives from these choices would be synchronized.

Therefore, we define decisions to be a list of dimension-tag pairs, representing the tag chosen from each dimension in the variational artifact.

```
type Dim = String
type Tag = String
type Decision = [(Dim,Tag)]
```

To implement locally scoped dimensions, we will need to "lift" the semantics to parameterize it with a notion of context. A context is propagated downward from selections in dimensions, so we represent it as a list associating dimensions with integers, where the integer represents the alternative to select from each choice bound by that dimension.

```
type Context = [(Dim,Int)]
```

We express the lifted semantics below and provide a function to "unlift" the semantics of a top-level variation artifact by applying an empty context.

```
type V' a = Context -> V a
type VLC' = V' LC

unlift :: V' a -> V a
unlift = ($ [])
```

Now we can define the syntax of the choice calculus in terms of this lifted semantics. We must define two operations, for declaring dimensions and introducing choices. The dimension declaration operation takes as arguments a dimension name, its list of tags, and the scope of the declaration. As before, `toList` transforms a `Map` into an association list; we introduce `fromList` to perform the inverse operation.

```
dim :: Dim -> [Tag] -> V' a -> V' a
dim d ts f c = fromList [((d,t):qs,e') | (t,i)   <- zip ts [0..]
                                       , (qs,e') <- toList (f ((d,i):c))]
```

The semantics of this operation is computed by selecting each tag independently in the scope (by prepending `(d,i)` to the context, where the selected tag is the `i`th tag in `ts`) and prepending that selection to the decision of the result.

The operation for introducing choices is much simpler. It accepts its binding dimension name and a list of alternatives as arguments, looks up its dimension in its associated context, and returns the `i`th alternative if an entry in the context is found.

```
chc :: Dim -> [V' a] -> V' a
chc d as c = case lookup d c of
                Just i  -> (as !! i) c
                Nothing -> error ("Unbound choice: " ++ d)
```

We also extend the syntax with smart constructors for variational lambda calculus expressions. Their implementations are omitted for lack of space, but each propagates the corresponding `LC` constructor over the argument semantics. For `app`, the result is a product of the two mappings, where entries are joined by concatenating the decisions and composing the resulting lambda calculus expressions with the `App` constructor.

```
var :: Name -> VLC'
abs :: Name -> VLC' -> VLC'
app :: VLC' -> VLC' -> VLC'
```

Finally, we provide an example of the language in action below. Note that we pretty print the dimension-qualified tag `("D","t")` as `D.t` for readability.

```
> unlift $ dim "A" ["t","u"]
         $ app (chc "A" [var "f", var "g"])
               (chc "A" [var "x", dim "B" ["v","w"]
                                 $ chc "B" [var "y", var "z"]])
[A.t] -> f x & [A.u,B.v] -> g y & [A.u,B.w] -> g z
```

Observe that the two choices in the `A` dimension are synchronized and that a selection in dimension `B` is only required if we select its alternative by selecting `A.u`.

This example demonstrates the flexibility of the semantics-driven approach by showing that it can relatively easily accommodate concepts like scoping that seem at first to be purely syntactic in nature. In particular, the strategy of lifting a semantics language into a functional representation is potentially very powerful, although a full exploration of this idea is left to future work.

7 Related Work

There is a vast literature on language design that approaches the problem from a syntactic point of view; Klint et al. provide a comprehensive overview [12]. Also, the recent flurry of work on language workbenches—essentially integrated development environments that support the creation of DSLs, takes a predominantly syntax-focused approach to language design; for overviews see [17, 20, 14]. In contrast, the approach described in this paper is characterized by its focus on semantics.

Paul Hudak was among the first to advocate DSELs [8, 9] (also called "internal DSLs" [6]) and the compositional approach to developing DSLs. This work has inspired many to develop DSELs for all kinds of application areas (some impressive examples can be found in this collection [10]). Our choice of Haskell as a metalanguage raises the question of how semantics-driven DSL design relates to DSELs. The answer is that the two concepts are quite independent. For example, one might define a DSEL in Haskell by first defining the abstract syntax of the language as a data type, which is decidedly syntax-driven. Similarly, the semantics-driven approach can be equally well applied to non-embedded (external) DSLs. The creation of combinator libraries (for example, in [8]) is a specific strategy for implementing DSELs in functional metalanguages that most closely resembles the semantics-driven approach. Although the design of a combinator library will not necessarily incorporate all aspects of the semantics-driven approach, combinator libraries can nonetheless be viewed as a specific realization of the semantics-driven approach in languages like Haskell.

The use of Haskell for language design and development has also been subject to research. Tim Sheard provides an overview of basic techniques and representations [18]. One particular problem that has been addressed repeatedly is the composition of languages (or language fragments). One proposal is to abstract the recursive structure of data types in a separate definition and use a fixpoint combinator on data types to tie together several languages into one mutual recursive definition [19]. Another approach is to systematically employ monad transformers to gradually extend languages by selected features [13]. Both of these proposals are quite creative and effective. However, they embrace the syntax-oriented view of languages. This is not a bad thing; on the contrary, as far as they go, these approaches provide also effective means to compose parsers for the built languages whereas the semantics-driven approach and internal DSLs have little control over syntax. On the other hand, the opportunities for language composition are rather limited when compared with the semantics-driven approach.

8 Conclusions

In this paper, we have promoted a semantics-driven approach to language development and identified a set of language operators that support the incremental extension and composition of languages in order to realize this approach. Our approach is based on a clear separation of syntax and semantics into different concepts of the chosen metalanguage Haskell, namely functions and data types, respectively.

We have illustrated our approach with several examples, including the design of non-toy languages that have been published and are in use, which demonstrates that semantics-driven language design actually works in practice.

The advantages of our approach are particularly relevant for language prototyping. And while semantics-driven language design can in some cases replace the traditional syntax-focused approach, it can also work as a supplement, to be used as tool to explore the design space before one commits to a specific design that is then implemented using the syntactic approach.

References

1. Erwig, M., Kollmansberger, S.: Probabilistic Functional Programming in Haskell. Journal of Functional Programming 16(1), 21–34 (2006)
2. Erwig, M., Walkingshaw, E.: A DSL for Explaining Probabilistic Reasoning. In: Taha, W.M. (ed.) DSL 2009. LNCS, vol. 5658, pp. 335–359. Springer, Heidelberg (2009)
3. Erwig, M., Walkingshaw, E.: Visual Explanations of Probabilistic Reasoning. In: IEEE Int. Symp. on Visual Languages and Human-Centric Computing, pp. 23–27 (2009)
4. Erwig, M., Walkingshaw, E.: The Choice Calculus: A Representation for Software Variation. ACM Transactions on Software Engineering and Methodology (2011) (to appear)
5. Felleisen, M., Findler, R.B., Flatt, M.: Semantics Engineering with PLT Redex. MIT Press, Cambridge (2009)
6. Fowler, M.: Domain-Specific Languages. Addison-Wesley Professional (2010)
7. Halpern, J., Pearl, J.: Causes and Explanations: A Structural-Model Approach, Part I: Causes. British Journal of Philosophy of Science 56(4), 843–887 (2005)
8. Hudak, P.: Modular Domain Specific Languages and Tools. In: IEEE 5th Int. Conf. on Software Reuse, pp. 134–142 (1998)
9. Hudak, P.: Building Domain-Specific Embedded Languages. ACM Computing Surveys 28(4es), 196 (1996)
10. Gibbons, J., de Moor, O. (eds.): The Fun of Programming. Palgrave MacMillan (2003)
11. Kats, L.C.L., Visser, E., Wachsmuth, G.: Pure and Declarative Syntax Definition: Paradise Lost and Regained. In: ACM Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications, pp. 918–932 (2010)
12. Klint, P., Lämmel, R., Verhoef, C.: Toward an Engineering Discipline for Grammarware. ACM Trans. Softw. Eng. Methodol. 14, 331–380 (2005)
13. Liang, S., Hudak, P., Jones, M.: Monad Transformers and Modular Interpreters. In: 22nd ACM Symp. on Principles of Programming Languages, pp. 333–343 (1995)
14. Merkle, B.: Textual Modeling Tools: Overview and Comparison of Language Workbenches. In: ACM Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications, pp. 139–148 (2010)
15. Mitchell, J.C.: Concepts in Programming Languages. Cambridge University Press, Cambridge (2003)
16. Peyton Jones, S.L.: Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press, Cambridge (2003)
17. Pfeiffer, M., Pichler, J.: A Comparison of Tool Support for Textual Domain-Specific Languages. In: OOPSLA Workshop on Domain-Specific Modeling, pp. 1–7 (2008)
18. Sheard, T.: Accomplishments and Research Challenges in Meta-Programming. In: Taha, W. (ed.) SAIG 2001. LNCS, vol. 2196, pp. 2–44. Springer, Heidelberg (2001)
19. Sheard, T., Pasalic, E.: Two-Level Types and Parameterized Modules. Journal of Functional Programming 14(5), 547–587 (2004)
20. Völter, M., Visser, E.: Language Extension and Composition With Language Workbenches. In: ACM Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications, pp. 301–304 (2010)

Integrating Attribute Grammar and Functional Programming Language Features[⋆]

Ted Kaminski and Eric Van Wyk

Department of Computer Science and Engineering
University of Minnesota, Minneapolis, MN 55455, USA
{tedinski,evw}@cs.umn.edu

Abstract. While attribute grammars have several features making them advantageous for specifying language processing tools, functional programming languages offer a myriad of features also well-suited for such tasks. Much other work shows the close relationship between these two approaches, often in the form of embedding attribute grammars into lazy functional languages. This paper continues in this tradition, but in the other direction, by integrating various functional language features into attribute grammars. Specifically we integrate rich static types (including parametric polymorphism, typed distinctions between decorated and undecorated trees, limited type inference, and generalized algebraic data-types) and pattern-matching, all in a manner that maintains familiar and convenient attribute grammar notations and especially their highly extensible nature.

1 Introduction

Attribute grammars[8] are a programming paradigm for the declarative specification of computations over trees, especially of interest in specifying the semantics of software languages. The underlying context free grammar of the language provides the structure for syntax-directed analysis, and synthesized and inherited attributes provide a convenient means for declaratively specifying the flow of information up and down the tree.

Over the years many additions to this formalism have been proposed to increase the expressiveness, flexibility, extensibility, and convenience of attribute grammars. Higher-order attributes [19] were introduced to, among other things, end the hegemony of the original syntax tree. Many computations are more easily expressed over transformed trees, which is why compilation often involves several intermediate languages, or are not possible without dynamically generating arbitrarily larger trees. Through higher-order attributes, these new trees can be constructed, stored in attributes, and also decorated with attributes.

Reference attributes [5] were introduced to handle non-local dependencies across a tree, and are often described as superimposing a graph structure on the syntax tree. A typical use of reference attributes is obtaining a direct reference

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

to the declaration node of an identifier at a use site of that identifier. These help dramatically in allowing specifications to be written at a high-level.

Forwarding [16] and production-valued attributes were introduced to solve an extensibility problem for attribute grammars. Independently designed *language extensions* can be written as attribute grammar fragments that can add new productions (new language constructs) or new attributes and attribute equations to existing productions (a new analysis or translation). But, these extensions may not compose because the new attributes will not have defining equations on the new productions. Forwarding provides a solution to this problem by permitting new productions to *forward* any queries for unspecified attributes to a *semantically equivalent* tree in the host language, where these attributes would have been defined. This tree does not have to be statically determined, and can be computed dynamically, often by using higher-order attributes.

There are many other useful extensions to attribute grammars such as remote attributes and collections [1], circular attributes [2], generic attribute grammars [12], and more. In this paper, however, we will only be considering the three described above.

Functional programming languages also offer a number of compelling features, such as strong static typing, parametric polymorphism, type inference, pattern matching, and generalized algebraic data types. We are interested in integrating these feature into attribute grammars in order to enjoy the best of both worlds. We have a number of goals in doing so:

1. *Safety.* The language should have features that help the language developer to identify and prevent bugs in their attribute grammar.
2. *Synergy.* The features should work together and not be separate, disjointed parts of the language.
3. *Simplicity.* It should not be a heavy burden on the implementer of the attribute grammar specification language.
4. *Extensible.* We do not want to compromise on one of the biggest advantages that attribute grammars and forwarding provide.
5. *Fully-featured.* These features should be integrated with attribute grammars well enough that they are still as useful and powerful as they are in functional languages.
6. *Natural.* Notation should be convenient, sensible and not overly cumbersome, and error messages should be appropriate and clear.

A strong motivating example for this integration is using attribute grammar constructs for representing type information in a language processor. In this case a nonterminal represent types in the language and productions (with this nonterminal on the left hand side) construct representations for different types. We would definitely like the extensibility properties that attribute grammars and forwarding provide so that we can add new types (new productions) and new analysis over existing types (new attributes). If the attribute grammar is embedded in Haskell, Java, or similar languages, the algebraic data types and classes available in these languages cannot meet these requirements to the same satisfaction that attribute grammars can.

We would equally like a number of functional programming language features for this application. Checking for type equality (or unifying two types) without making use of pattern matching often results in programmers creating an `isFoo` attribute for every production `foo`, along with attributes used only to access the children of a production, and using those to test for equality. This is essentially reinventing pattern matching, badly. These tedious "solutions" are elegantly avoided by allowing pattern matching on nonterminals.

Pattern matching proves useful far beyond just representing types, however. There are many cases in language processing where we care about the *local structure* of sub-trees, and these are all ideal for handling with pattern matching. In many cases, the expressiveness pattern matching provides can be difficult to match with attributes since a large number of attributes are typically necessary to emulate a pattern.

Having decided to represent types using grammars, we may ask what other "data structure"-like aspects of the language definition might benefit from being represented as grammars as well? One example is the type of information typically stored in a symbol table. Extensible *environments*, where new language features can easily add new namespaces, scopes, or other contextual information, become possible simply by representing them as grammars with existing attribute grammar features. New namespaces, for example, can simply be new attributes on the environment nonterminal, and new information can be added to the environment in an extensible way through new productions that make use of forwarding. But without parametric polymorphism, a specialized nonterminal has to be rewritten for every type of information that we wish to store in the environment, which quickly becomes tedious. But with it, we can design environments and symbol table structures in a generic way so that they can be implemented once in a library and reused in different language implementations.

We make the following contributions:

– We describe a small attribute grammar specification language AG, a subset of Silver [17], that captures the essence of most attribute grammar specification languages (section 2.)
– We show how types help simplify the treatment of higher-order, reference, and production-valued attributes (section 2.)
– We describe a type system for AG that carefully integrates all of the desired features (section 3). We also identify and work around a weakness of applying the Hindley-Milner type system to attribute grammars (section 3.2.)
– We describe a method for using types to improve the notation of the language, by automatically inferring whether a child tree node identifier intends to reference the originally supplied tree (on which values for attributes are not available) or the version decorated with attributes (section 3.3.)
– We describe a new interaction with forwarding that permits pattern matching to be used on attribute grammars without compromising the extensibility of the grammar (section 4.)

$$
\begin{array}{lll}
T & ::= & n_v \mid n_n < \overline{T} > \mid \texttt{Decorated}\ n_n < \overline{T} > \mid \texttt{Production}\ (n_n < \overline{T} > \texttt{::=}\ \overline{T}) \\
D & ::= & \cdot \mid \texttt{nonterminal}\ n_n < \overline{n_v} > \ \texttt{;}\ D \\
 & \mid & \texttt{synthesized attribute}\ n_a < \overline{n_v} > \texttt{::}\ T\ \texttt{;}\ D \\
 & \mid & \texttt{inherited attribute}\ n_a < \overline{n_v} > \texttt{::}\ T\ \texttt{;}\ D \\
 & \mid & \texttt{attribute}\ n_a < \overline{T} >\ \texttt{occurs on}\ n_n < \overline{n_v} >\ \texttt{;}\ D \\
 & \mid & \texttt{production}\ n\quad n_l\ \texttt{::}\ n_n < \overline{T} > \texttt{::=}\ \overline{n\ \texttt{::}\ T}\ \{\ \overline{S}\ \}\ D \\
S & ::= & n\ \texttt{.}\ n_a\ \texttt{=}\ E\ \texttt{;} \mid \texttt{forwards to}\ E\ \{\ \overline{A}\ \}\ \texttt{;} \\
A & ::= & n_a\ \texttt{=}\ E \\
E & ::= & n \mid E_f(\overline{E}) \mid E\ \texttt{.}\ n_a \mid \texttt{decorate}\ E\ \texttt{with}\ \{\ \overline{A}\ \} \mid \texttt{new}\ E
\end{array}
$$

Fig. 1. The language AG

2 The AG Language

A number of Silver features are omitted from the language AG, as we wish to focus on those parts of the language that are interesting from a typing and semantics perspective and are generally applicable to other attribute grammar languages. To that end, terminals, other components related to parsing and concrete syntax, aspects productions, collection and local attributes, functions, operations on primitive types, as well as many other basic features are all omitted from AG. There are no major difficulties in extending the contributions of this paper to the full language.

The grammar for the language AG is given in Fig. 1. Names of values (*e.g.* productions and trees) are denoted n, and we follow the convention of denoting nonterminal names as n_n, attribute names as n_a, and type variables as n_v.

A program in AG is a set of declarations, denoted D. These declarations would normally be mutually recursive, but for simplicity of presentation, we consider them in sequence in AG. (Mutual recursion could cause problems for type reconstruction, but we are not relying on type reconstruction in any way for declarations in AG or Silver.) The forms of declaration should be relatively standard for attribute grammars; we take the view of attributes being declared separately from the nonterminals on which they occur.

Nonterminals are parameterized by a set of type variables ($\overline{n_v}$) in angle brackets. We will adopt the convention of omitting the angle brackets whenever this list is empty. Attributes, too, are parameterized by a set of type variables, and it is the responsibility of the `occurs on` declaration to make clear the association between any variables an attribute is parameterized by, and those variables the nonterminal is parameterized by.

Production declarations give a name (n_l) and type ($n_n < \overline{T} >$) to the nonterminal they construct. The name is used to define synthesized attributes for this production or access inherited attributes given to this production inside the body of the production ($\overline{S}$). Each of the children of the production is also given a name and type, and the body of the production consists of a set of (what we will call) statements (S).

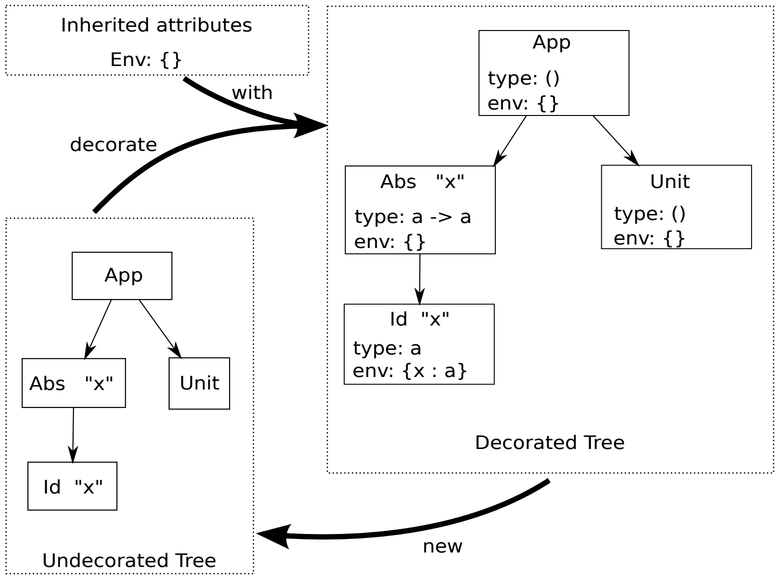


Fig. 2. The distinction between undecorated and decorated trees, and the operations `decorate` and `new` on them. The trees are a representation of the lambda calculus expression $(\lambda x.x)()$.

The attribute definition statement may define synthesized attributes for the node created by the current production, or inherited attributes for its children, depending on whether the name (n) is the left hand side (n_l in production declarations) or the name of a child, respectively.

Forwarding is simply another form of equation that can be written as part of the production body. Forwarding works by *forwarding* requests of any attributes not defined by this production to the *forwarded-to* tree (E). Any inherited attributes requested by the *forwarded-to* tree may be supplied in A, or will otherwise be forwarded to the inherited attributes supplied to the forwarding tree. If no new synthesized attribute equations are given in a production, and no inherited attribute equations are given in the forward, then forwarding behaves identically to simple macro expansion.

Expressions are denoted E. Production application (tree construction) and attribute access are standard. The `new` and `decorate` expressions will be explained momentarily.

In the specification of types, T, we see that every (parameterized) nonterminal n_n produces two distinct, but related, types:

- the undecorated type, denoted n_n, is for trees without computed attribute values (these play the same role as algebraic data types in ML or Haskell)

– the decorated type, denoted `Decorated` n_n, is a tree that is decorated with attributes, created by supplying an undecorated tree with its inherited attributes.

The observed distinction between these two goes back at least as far as [3], and a type distinction between decorated and undecorated trees shows up naturally in functional embeddings of attribute grammars (such as in [6]), but with less familiar notation. Despite this, to the best of our knowledge, the type distinction has not been deliberately exposed as part of an attribute grammar specification language before. For a comparison with other languages, see related work in section 5.

To go along with these types, there are two expressions to convert between the two: `new` creates an undecorated tree from a decorated tree, and `decorate` creates a new decorated tree by supplying it with a list of inherited attribute definitions, denoted A. These operation are illustrated visually in Fig. 2.

Distinguishing these two kinds of trees by types provides some advantages:

– *Enhanced Static Type Safety.* Object-oriented embeddings in particular often do not make this distinction, allowing either kind of tree to be used incorrectly. This stems from the ability to set inherited attributes by side-effects on objects representing trees.
– *Simplicity.* Higher-order [19], reference [5] and production-valued [16] attributes are just ordinary attributes of different types (respectively, n_n, `Decorated` n_n, and `Production`(...)), and need no special treatment by the evaluator or the language.
– *Maintain expressiveness.* Productions can deliberately take decorated trees as children, for example, allowing a tree to share sub-trees with other trees. This is technically allowed in other systems with reference attributes, but at the expense of type safety, and it's not clear it is an intended feature.
– *Convenience.* In section 3.3 we show how these types can be used to provide a convenient notation in Ag.

A small example of a grammar for boolean propositions is given in Fig. 3. The example assumes the existence of a primitive boolean type, not included in our definition of Ag, and one shorthand notation: the `with` syntax of the nonterminal declaration stands in for `occurs on` declarations of Ag for each of the attributes that follow it.

The grammar shows the use of a few of the basic features of attribute grammars, in the notation of Ag. The `eval` attribute is an ordinary synthesized attribute, while the `negation` attribute is a so-called higher-order attribute. In Ag, we can see this is just an ordinary attribute with a different type. The `implies` production shows the use of forwarding in a macro-like fashion, while the `iff` production demonstrates that it's still possible to provide equations for some of the attributes when forwarding.

The example grammar in Fig. 4 shows the use of the polymorphic syntax for nonterminal, attribute, and production declarations for the simple example of the *pair* data structure. We will be referring back to these two example grammars later on in the paper to illustrate some subtle details.

```
nonterminal Expr with eval, negation;
synthesized attribute eval :: Boolean;
synthesized attribute negation :: Expr;

production and
e::Expr ::= l::Expr r::Expr
{ e.eval = l.eval && r.eval;
  e.negation = or(not(l),not(r));
}
production or
e::Expr ::= l::Expr r::Expr
{ e.eval = l.eval || r.eval;
  e.negation = and(not(l),not(r));
}
production not
e::Expr ::= s::Expr
{ e.eval = !s.eval;
  e.negation = s;
}
```

```
production literal
e::Expr ::= b::Boolean
{ e.eval = b;
  e.negation = literal(!b);
}
production implies
e::Expr ::= l::Expr r::Expr
{ forwards to or(not(l),r);
}
production iff
e::Expr ::= l::Expr r::Expr
{ e.eval = l.eval == r.eval;
  forwards to and(implies(l,r),
                  implies(r,l));
}
```

Fig. 3. An simple example grammar for boolean propositions, written in AG

```
nonterminal Pair<a b>;
synthesized attribute fst<a> :: a;
synthesized attribute snd<a> :: a;
attribute fst<a> occurs on Pair<a b>;
attribute snd<d> occurs on Pair<c d>;
```

```
production pair
p::Pair<a b> ::= f::a  s::b
{ p.fst = f;
  p.snd = s;
}
```

Fig. 4. An simple example defining a pair type, written in AG

3 The AG Type System

3.1 The Type Inference Rules

Each of the nonterminals in the (meta) language AG has a rather special typing relation, and so we will describe them in some detail.

$$\boxed{N; P; S; I; O; \Gamma \vdash D} \text{ Declarations}$$

Here N, P, S, I, and O represent, respectively, declared nonterminal types, production names, synthesized attributes, inherited attributes, and occurs-on declarations. These reflect the various components of an attribute grammar specification, but here we specify them explicitly. Since these do not change except at the top level of declarations (D), we will omit writing them for the other typing relations and consider them to be implicitly available.

$\boxed{L; R; \Gamma \vdash S}$ Production body statements

L is the pair of the name and type of the left-hand side symbol of the production (the type the production constructs.) R is the set of name/type pairs for the right-hand side symbols (the children) of the production. These are used to distinguish when it is acceptable to defined inherited or synthesized attributes inside the production statements.

$\boxed{X; \Gamma \vdash A}$ Inherited attribute assignments

Here, X is the type of the nonterminal that inherited attributes are being supplied to by `decorate` expressions and `forwards to` statements.

$\boxed{\Gamma \vdash E : T}$ Expressions

This is the standard relation for expressions, except for N, P, S, I, and O that are implicitly supplied.

Inference Rules. The type inference rules for AG are shown in Fig. 5. In all rules, we omit explicitly checking the validity of types (T) written in the syntax. All that is required to ensure types are valid is that n_n actually refer to a declared nonterminal, with an appropriate number of parameter types. Wherever n_n appears on its own in the syntax, however, we will write these checks explicitly. Additionally, we always require lists of type variables $\overline{n_v}$ to contain no duplicates.

We use $fv(T)$ to represent the free type variables of a type T. This may also be applied to many types ($\overline{T}$), in which case it is the union of the free type variables. To ensure that different sequences of types or type variables have the same number of elements, we use the notation $\overline{T \forall k}$ to indicate that there are k elements in the sequence $\overline{T}$.

The rule D-NT declaring nonterminals is straightforward and adds the nonterminal type to N. We omit basic checks for redeclarations here for brevity. The rule D-SYN adds the type of the synthesized attribute to S and ensures the type of the attribute is closed under the variables it is parameterized by. The rule for inherited attributes is symmetric and not shown.

The rule D-OCC requires some explanation. The actual value stored in O for an occurrence of an attribute on a nonterminal is a function, α, from the nonterminal's type to the type of the attribute. We write $[\overline{n_v \mapsto T}]$ to represent a substitution that maps each type variable to its respective type. The definition of α looks complex but is quite simple: first, we are interested in the type of the attribute (T_a), so that is what the substitution is applied to. We want to equate the variables declared as parameters of the nonterminal (n_{vdn}) with both the variables written in this occurs declaration (n_v) and with the types supplied as a parameter to this function (T_p). Finally, we want to equate the type variables that are parameters of the attribute (n_{vda}) with the actual type supplied for those parameters in this occurs declaration (T). The directions of these rewrites

$$\frac{N \cup n_n < \overline{n_v} >; P; S; I; O; \Gamma \vdash D}{N; P; S; I; O; \Gamma \vdash \texttt{nonterminal}\ \ n_n < \overline{n_v} >\ \ \texttt{;}\ \ D}\ (\text{D-}\textsc{nt})$$

$$\frac{fv(T) \setminus \overline{n_v} = \emptyset \qquad N; P; S \cup n_a < \overline{n_v} > : T; I; O; \Gamma \vdash D}{N; P; S; I; O; \Gamma \vdash \texttt{synthesized attribute}\ \ n_a < \overline{n_v} > \texttt{::}\ T\ \ \texttt{;}\ \ D}\ (\text{D-}\textsc{syn})$$

$$\frac{\begin{array}{c} fv(\overline{T}) \setminus \overline{n_v} = \emptyset \quad n_a < \overline{n_{vda} \forall k} > : T_a \in S \cup I \qquad n_n < \overline{n_{vdn} \forall j} > \in N \\ \alpha(n_n < \overline{T_p \forall j} >) = ([\overline{n_{vda} \mapsto T}] \circ [\overline{n_v \mapsto n_{vdn}}] \circ [\overline{n_{vdn} \mapsto T_p}])(T_a) \\ N; P; S; I; O \cup n_a @ n_n = \alpha; \Gamma \vdash D \end{array}}{N; P; S; I; O; \Gamma \vdash \texttt{attribute}\ \ n_a < \overline{T \forall k} >\ \ \texttt{occurs on}\ \ n_n < \overline{n_v \forall j} >\ \ \texttt{;}\ \ D}\ (\text{D-}\textsc{occ})$$

$$\frac{\begin{array}{c} n_n < \overline{n_{vdn} \forall k} > \in N \\ n_l : n_n < \overline{T_n} >; \overline{n_c : T_c}; \Gamma \cup n_l : \texttt{Decorated}\ n_n < \overline{T_n} > \cup \overline{n_c : dec(T_c)} \vdash \overline{S} \\ T_p = \texttt{Production}\,(n_n < \overline{T_n} > \texttt{::=}\ \ \overline{T_c}) \\ N; P \cup n; S; I; O; \Gamma \cup n : \forall fv(T_p).T_p \vdash D \end{array}}{N; P; S; I; O; \Gamma \vdash \texttt{production}\ \ n\quad n_l \texttt{::}\ n_n < \overline{T_n \forall k} > \texttt{::=}\ \ \overline{n_c \texttt{::}\ T_c}\ \ \{\ \ \overline{S}\ \ \}\ \ D}\ (\text{D-}\textsc{prod})$$

$$\frac{n : T = L,\ \ n_a \in S,\ \ T = n_n < \overline{T_n} >,\ \ n_a @ n_n = \alpha \in O,\ \ \Gamma \vdash E : \alpha(T)}{L; R; \Gamma \vdash n\ \texttt{.}\ n_a\ \ \texttt{=}\ \ E\ \ \texttt{;}}\ (\text{S-}\textsc{syn})$$

$$\frac{n : T \in R,\ \ n_a \in I,\ \ T = n_n < \overline{T_n} >,\ \ n_a @ n_n = \alpha \in O,\ \ \Gamma \vdash E : \alpha(T)}{L; R; \Gamma \vdash n\ \texttt{.}\ n_a\ \ \texttt{=}\ \ E\ \ \texttt{;}}\ (\text{S-}\textsc{inh})$$

$$\frac{n : T = L \qquad \Gamma \vdash E : T \qquad \overline{T; \Gamma \vdash A}}{L; R; \Gamma \vdash \texttt{forwards to}\ \ E\ \ \{\ \ \overline{A}\ \ \}\ \ \ \texttt{;}}\ (\text{S-}\textsc{fwd})$$

$$\frac{n_a \in I \qquad X = n_n < \overline{T_n} > \qquad n_a @ n_n = \alpha \in O \qquad \Gamma \vdash E : \alpha(T)}{X; \Gamma \vdash n_a\ \ \texttt{=}\ \ E}\ (\text{A-}\textsc{inh})$$

$$\frac{n : \forall \overline{n_v}.T_q \in \Gamma}{\Gamma \vdash n : [\overline{n_v \mapsto \nu}]T}\ (\text{E-}\textsc{var}) \qquad\qquad \frac{\Gamma \vdash E : \texttt{Decorated}\ n_n < T_n >}{\Gamma \vdash \texttt{new}\ \ E : n_n < T_n >}\ (\text{E-}\textsc{new})$$

$$\frac{\Gamma \vdash E_f : \texttt{Production}\,(T\ \texttt{::=}\ \ \overline{T_c}) \qquad \overline{\Gamma \vdash E : T_c}}{\Gamma \vdash E_f(\overline{E}) : T}\ (\text{E-}\textsc{app})$$

$$\frac{\Gamma \vdash E : \texttt{Decorated}\ n_n < T_n > \qquad n_a @ n_n = \alpha \in O}{\Gamma \vdash E\ \texttt{.}\ n_a : \alpha(n_n < T_n >)}\ (\text{E-}\textsc{acc})$$

$$\frac{\Gamma \vdash E : n_n < T_n > \qquad \overline{n_n < T_n >; \Gamma \vdash A}}{\Gamma \vdash \texttt{decorate}\ \ E\ \ \texttt{with}\ \ \{\ \ \overline{A}\ \ \}\ \ : \texttt{Decorated}\ n_n < T_n >}\ (\text{E-}\textsc{dec})$$

Fig. 5. Type inference rules AG

are simply such that no type variables from these declarations "escape" into the resulting type. For example, the function α for `fst` attribute on `Pair` shown in Fig. 4 would map `Pair` $< T\ S >$ to T.

The rule D-PROD has a few very particular details. The types written in the production signature are those types that should be supplied when the production is applied. Inside the body of the production, however, the children and left-hand side should appear decorated. So, if a child is declared as having type `Expr` (as in many of the productions of Fig. 3), then inside the production body, its type is seen to be `Decorated Expr`. To accomplish this, we apply *dec* to the types of the children when adding them to the environment (Γ). *dec*'s behavior is simple: it is the identity function, except that nonterminal types $n_n \!<\! \overline{T} \!>$ become their associated decorated types `Decorated` $n_n \!<\! \overline{T} \!>$. The purpose of this is to reflect what the production does: there will be rules inside the production body ($\overline{S}$) that define inherited attributes for its children, and therefore, the children are being automatically decorated by the production and should be seen as decorated within the production body.

Note that when D-PROD checks the validity of its statements $\overline{S}$, it supplies R with types unchanged (that is, without *dec* applied.) This is also important, as inherited attributes can only be supplied to previously undecorated children. Children of already decorated type already have their inherited attributes, and so this information (R, without *dec* applied) is necessary to distinguish between children that were initially undecorated and those that were already decorated and cannot be supplied new inherited attributes.

The rules S-SYN and S-INH are again symmetric, and apply to the same syntax. Which rule is used depends on whether an inherited attribute is being defined for a child, or a synthesized attribute is being defined for the production. Note that we use the shorthand $n_a \in S$ to mean that it is a declared attribute of the appropriate kind, as we no longer care about the type declared for the attribute specifically, that will be obtained from the occurs declaration via the function α. A major subtlety of the rule S-FWD is that the expression type is undecorated. Rule A-INH is similar to S-INH except that we obtain the type from the context, rather than by looking up a name.

The rules E-VAR, and E-APP are slight adaptations of the standard versions of these for the lambda calculus. Notice in the rule E-ACC that the expression type is required to be decorated (attributes cannot be accessed from trees that have not yet been decorated with attributes.) In section 3.2, we consider the problem with this rule, as written, where we must know the type of the left hand side in order to report any type at all for the whole expression, due to the function α. The rules E-DEC and E-NEW should be straightforward, based on their descriptions in the previous section, and the visual in Fig. 2.

Generalized Algebraic Data Types. A full description of GADTs can be found in [11]. Examples of the utility of GADTs are omitted here for space reasons, but many of the examples in the cited functional programming literature make use of them for syntax trees—the application to attribute grammars should be obvious.

The language AG (as it currently stands, without pattern matching) supports GADTs effortlessly. The type system presented in Fig. 5 needs no changes at all, whether GADTs are allowed or not. The only difference is actually syntactic. If the type of the nonterminal on the left hand side of `production` declarations permits types ($\overline{T}$) inside the angle brackets (as they do in Fig. 1), GADTs are supported. If instead, these are restricted to type variables ($\overline{n_v}$), then GADTs are disallowed. All of the complication in supporting GADTs appears to lie in pattern matching, as we will see in section 4.2 when we introduce pattern matching to AG.

3.2 Polymorphic Attribute Access Problem

We encountered an issue in adapting a Hindley-Milner style type system to attributes grammars. Typing the attribute access expression `e.a` immediately raises two problems with the standard inference algorithm:

– There is no type we can unify `e`'s type with. The constraint we wish to express is that, whatever `e`'s type, the attribute `a` occurs on it. That is, $n_a@n_n = \alpha$ has to be in O.
– There is no type that we can report as the type of the whole expression, without knowing `e`'s type, because without that nonterminal type we cannot look up the function α needed to report the attribute's type.

These problems can occur even in the simplest case of parameterized attributes. For example, we cannot know that `e.fst` means that `e` should be a `Pair` or a `Triple`, another nonterminal also decorated by `fst`. Any access of `fst` simply requires knowing what type we're accessing `fst` on.

Fortunately, a sufficient level of type annotation guarantees that the types of subexpressions can be inferred before reporting a type for the attribute access expression. For AG this is not a significant burden: production and attributes need type annotations. Productions require type signatures since they define the underlying context free grammar. And attributes are similar to type classes in Haskell where they also require type signatures. The major downside of needing type annotations is for features not present in AG but present in Silver: functions, let-expressions, "local attributes" and some other features require type annotations whereas type inference on these may be preferred.

Type inference does still provide a significant advantage since it infers the "type parameters" to parameterized productions. In a prototype implementation of parametric polymorphism in Silver [4] without inference one needed to specify, for example, the type of elements in an empty list literal. Explicitly specifying such type parameters quickly becomes tedious.

3.3 Putting Types to Work

In rule D-PROD, nonterminal children are added to the environment in their decorated form for the body of the production (using the function *dec*.) While this is correct behavior, it can be inconvenient, as it can lead to a tedious proliferation of `new` wherever the undecorated form of a child is needed instead.

What we'd like is to have these names refer to *either* of their decorated or undecorated values, and simply disambiguate based upon type. The example grammar in Fig. 3 is already relying on this desired behavior. In the `not` production, we happily access the `eval` attribute from the child `s`, when defining the equation for `eval` on this production. But, we also use `s` as an undecorated value when defining `negation`. As currently written, the type rules would require us to write `new(s)` in the latter case, because the higher-order attribute expects an undecorated value, and `s` is seen as decorated within the production.

The simplest change to the type rules to reflect this idea would be to add a new rule for expressions that is able to refer implicitly to the R and L contextual information given to statements:

$$\frac{n : T \in R \cup L}{\Gamma \vdash n : T} \text{ (E-AsIs)}$$

Unfortunately, simply introducing this rule leads to nondeterminism when type checking. With it, there is no obvious way to decide whether to use it or E-VAR, which is problematic.

To resolve this issue, we introduce a new pseudo-union type of both the decorated and undecorated versions of a nonterminal. But this type, called *Und* for undecorable, will also carry with it a type variable that is specialized to the appropriate decorated or undecorated type when it is used in one way or the other. This restriction reflects the fact that we need to choose between one of these values or the other.

Und is introduced by altering the *dec* function used in D-PROD to turn undecorated child types into undecorable types, rather than decorated types. An undecorable type will freely unify with its corresponding decorated and undecorated type, but in doing so, refines its corresponding hidden type variable.

$$\begin{array}{lrl}
\mathcal{U}(Und\langle n_n < \overline{n_v} >, a\rangle, & n_n < \overline{n_v} >) & \texttt{:-}\ \mathcal{U}(a, n_n < \overline{n_v} >) \\
\mathcal{U}(Und\langle n_n < \overline{n_v} >, a\rangle, & \texttt{Decorated}\ n_n < \overline{n_v} >) & \texttt{:-}\ \mathcal{U}(a, \texttt{Decorated}\ n_n < \overline{n_v} >) \\
\mathcal{U}(Und\langle n_n < \overline{n_v} >, a\rangle, & Und\langle n_n < \overline{n_v} >, b\rangle) & \texttt{:-}\ \mathcal{U}(a, b)
\end{array}$$

Now, suppose we have the admittedly contrived types for `foo` and `bar` below, and we attempt to type the expression `foo(child1, bar, child2)`

```
bar :: Production(Baz ::= Expr)
foo :: Production(Baz ::= a Production(Baz ::= a) a)
```

`child1` will report type $Und\langle Expr, a\rangle$, and `child2` will report $Und\langle Expr, b\rangle$. We will then enforce two constraints while checking the application of `foo`: $Und\langle Expr, a\rangle = Expr$, which using the first rule above will result in requiring $a = Expr$, then $Und\langle Expr, Expr\rangle = Und(\langle Expr\rangle, b)$ which will using the third rule requires $b = Expr$.

$$\begin{aligned} E &::= \texttt{case}\ \ E\ \ \texttt{of}\ \ \overline{p \to E_p} \\ p &::= n_p(\overline{n}) \quad |\ \ \_ \end{aligned}$$

Fig. 6. The pattern extension to AG

The introduction of this undecorable type is something of a special-purpose hack, but the notation gains are worth it. The notational gains could also be achieved with more sophisticated type machinery (like type classes), but it seems worthwhile to stick to the simple Hindley-Milner style of type systems.

4 Pattern Matching

In this section we consider the extensions that must be made to include pattern matching in AG. The main challenge lies in the interaction of forwarding and pattern matching.

4.1 Adding Pattern Matching to AG

Fig. 6 shows the extension to expression syntax for patterns. Note that to simplify our discussion, we are considering only single-value, non-nested patterns. Support for nested patterns that match on multiple values at once can be obtained simply by applying a standard pattern matching compiler, such as [20].

As already noted in the introduction, pattern matching can be emulated with attributes, but that emulation comes at the cost of potentially needing many attributes. One possible translation of pattern matching to attributes begins by creating a new synthesized attribute for each match expression, occurring on the nonterminal it matches on, with the corresponding pattern expression as the attribute equation for each production[1]. This also requires every name referenced in that equation to be turned into an inherited attribute that is passed into that nonterminal by the production performing the match. These names not only include children of the production, but also any pattern variables bound by enclosing pattern matching expressions, such as those created by the pattern compiler from multi-value, nested patterns.

This translation actually does not quite work in AG: pattern matching on reference attributes is problematic because they're already decorated values that we cannot supply with more inherited attributes. In practice, though, there are other language features available that can be used to avoid this problem. Still, this is not a good approach for implementing pattern matching. The most prolific data structures are probably also those pattern matched upon the most, and unless the attribute grammar implementation is specifically designed around solving this problem, there will be overhead for every attribute. A `List` nonterminal,

[1] The observant reader may note here that we have left out wildcards. This is deliberate, and will be considered shortly.

```
nonterminal Type with eq, eqto;
synthesized attribute eq :: Boolean;
inherited attribute eqto :: Type;
```

```
production pair
t::Type ::= l::Type r::Type
{ t.eq =
    case t.eqto of
     pair(a, b) ->
       (decorate l with { eqto = a }).eq &&
       (decorate r with { eqto = b }).eq
    | _ -> false
    end;
}
```

```
production tuple
t::Type ::= ts::[Type]
{ forwards to
    case ts of
      [] -> unit()
    | a:[] -> a
    | a:b:[] -> pair(a, b)
    | f:r -> pair(f, tuple(r))
    end;
}
```

Fig. 7. A use of pattern matching in types

for example, could easily balloon to very many attributes that are the result of translated-away patterns, and there could easily be very many more *cons* nodes in memory. The result would not be memory efficient, to say the least.

The true value of considering this translation to attributes is in trying to resolve the problem pattern matching raises for extensibility. Patterns are explicit lists of productions (constructors), something that works just fine for data types in functional languages because data types are closed: no new constructors can be introduced. Nonterminals are not closed, and this is a major friction in integrating these two language features. However, if pattern matching has a successful reduction to attributes, that problem is already solved: forwarding gives us the solution.

But, the translation to attributes is not quite fully specified: what do we do in the case of wild cards? We choose the smallest possible answer that could still allow us to be sure we cover all cases: wild cards will apply to all productions, not already elsewhere in the list of patterns, that *do not forward*. Thus, we would continue to follow forwards down the chain until either we reach a case in the pattern matching expression, or we reach a non-forwarding, non-matching production, in which case we use the wild card. The dual wild card behavior (applying to all productions, not just those that do not forward) would mean that the "look through forwards" behavior of pattern matching would only occur for patterns without wild cards, which seems unnecessarily limiting.

In Fig. 7, we show a very simple example of the use of pattern matching in determining equality of types. (We again take a few small liberties in notation; new in this example is the use of a list type and some notations for it borrowed from Haskell.) The advantage of interacting pattern matching and forwarding quickly becomes apparent in the example of a tuple extension to the language of types. The tuple type is able to "inherit" its equality checking behavior from whatever type it forwards to, as is normal for forwarding. But, this alone is not sufficient:

consider checking two tuples (`tuple([`S, T, U`])`) against each other. The first will forward to `pair(`S`, pair(`T`, `U`))`, and pattern match on the second. But, without the "look-through" behavior we describe here, the pattern will fail to match, as it will look like a `tuple`. With the behavior, it will successfully match the `pair` production it forwards to, and proceed from there.

Alternative Wild Card Behavior. One alternative might be to take advantage of higher level organizational information (not considered in Ag) to decide which productions to apply the wildcard case to. For example, the wildcard could apply to all *known* productions wherever the case expression appears (based on imports or host/extension information), instead of all non-forwarding productions. This has the advantage that we wouldn't need to repeat the wildcard case for some forwarding productions in those cases where we'd like to distinguish between a production and the production it forwards to, but it has a few disadvantages as well:

- We may now need to repeat case alternatives if we *don't* want to distinguish between a forward we know about (e.g. for syntactic sugar.)
- The meaning of a case expression might change based on where it appears or by changing the imports of the grammar it exists in.
- A "useless imports" analysis would have to become more complex to ensure no pattern matching expressions would change behavior, as the wildcard of a case expression may be *implicitly* referencing that grammar.

As a result, this behavior has enough additional implementation and conceptual complexities that we have not adopted it.

4.2 Typing Pattern Matching Expressions

Matching on undecorated trees seems to introduce no new interesting behavior different from pattern matching on ordinary data types, which makes sense because in a sense undecorated trees are not different from ordinary data types. Pattern matching on decorated trees, however, introduces a couple of interesting behaviors:

- As we saw in the previous section, we can evaluate the forward of a production and allow pattern matching to "look through" to the forward.
- We can also allow pattern variables to extract the decorated children of a production, rather than just the undecorated children.

Further, restricting pattern matching to only apply to decorated trees doesn't lose us anything: if it makes sense to pattern match on an undecorated tree, then the `case` construct can simply decorate a tree with no inherited attributes to pattern match upon it. As a result, we have decided to just consider pattern matching on decorated trees in Ag.

The type rules for patterns are shown in Fig. 8. Notice in E-case that the scrutinee expression (E) must be a decorated type. Also note that *dec* is applied

$$\frac{\Gamma \vdash E : \texttt{Decorated}\ n_n < \overline{T} > \qquad \Gamma \vdash \overline{p \to E_p} : n_n < \overline{T} > \to T}{\Gamma \vdash \texttt{case}\ \ E\ \ \texttt{of}\ \ \overline{p \to E_p} : T} \ (\text{E-CASE})$$

$$\frac{\begin{array}{cc} n_p \in P & \Gamma \vdash n_p : \texttt{Production}(T_n \texttt{ ::= } \overline{T_c}) \\ \theta \in mgu(T_s = T_n) & \theta(\Gamma, \overline{n : dec(T_c)}) \vdash E_p : \theta(T_r) \end{array}}{\Gamma \vdash n_p(\overline{n}) \to E_p : T_s \to T_r} \ (\text{P-PROD})$$

Fig. 8. The additional typing rules for pattern matching expressions

directly to T_c in P-PROD. The reason for this is to allow pattern matching to extract the decorated trees corresponding to a node's children. This function (*dec*) must be applied prior to any type information outside the original declaration of the production being considered, in order to be accurate about which children are available as decorated trees. For example, the `pair` production in Fig. 4 would not be decorating its children, even though they might turn out to be a (undecorated) nonterminal type (i.e. a pair of `Expr`), because to the pair production, the types of its children are type variables. Applying *dec* early means that here we see the type of the children as type variables, rather than a specific type, just as the original production would have.

The use of θ in the type rule P-PROD is the cost that we must pay for supporting GADTs in patterns. The details for handling GADTs in patterns are adapted from [11], as this approach seemed especially simple to implement. In that paper, much attention is paid to a notion of *wobbly* and *rigid* types. Thanks to the concessions in type reconstruction we must make due to the attribute access problem discussed in section 3.2, all bindings in AG can be considered rigid in their sense, vastly simplifying the system even more.

The essential idea is to compute a *most general unifier* (θ) between the pattern scrutinee's type and the result type of the production [2]. We then check the right hand side of the alternative, under the assumptions of the unifier. In effect, all this rule is really stating is that whatever type information we learn from successfully matching a particular GADT-like production stays confined to that branch of the pattern matching expression.

4.3 Other Concerns

No new special cases need to be introduced to perform a well-definedness test in the presence of pattern matching, as pattern matching can be translated to attributes (the troubles mentioned earlier are eliminated if we are allowed full-program information), and forwarding can also be translated away to higher order attribute grammars [16].

[2] The need to concern ourselves with "fresh" most general unifiers in the sense of the cited paper is eliminated again due to the lack of "wobbly" types.

The standard techniques apply for ensuring exhaustive matching of patterns, except that we only need to consider productions that do not forward as the essential cases to cover.

Although it is often glossed over in descriptions of type systems, it's worth noting that we're allowing type variables to appear in productions' right hand sides (that is, in the children) that do not appear in the left hand side. This corresponds to a notion of existential types in functional languages, but we do not require any special `forall` notation to include them. Background discussion on existential types can be found in [9].

5 Related Work

The integration of pattern matching and forwarding we present in this paper is novel. Some aspects of the rest of the system can be found in scattered in other attribute grammar languages in various forms, but not in ways that provide both the type safety and the familiar and convenient notations that we provide here.

In JastAdd [5] and Kiama[14], trees are represented as objects and attribute evaluation mutates the tree effectfully (either directly as in JastAdd or indirectly via memoization as in Kiama.) As a result, both of these languages lack a type distinction between the two kinds of trees. Instead, the user must remember to invoke a special copy method, analogous to our `new` expression, wherever a new undecorated tree is needed. These copy methods do not change the type of the tree, as our `new` operation does, resulting in a lack of the type safety that we have here. UUAG[15] does not appear to support reference attributes, and so the type distinction is irrelevant. In functional embeddings these type distinctions occur naturally but at the notational cost of typically having different names for the two views of the tree and needing to explicitly create the decorated tree from the undecorated one. AspectAG [18] is a sophisticated embedding into Haskell that naturally maintains the type safety we seek but at some loss of notational convenience. It also requires a fair amount of so-called "type-level" programming that is less direct than the Silver specifications, and the error messages generated can be opaque.

Kiama and UUAG, by virtue of their embedding in functional languages, do support parameterized nonterminals and attributes. UUAG side-steps the attribute access problem of section 3.2 by simply not having reference attributes. All attribute access are therefore only on children, which have an explicit type signature provided. UUAG does not appear to support GADT-like productions, but we suspect it could be easily extended to. Both also support pattern matching on nonterminals. In UUAG, this is only supported for undecorated trees, and its behavior is identical to ordinary pattern matching in Haskell. In Kiama, pattern matching can extract decorated children from a production. But in both cases, use of pattern matching would compromise the extensibility of the specification. Rascal [7] allows trees to be dynamically annotated with values, similar to adding attribute *occurs-on* specifications dynamically. However, the presence of annotations is not part of the static type system and thus neither is the distinction between decorated and undecorated trees.

Scala's [10] support for pattern matching and inheritance presents the same type of extensibility problem we faced when integrating pattern matching with forwarding. Their solution is a notion of *sealed* classes that simply prevent new classes from outside the current file from inheriting from it directly.

In [13], a type system with constraints powerful enough to capture the α functions created by our *occurs* declarations is presented. To regain full type inference, we believe the basic Hindley-Milner style system must be abandoned in favor of something at least this powerful.

6 Future Work

Typing attribute grammars offers a wealth of future work possibilities. The language AG is not quite suitable for proving soundness results, as writing down operational semantics for it would be overly complicated. Instead, we would like to develop a smaller core *attribute calculus*, with an appropriate operational semantics and obtain a soundness result from that. To get the simple Hindley-Milner type system to apply, we sacrificed the ability to remove some type annotations from the languages. We believe a more powerful core type system (such as in [13]) will permit inference to work freely. The type system currently does not permit many forms of functions over data structures to be recast as attributes. For example, quantifiers are not permitted in the right places to allow `if-then-else` to somehow be written as attributes on a `Boolean` nonterminal. Attributes also cannot occur on only some specializations of a nonterminal, which means natural functions like `sum` over a list of integers cannot be recast as attributes. (Such computations are realized as functions in Silver.)

Furthermore, the traditional well-definedness tests for attribute grammars may have another useful interpretation in terms of types, perhaps refining our blunt distinction between undecorated and decorated types. There may also be refinements possible due to the presence of pattern matching. Generic attribute grammars[12] are partly covered by polymorphic nonterminals, except for their ability to describe constraints on the kinds of types that can be incorporated. For example, in Silver, permitting nonterminals to require type variables to be concrete types, permitting these type variables to appear in concrete syntax, and reifying the result before it is sent to the parser generator would be a useful addition to the language. Finally, we would like to account for circular attributes, which are extremely useful for fixed point computations. It would be interesting to see if there is a type-based distinction for circular attributes, just as we show for reference, higher-order, and production attributes in this paper.

7 Conclusion

In this paper we have claimed that certain features found in modern functional languages can be added to an attribute grammar specification language to provide a number of benefits. By using types to distinguish decorated and undecorated trees the type system can prevent certain errors and help to provide more convenient notations. Pattern matching on decorated trees adds a measure of convenience and expressiveness (in the informal sense) to attribute grammar specification languages, and crucially, it can be done while maintaining the extensibility possible with forwarding. Parameterized nonterminals and productions can play the same role as algebraic data types in functional languages; they can be used as syntax trees or for more general purpose computations. Furthermore, GADT-like productions are a very natural fit for attribute grammars.

However, in scaling Ag up to Silver, the type annotations requirement to get around the attribute access problem stands in the way of meeting our *full-featured* goal described in section 1. This means that functions and local attributes, for example, must specify their types.

In integrating these features into an attribute grammar specification language we found that some small modifications to the implementation of Hindley-Milner typing were needed. To meet our goals of having natural and familiar notations (for attribute access and in order to infer if the decorated or undecorated version of a tree is to be used) it was helpful to have direct control over the type system to make modifications so that attribute grammar-specific concerns could be addressed. Supporting GADT-like productions and pattern matching that is compatible with forwarding required similar levels of control of the languages implementation and translation.

We previously added polymorphic lists and a notion of pattern-matching that was not compatible with forwarding using language extensions [17]. While this approach does allow expressive new features to be added to the language, it could not accomplish all of our goals, as adding a new typing infrastructure (for type inference) *replaces* and does not extend the previous type system in Silver. Adding these kinds of features by embedding attribute grammars in a function language or writing a preprocessor that is closely tied to the underlying implementation language can also make it more difficult to achieve these goals. However, an advantage of these approaches that should not be overlooked is that many useful features of the underlying language can be used "for free" with no real effort on the attribute grammar system designer to include them into their system. It is difficult to draw any conclusions beyond noting that these are the sort of trade-offs that AG system implementers, specifically (and DSL implementers, more generally) need to consider.

References

1. Boyland, J.T.: Remote attribute grammars. J. ACM 52(4), 627–687 (2005)
2. Farrow, R.: Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. SIGPLAN Notices 21(7) (1986)
3. Ganzinger, H., Giegerich, R.: Attribute coupled grammars. SIGPLAN Notices 19, 157–170 (1984)
4. Gao, J.: An Extensible Modeling Language Framework via Attribute Grammars. Ph.D. thesis, University of Minnesota, Department of Computer Science and Engineering, Minneapolis, Minnesota, USA (2007)
5. Hedin, G.: Reference attribute grammars. Informatica 24(3), 301–317 (2000)

6. Johnsson, T.: Attribute grammars as a functional programming paradigm. In: Kahn, G. (ed.) FPCA 1987. LNCS, vol. 274, pp. 154–173. Springer, Heidelberg (1987)
7. Klint, P., van der Storm, T., Vinju, J.: Rascal: a domain specific language for source code analysis and manipulation. In: Proc. of Source Code Analysis and Manipulation, SCAM 2009 (2009)
8. Knuth, D.E.: Semantics of context-free languages. Mathematical Systems Theory 2(2), 127–145 (1968); corrections in 5, 95–96 (1971)
9. Läufer, K., Odersky, M.: Polymorphic type inference and abstract data types. ACM Trans. on Prog. Lang. and Systems (TOPLAS) 16(5), 1411–1430 (1994)
10. Odersky, M., Spoon, L., Venners, B.: Programming in Scala, 2nd edn. Artima (2010)
11. Peyton Jones, S., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for GADTs. In: Proc. of the Eleventh ACM SIGPLAN International Conf. on Functional Programming, pp. 50–61. ACM (2006)
12. Saraiva, J., Swierstra, D.: Generic Attribute Grammars. In: 2nd Workshop on Attribute Grammars and their Applications, pp. 185–204 (1999)
13. Schrijvers, T., Peyton Jones, S., Sulzmann, M., Vytiniotis, D.: Complete and decidable type inference for GADTs. In: Proc. of the 14th ACM SIGPLAN International Conf. on Functional Programming, pp. 341–352. ACM (2009)
14. Sloane, A., Kats, L., Visser, E.: A pure object-oriented embedding of attribute grammars. In: Proc. of Language Descriptions, Tools, and Applications (LDTA 2009). ENTCS, vol. 253, pp. 205–219. Elsevier Science (2010)
15. Swierstra, S., Alcocer, P., Saraiva, J.: Designing and Implementing Combinator Languages. In: Swierstra, S.D., Oliveira, J.N. (eds.) AFP 1998. LNCS, vol. 1608, pp. 150–206. Springer, Heidelberg (1999)
16. Van Wyk, E., de Moor, O., Backhouse, K., Kwiatkowski, P.: Forwarding in Attribute Grammars for Modular Language Design. In: Horspool, R.N. (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. 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 Conf. on Functional Programming, ICFP 2009 (2009)
19. 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 (1990)
20. Wadler, P.: Efficient compilation of pattern matching. In: The Implementation of Functional Programming Languages, pp. 78–103. Prentice-Hall (1987)

Parse Forest Diagnostics with Dr. Ambiguity

Hendrikus J.S. Basten and Jurgen J. Vinju

Centrum Wiskunde & Informatica (CWI)
Science Park 123, 1098 XG Amsterdam, The Netherlands
{Jurgen.Vinju,Bas.Basten}@cwi.nl

Abstract. In this paper we propose and evaluate a method for locating causes of ambiguity in context-free grammars by automatic analysis of parse forests. A parse forest is the set of parse trees of an ambiguous sentence. Deducing causes of ambiguity from observing parse forests is hard for grammar engineers because of (a) the size of the parse forests, (b) the complex shape of parse forests, and (c) the diversity of causes of ambiguity.

We first analyze the diversity of ambiguities in grammars for programming languages and the diversity of solutions to these ambiguities. Then we introduce DR. AMBIGUITY: a parse forest diagnostics tools that explains the causes of ambiguity by analyzing differences between parse trees and proposes solutions. We demonstrate its effectiveness using a small experiment with a grammar for Java 5.

1 Introduction

This work is motivated by the use of parsers generated from general context-free grammars (CFGs). General parsing algorithms such as GLR and derivates [3, 6, 9, 17, 35], GLL [22,34], and Earley [16,32] support parser generation for highly non-deterministic context-free grammars. The advantages of constructing parsers using such technology are that grammars may be modular and that real programming languages (often requiring parser non-determinism) can be dealt with efficiently[1]. It is common to use general parsing algorithms in (legacy) language reverse engineering, where a language is given but parsers have to be reconstructed [25], and in language extension, where a base language is given which needs to be extended with unforeseen syntactical constructs [10].

The major disadvantage of general parsing is that multiple parse trees may be produced by a parser. In this case, the grammar was not only non-deterministic, but also *ambiguous*. We say that a grammar is ambiguous if it generates more than one parse tree for a particular input sentence. Static detection of ambiguity in CFGs is undecidable in general [13, 15, 18].

It is not an overstatement to say that ambiguity is the Achilles' heel of CFG-general parsing. Most grammar engineers who are building a parser for a programming language intend it to produce a single tree for each input program. They use a general parsing algorithm to efficiently overcome problematic non-determinism, while ambiguity is an unintentional and unpredictable side-effect. Other parsing technologies, for

[1] Linear behavior is usually approached and most algorithms can obtain cubic time worst time complexity [33].

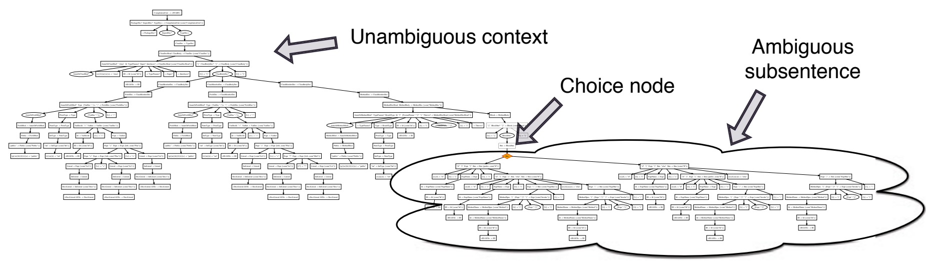


Fig. 1. The complexity of a parse forest for a trivial Java class with one method; the indicated subtree is an ambiguous if-with-dangling-else issue (180 nodes, 195 edges)

```
If (                                   <
ExprName ( Id ( "a" ) ),               <
IfElse (                                 IfElse (
                                       > ExprName ( Id ( "a" ) ),
                                       > If (
ExprName ( Id ( "b" ) ),                 ExprName ( Id ( "b" ) ),
ExprStm (                                ExprStm (
Invoke (                                 Invoke (
Method ( MethodName ( Id ( "a" ) ) ),    Method ( MethodName ( Id ( "a" ) ) ),
[                                        [
] ) ),                                 | ] ) ) ),
ExprStm (                                ExprStm (
Invoke (                                 Invoke (
Method ( MethodName ( Id ( "b" ) ) ),    Method ( MethodName ( Id ( "b" ) ) ),
[                                        [
] ) ) ) )                              | ] ) ) )
```

Fig. 2. Using `diff -side-by-side` to diagnose a trivial ambiguous syntax tree for a dangling else in Java (excerpts of Figure [1])

example Ford's PEG [19] and Parr's LL(*) [28], do not report ambiguity. Nevertheless, these technologies also employ disambiguation techniques (ordered choice, dynamic lookahead). In combination with a debug-mode that does produce all derivations, the results in this paper should be beneficial for these parsing techniques as well. It should help the user to intentionally select a disambiguation method. In any case, the point of departure for the current paper is any parsing algorithm that will produce all possible parse trees for an input sentence.

In other papers [4,5] we present a fast ambiguity detection approach that combines approximative and exhaustive techniques. The output of this method are the ambiguous sentences found in the language of a tested grammar. Nevertheless, this is only a observation that the patient is ill, and now we need a cure. We therefore will diagnose the sets of parse trees produced for specific ambiguous sentences. The following is a typical grammar engineering scenario:

1. While testing or using a generated parser, or after having run a static ambiguity detection tool, we discover that one particular sentence leads to a set of multiple parse trees. This set is encoded as a single parse forest with choice nodes where sub-sentences have alternative sub-trees.
2. The parser reports the location in the input sentence of each choice node. Note that such choice nodes may be nested. Each choice node might be caused by a different ambiguity in the CFG.
3. The grammar engineer extracts an arbitrary ambiguous sub-sentence and runs the parser again using the respective sub-parser, producing a set of smaller trees.
4. Each parse tree of this set is visualized on a 2D plane and the grammar engineer spots the differences, or a (tree) diff algorithm is run by the grammar engineer to spot the differences. Between two alternative trees, either the shape of the tree is totally different (rules have moved up/down, left/right), or completely different rules have been used, or both. As a result the output of diff algorithms and 2D visualizations typically require some effort to understand. Figure 1 illustrates the complexity of an ambiguous parse forest for a 5 line Java program that has a dangling else ambiguity. Figure 2 depicts the output of diff on a strongly simplified representation (abstract syntax tree) of the two alternative parse trees for the same nested conditional. Realistic parse trees are not only too complex to display here, but are often too big to visualize on screen as well. The common solution is to prune the input sentence step-by-step to eventually reach a very minimal example that still triggers the ambiguity but is small enough to inspect.
5. The grammar engineer hopefully knows that for some patterns of differences there are typical solutions. A solution is picked, and the parser is regenerated.
6. The smaller sentence is parsed again to test if only one tree (and which tree) is produced.
7. The original sentence is parsed again to see if all ambiguity has been removed or perhaps more diagnostics are needed for another ambiguous sub-sentence. Typically, in programs one cause of ambiguity would lead to several instances distributed over the source file. One disambiguation may therefore fix more "ambiguities" in a source file.

The issues we address in this paper are that the above scenario is (a) an expert job, (b) time consuming and (c) tedious. We investigate the invention of an *expert system* that can automate finding a concise grammar-level explanation for any choice node in a parse forest and propose a set of solutions that will eliminate it. This expert system is shaped as a set of algorithms that analyze sets of alternative parse trees, simulating what an expert would do when confronted with an ambiguity.

The contributions of this paper are an overview of common causes of ambiguity in grammars for programming language (Section 3), an automated tool (DR. AMBIGUITY) that diagnoses parse forests to propose one or more appropriate disambiguation techniques (Section 4) and an initial evaluation of its effectiveness (Section 5). In 2006 we published a manual [36] to help users disambiguate SDF2 grammars. This well-read manual contains recipes for solving ambiguity in grammars for programming languages. DR. AMBIGUITY automates all tasks that users perform when applying the recipes from this manual, except for finally adding the preferred disambiguation declaration.

We need the following definitions. A *context-free grammar* G is defined as a 4-tuple (T, N, P, S), namely finite sets of terminal symbols T and non-terminal symbols N, production rules P in $A \times (T \cup N)^*$ written like $A \to \alpha$, and a start symbol S. A *sentential form* is a finite string in $(T \cup N)^*$. A *sentence* is a sentential form without non-terminal symbols. An ε denotes the empty string. We use the other lowercase greek characters $\alpha, \beta, \gamma, \ldots$ for variables over sentential forms, uppercase roman characters for non-terminals $(A, B, \ldots)$ and lowercase roman characters and numerical operators for terminals $(a, b, +, -, *, /)$. By applying production rules as substitutions we can generate new sentential forms. One substitution is called a *derivation step*, e.g. $\alpha A\beta \Rightarrow \alpha\gamma\beta$ with rule $A \to \gamma$. We use $\Rightarrow^*$ to denote sequences of derivation steps. A *full derivation* is a sequence of production rule applications that starts with a start symbol and ends with a sentence. The *language* of a grammar is the set of all sentences derivable from S. In a *bracketed derivation* [20] we record each application of a rule by a pair of brackets, for example $S \Rightarrow (\alpha E\beta) \Rightarrow (\alpha(E + E)\beta) \Rightarrow (\alpha((E * E) + E)\beta)$. Brackets are (implicitly) indexed with their corresponding rule.

A *non-deterministic derivation sequence* is a derivation sequence in which a $\diamond$ operator records choices between different derivation sequences. I.e. $\alpha \Rightarrow (\beta) \diamond (\gamma)$ means that either β or γ may be derived from α using a single derivation step. Note that β does not necessarily need to be different from γ. An example non-deterministic derivation is $E \Rightarrow (E + E) \diamond (E * E) \Rightarrow (E + (E * E)) \diamond ((E + E) * E)$. A *cyclic derivation sequence* is any sequence $\alpha \Rightarrow^+ \alpha$, which is only possible by applying rules that do not have to eventually generate terminal symbols, such as $A \to A$ and $A \to \varepsilon$.

A *parse tree* is an (ordered) *finite* tree representation of a bracketed *full* derivation of a specific sentence. Each pair of brackets is represented by an internal node labeled with the rule that was applied. Each terminal is a leaf node. This implies the leafs of a parse tree form a sentence. Note that a single parse tree may represent several equivalent derivation sequences. Namely in sentential forms with several non-terminals one may always choose which non-terminal to expand first. From here on we assume a canonical left-most form for such equivalent derivation sequences, in which expansion always occurs at the left-most non-terminal in a sentential form.

A *parse forest* is a set of parse trees possibly extended with *ambiguity nodes* for each use of choice ($\diamond$). Like parse trees, parse forests are limited to represent full derivations of a *single* sentence, each child of an ambiguity node is a derivation for the same sub-sentence. One such child is called an *alternative*. For simplicity's sake, and without loss of generality, we assume that all ambiguity nodes have exactly two alternatives.

A parse forest is *ambiguous* if it contains at least one ambiguity node. A sentence is *ambiguous* if its parse forest is ambiguous. A grammar is *ambiguous* if it can generate at least one ambiguous sentence. An *ambiguity* in a sentence is an ambiguity node. An *ambiguity* of a grammar is the cause of such aforementioned ambiguity. We define *cause of ambiguity* precisely in Section 3. Note that *cyclic derivation* sequences can be represented by parse forests by allowing them to be graphs instead of just trees [29].

A *recognizer* for G is a terminating function that takes any sentence α as input and returns true if and only if $S \Rightarrow^* \alpha$. A *parser* for G is a terminating function that takes any finite sentence α as input and returns an error if the corresponding recognizer would not return true, and otherwise returns a *parse forest* for α. A *disambiguation filter* is a function that takes a parse forest for α and returns a smaller parse forest for α [24]. A *disambiguator* is a function that takes a parser and returns a parser that produces smaller parse forests. Disambiguators may be implemented as parser actions, or by parser generators that take additional disambiguation constructs as input [9]. We use the term *disambiguation* for both disambiguation filters and disambiguators.

2 Solutions to Ambiguity

There are basically two kinds of solutions to removing ambiguity from grammars. The first involves restructuring the grammar to accept the same set of sentences but using different rules. The second leaves the grammar as-is, but adds disambiguations (see above). Although grammar restructuring is a valid solution direction, we restrict ourselves to disambiguations described below. The benefit of disambiguation as opposed to grammar restructuring is that the shape of the rules, and thus the shape of the parse trees remains unchanged. This allows language engineers to maintain the intended semantic structure of the language, keeping parse trees directly related to abstract syntax trees (or even synonymous) [21].

Any solution may be *language preserving*, or not. We may change a grammar to have it generate a different language, or we may change it to generate the same language differently. Similarly, a disambiguation may remove sentences from a language, or simply remove some ambiguous derivation without removing a sentence. This depends on whether or not the filter is applied always in the context of an ambiguous sentence, i.e. whether another tree is guaranteed to be left over after a certain tree is filtered. It may be hard for a language engineer who adds a disambiguation to understand whether it is actually language preserving. Whether or not it is good to be language preserving depends entirely on ad-hoc requirements. The current paper does not answer this question. Where possible, we do indicate whether adding a certain disambiguation is expected to be language preserving. Proving this property is out-of-scope.

Solving ambiguity is sometimes confused with making parsers deterministic. From the perspective of the current paper, non-determinism is a non-issue. We focus solely on solutions to ambiguity.

We now quote a number of disambiguation methods here. Conceptually, the following list contains nothing but disambiguation methods that are commonly supported by lexer and parser generators [1]. Still, the precise semantics of each method we present here may be specific to the parser frameworks of SDF2 [21, 37] and RASCAL [23]. In particular, some of these methods are specific to *scannerless* parsing, where a context-free grammar specifies the language down to the character level [30, 37]. We recommend [7], to appreciate the intricate differences between semantics of operator priority mechanisms between parser generators.

Priority disallows certain direct edges between pairs of rules in parse trees in order to affect operator priority. For instance, the production for the + operator may not be a direct child of the $*$ production [9].
Formally, let a priority relation $>$ be a partial order between recursive rules of an expression grammar. If $A \rightarrow \alpha_1 A \alpha_2 > A \rightarrow \beta_1 A \beta_2$ then all derivations $\gamma A \delta \Rightarrow \gamma(\alpha_1 A \alpha_2)\delta \Rightarrow \gamma(\alpha_1(\beta_1 A \beta_2)\alpha_2)$ are illegal.

Associativity is similar to priority, but father and child are the same rule. It can be used to affect operator associativity. For instance, the production of the + operator may not be a direct *right* child of itself because + is left associative [9]. Left and right associativity are duals, and *non-assocativity* means no nesting is allowed at all. Formally, if a recursive rule $A \rightarrow A\alpha A$ is defined left associative, then any derivation $\gamma A\delta \Rightarrow \gamma(A\alpha A)\delta \Rightarrow \gamma(A\alpha(A\alpha A))\delta$ is illegal.

Offside disallows certain derivations using the would-be indentation level of an (indirect) child. If the child is "left" of a certain parent, the derivation is filtered [26]. One example formalization is to let $\Pi(x)$ compute the start column of the sub-sentence generated by a sentential form x and let $>$ define a partial order between production rules. Then, if $A \rightarrow \alpha_1 X\alpha_2 > B \rightarrow \beta$ then all derivations $\gamma A\delta \Rightarrow \gamma(\alpha_1 X\alpha_2)\delta \Rightarrow^* \gamma(\alpha_1(\ldots(\beta)\ldots)\alpha_2)\delta)$ are illegal if $\Pi(\beta) < \Pi(\alpha_1)$. Parsers may employ subtly different offside disambiguators, depending on how Π is defined for each different language or even for each different production rule within a language.

Preference removes a derivation, but only if another one of higher preference is present. Again, we take a partial ordering $>$ that defines preference between rules for the same non-terminal. Let $A \rightarrow \alpha > A \rightarrow \beta$, then from all derivations $\gamma A\delta \Rightarrow \gamma((\alpha) \diamond (\beta))\delta$ we must remove (β) to obtain $A \Rightarrow \gamma(\alpha)\delta$.

Reserve disallows a fixed set of terminals from a certain (non-)terminal, commonly used to reserve keywords from identifiers. Let K be a set of sentences and let I be a non-terminal from which they are declared to be reserved. Then, for every $\alpha \in K$, any derivation $I \Rightarrow^* \alpha$ is illegal.

Reject disallows the language of a certain non-terminal from that of another one. This may be used to implement **Reserve**, but it is more powerful than that [9]. Let $(I$ **-** $R)$ declare that the non-terminal R is rejected from the non-terminal I. Then any derivation sequence $I \Rightarrow^* \alpha$ is illegal if and only if $R \Rightarrow^* \alpha$.

Not Follow/Precede declarations disallow derivation steps if the generated sub-sentence in its context is immediately followed/preceded by a certain terminal. This is used to affect longest match behavior for regular languages, but also to solve "dangling else" by not allowing the short version of `if`, when it would be immediately followed by `else` [9]. Formally, we define follow declaration as follows. Given $A \Rightarrow^* \alpha$ and a declaration A **not-follow** β, where β is a sentence, any derivation $S \Rightarrow^* \gamma A\beta\delta \Rightarrow^* \gamma(\alpha)\beta\delta$ is illegal. We should mention that **Follow** declarations may simulate the effect of "shift before reduce" heuristics that deterministic — LR, LALR — parsers use when confronted with a shift/reduce conflict.

Dynamic Reserve disallows a dynamic set of sub-sentences from a certain non-terminal, i.e. using a symbol table [1]. The semantics is similar to **Reject**, where the set R is dynamically changed as certain derivations (i.e. type declarations) are applied.

Types removes certain type-incorrect sub-trees using a type-checker, leaving correctly typed trees as-is [12]. Let $C(d)$ be true if and only if derivation d (represented by a tree) is a type-correct part of a program. Then all derivations $\gamma A\delta \Rightarrow \gamma(\alpha)\delta$ are illegal if $C(\alpha)$ is false.

Heuristics There are many kinds of heuristic disambiguation that we bundle under a single definition here. The preference of "Islands" over "Water" in island grammars is an example [27]. Preference filters are sometimes generalized by counting the number of preferred rules as well [9]. Counting rules is used sometimes to choose a "simplest" derivation, i.e. the most shallow trees are selected over deeper ones. Formally, Let $C(d)$ be any function that maps a derivation (parse tree) to an integer. If $C(A \Rightarrow \alpha) > C(A \Rightarrow \beta)$ then from all derivations $A \Rightarrow^* (\alpha) \diamond (\beta)$ we must remove (β) to obtain $A \Rightarrow (\alpha)$.

Not surprisingly, each kind of disambiguation characterizes certain properties of derivations. In the following section we link such properties to causes of ambiguity. Apart from **Types** and **Heuristics** (which are too general to automatically report specific suggestions for), we can then link the causes explicitly back to the solution types.

3 Causes of Ambiguity

Ambiguity is caused by the fact that the grammar can derive the same sentence in at least two ways. This is not a particularly interesting cause, since it characterizes all ambiguity in general. We are interested in explaining to a grammar engineer what is wrong for a very particular grammar and sentence and how to possibly solve this particular issue. We are interested in the *root causes* of specific occurrences of choice nodes in parse forests.

For example, let us consider a particular grammar for the C programming language for which the sub-sentence "`{S * b;}`" is ambiguous. In one derivation it is a block of a single statement that multiplies variables `S` and `b`, in another it is a block of a single declaration of a pointer variable `b` to something of type `S`. From a language engineer's perspective, the causes of this ambiguous sentence are that:

- "$*$" is used both in the rule that defines multiplication, and in the rule that defines pointer types, *and*
- type names and variable names have the same lexical syntax, *and*
- blocks of code start with a possibly empty list of declarations and end with a possibly empty list of statements, *and*
- both statements and declarations end with "`;`".

The conjunction of all these causes explains us why there is an ambiguity. The removal of just one of them fixes it. In fact, we know that for C the ambiguity was fixed by introducing a disambiguator that reserves any declared type name from variable names using a symbol table at parse time, effectively removing the second cause.

We now define a *cause* of an ambiguity in a sub-sentence to be the existence of any edge that is in the parse tree of one alternative of an ambiguity node, but not in the other. In other words, each *difference* between two alternative parse trees in a forest is *one cause* of the ambiguity. For example, two parse tree edges differ if they represent the application of a different production rule, span a different part of the ambiguous sub-sentence, or are located at different heights in the tree.

We define an *explanation* of an ambiguity in a sentence to be the conjunction of all causes of ambiguity in a sentence. An explanation is a set of differences. We call it an explanation because an ambiguity exists if and only if all of its causes exist. A *solution* is any change to the grammar, addition of a disambiguation filter or use of a disambiguator that removes at least one of the causes.

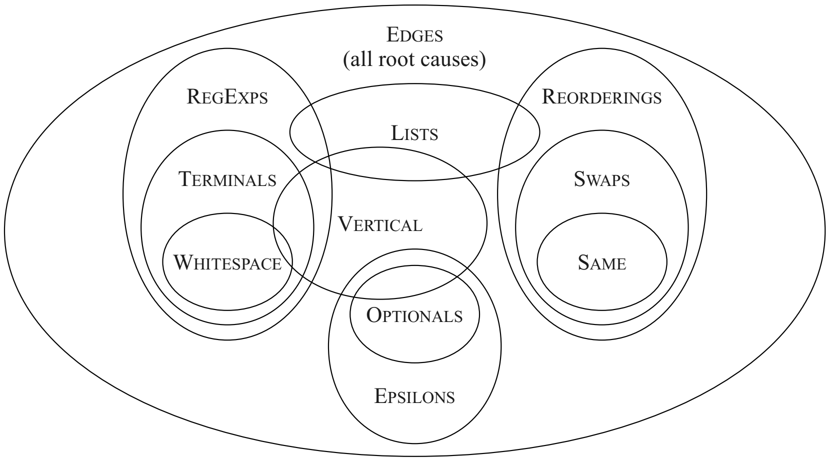


Fig. 3. Euler diagram showing the categorization of parse tree differences

Some causes of ambiguity may be solvable by the disambiguation methods defined in Section 2, some may not. Our goals are therefore to first explain the cause of ambiguity as concisely as possible, and then if possible propose a palette of applicable disambiguations. Note that even though the given common disambiguations have limited scope, disambiguation in general is always possible by writing a disambiguation filter in any computationally complete programming language.

3.1 Classes of Parse Tree Differences

Having defined ambiguity and the causes thereof, we can now categorize different kinds of causes into classes of differences between parse trees. The difference classes are the theory behind the workings of DR. AMBIGUITY (Section 5). Figure 3 summarizes the cause classes that we will identify in the following.

For completeness we should explain that ambiguity of CFGs is normally bisected into a class called HORIZONTAL ambiguity and a class called VERTICAL ambiguity [2,8,31]. VERTICAL contains all the ambiguity that causes parse forests that have two different production rules directly under a choice node. For instance, all edges of derivation sequences of form $\gamma A\delta \Rightarrow \gamma((\alpha) \diamond (\beta))\delta$ provided that $\alpha \neq \beta$ are in VERTICAL. VERTICAL clearly identifies a difference class, namely the trees with different edges directly under a choice node.

HORIZONTAL ambiguity is defined to be all the other ambiguity. HORIZONTAL does not identify any difference class, since it just implies that the two top rules are the same. Our previous example of ambiguity in a C grammar is an example of such ambiguity. We conclude that in order to obtain full explanations of ambiguity the HORIZONTAL/VERTICAL dichotomy is not detailed enough. VERTICAL provides only a partial explanation (a single cause), while HORIZONTAL provides no explanations at all.

We now introduce a number of difference classes with the intention of characterizing differences which can be solved by one of the aforementioned disambiguation methods. Each element in a different class points to a single cause of ambiguity. A particular disambiguation method may be applicable in the presence of elements in one or more of these classes.

The EDGES class is the universe of all difference classes. In EDGES are all single derivation steps (equivalent to edges in parse forests) that occur in one alternative but not in the other. If no such derivation steps exist, the two alternatives are exactly equal. Note that EDGES $=$ HORIZONTAL $\cup$ VERTICAL.

The TERMINALS class contains all parse tree edges to non-ε leafs that occur in one alternative but not in the other. If an explanation contains a difference in TERMINALS, we know that the alternatives have used different terminal tokens—or in the case of scannerless, different character classes—for the same sub-sentences. This is sometimes called *lexical ambiguity*. If no differences are in TERMINALS, we know that the terminals used in each alternative are equal.

The WHITESPACE class ($\subset$ TERMINALS) simply identifies the differences in TERMINALS that produce terminals consisting of nothing but spaces, tabs, newlines, carriage returns or linefeeds.

The REGEXPS class contains all edges of derivation steps that replace a non-terminal by a sentential form that generates a regular language, occurring in one derivation but not in the other, i.e. $A \Rightarrow (\rho)$ where ρ is a regular expression over terminals. Of course, TERMINALS $\subset$ REGEXPS. In character level grammars (scannerless [9]), the REGEXPS class often represents lexical ambiguity. Differences in REGEXPS may point to solutions such as **Reserve**, **Follow** and **Reject**, since longest match and keyword reservation are typical solution scenarios for ambiguity on the lexical level.

In the SWAPS class we put all edges that have a corresponding edge in the other alternative of which the source and target productions are equal but have swapped order. For instance, the lower edges in the parse tree fragment $((E*E)+E) \diamond (E*(E+E))$ are in SWAPS. If all differences are in SWAPS, the set of rules used in the derivations of both alternatives are the same and each rule is applied the same number of times—only their order of application is different.

The SAME class is the subset of edges in SWAPS that have the same source and target productions. In this case, the only difference between two corresponding edges are the substrings they span. For instance, the lower edges in the parse tree fragment $((E+E)+E) \diamond (E+(E+E))$ are in SAME. Differences in this class typically require **Associativity** solutions.

The REORDERINGS class generalizes SWAPS with more than two rules to permute. This may happen when rules are not directly recursive, but mutually recursive in longer chains. Differences in REORDERINGS or SWAPS obviously suggest a **Priority** solution, but especially for non-directly recursive derivations **Priority** will not work. For example, the notorious "dangling else" issue [1] generates differences in application order of mutually recursive statements and lists of statements. For some grammars, a difference in REORDERINGS may also imply a difference in VERTICAL, i.e. a choice between an `if` with an `else` and one without. In this case a **Preference** solution would work. Some grammars (e.g. the IBM COBOL VS2 standard) only have differences in HORIZONTAL and REORDERINGS. In this case a **Follow** solution may prevent the use of the `if` without the `else` if there is an `else` to be parsed. Note that the **Offside** solution is an alternative method to remove ambiguity caused by REORDERINGS. Apparently, we need even smaller classes of differences before we can be more precise about suggesting a solution.

The LISTS class contains differences in the length of certain lists between two alternatives. For instance, we consider rules $L \rightarrow LE$ and observe differences in the amount of times these rules are applied by the derivation steps in each alternative. More precisely, for any L and E with the rule $L \rightarrow LE$ we find chains of edges for derivation sequences $\alpha L \beta \Rightarrow \alpha LE \beta \Rightarrow\Rightarrow^* \alpha LE^+ \beta$, and compute their length. The edges of such chains of different lengths in the two alternatives are members of LISTS. Examples of ambiguities caused by LISTS are those caused by not having "longest match" behavior: an identifier "`aa`" generated using the rules $I \rightarrow a$ and $I \rightarrow I\ a$ may be split up in two shorter identifiers "`a`" and "`a`" in another alternative. We can say that LISTS $\cap$ REGEXPS $\neq \emptyset$.

Note that differences in LISTS$\cap$REORDERINGS indicate a solution towards **Follow** or **Offside** for they flag issues commonly seen in dangling constructs. On the other hand a difference in LISTS $\setminus$ REORDERINGS indicates that there must be another important difference to explain the ambiguity. The "'`{S * a}`'" ambiguity in C is of that sort, since the length of declaration and statement lists differ between the two alternatives, while also differences in TERMINALS are necessary.

The EPSILONS class contains all edges to ε leaf nodes that only occur in one of the alternatives. They correspond to derivation steps $\alpha A \beta \Rightarrow \alpha()\beta$, using $A \rightarrow \varepsilon$. All cyclic derivations are caused by differences in EPSILONS because one of the alternatives of a cyclic ambiguity must derive the empty sub-sentence, while the other eventually loops back. However, differences in EPSILONS may also cause other ambiguity than cyclic derivations.

The OPTIONALS class ($\subset$ EPSILONS) contains all edges of a derivation step $\alpha A \beta \Rightarrow \alpha()\beta$ that only exist in one alternative, while a corresponding edge of $\delta A \zeta \Rightarrow \delta(\gamma)\zeta$ only exists in the other alternative. Problems that are solved using longest match (**Follow**) are commonly caused by optional whitespace for example.

4 Diagnosing Ambiguity

We provide an overview of the architecture and the algorithms of DR. AMBIGUITY in this section. In Section 5 we demonstrate its output on example parse forests for an ambiguous Java grammar.

4.1 Architecture

Figure 4 shows an overview of our diagnostics tool: DR. AMBIGUITY. We start from the parse forest of an ambiguous sentence that is either encountered by a language engineer or produced by a static ambiguity detection tool like AMBIDEXTER. Then, either the

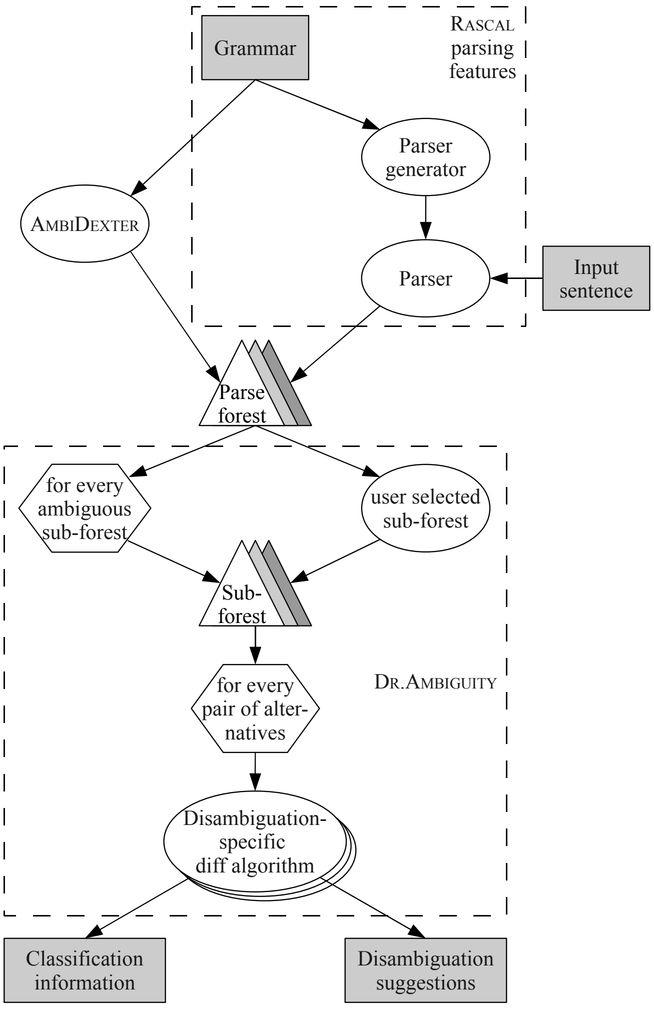

Grammar
RASCAL parsing features
Parser generator
AMBIDEXTER
Parser
Input sentence
Parse forest
for every ambiguous sub-forest
user selected sub-forest
Sub-forest
DR.AMBIGUITY
for every pair of alter-natives
Disambiguation-specific diff algorithm
Classification information
Disambiguation suggestions

Fig. 4. Contextual overview (input/output) of DR. AMBIGUITY

user points at a specific sub-sentence[2], or DR. AMBIGUITY finds all ambiguous sub-sentences (e.g. choice nodes) and iterates over them. For each choice node, the tool then generates all unique combinations of two children of the choice node and applies a number of specialized diff algorithms to them.

Conceptually there exists one diff algorithm per disambiguation method (Section 2). However, since some methods may share intermediate analyses there is some additional intermediate stages and some data-dependency that is not depicted in Figure 4. These in-

[2] We use Eclipse IMP [14] as a platform for generating editors for programming languages defined using RASCAL [23]. IMP provides contextual pop-up menus.

termediate stages output information messages about the larger difference classes that are to be analyzed further if possible. This output is called "Classification Information" in Figure 4. The other output, called "Disambiguation Suggestions" is a list of specific disambiguation solutions (with reference to specific production rules from the grammar).

If no specific or meaningful disambiguation method is proposed the classification information will provide the user with useful information on designing an ad-hoc disambiguation.

DR. AMBIGUITY is written in the RASCAL domain specific programming language [23]. This language is specifically targeted at analysis, transformation, generation and visualization of source code. Parse trees are a built-in data-type which can be queried using (higher order) pattern matching, visiting and set, list and map comprehension facilities. To understand some of the RASCAL snippets in this section, please familiarize yourself with this definition for parse trees (as introduced by [37]):

```
data Tree
  = appl(Production prod, list[Tree] args) // production nodes
  | amb(set[Tree] alternatives)            // choice nodes
  | char(int code);                        // terminal leaves
data Production
  = prod(Symbol lhs, list[Symbol] rhs, Attributes attributes); // rules
```

DR. AMBIGUITY, in total, is 250 lines of RASCAL code that queries and traverses terms of this parse tree format. The count includes source code comments. It is slow on big parse forests[3], which is why the aforementioned user-selection of specific sub-sentences is important.

4.2 Algorithms

Here we show some of the actual source code of DR. AMBIGUITY.

First, the following two small functions iterate over all (deeply nested) choice nodes (`amb`) and over all possible pairs of alternatives. This code uses deep match (`/`), set matching, and set or list comprehensions. Note that the match operator (`:=`) iterates over all possible matches of a value against a pattern, thus generating all different bindings for the free variables in the pattern. This feature is used often in the implementation of DR. AMBIGUITY.

```
list[Message] diagnose(Tree t) {
  return [findCauses(x) | x <- {a | /a:amb(_) := t}];
}
list[Message] findCauses(Tree a) {
  return [findCauses(x, y) | {x, y, _*} := a.alternatives];
}
```

The following functions each implement one of the diff algorithms from Figure 4. The following two (slightly simplified[4]) functions detect opportunities to apply priority or associativity disambiguations.

[3] The current implementation of RASCAL lacks many trivial optimizations.

[4] We have removed references to location information that facilitates IDE features.

```
list[Message] priorityCauses(Tree x, Tree y) {
  if (/appl(p,[appl(q,_),_*]) := x,
      /t:appl(q,[_*,appl(p,_)]) := y, p != q) {
      return [error("You␣might␣add␣this␣priority␣rule:␣<p>␣\>␣<q>")
             ,error("You␣might␣add␣this␣associativity␣group:␣left␣(<p>␣|␣<q>)")];
  }
  return [];
}
list[Message] associativityCauses(Tree x, Tree y) {
  if (/appl(p,[appl(p,_),_*]) := x, /Tree t:appl(p,[_*,appl(p,_)]) := y) {
    return [error("You␣might␣add␣this␣associativity␣declaration:␣left␣<p>")];
  }
  return [];
}
```

Both functions “simultaneously” search through the two alternative parse trees `p` and `q`, detecting a vertical swap of two different rules `p` and `q` (priority) or a horizontal swap of the same rule `p` under itself (associativity).

This slightly more involved function detects dangling-else and proposes a follow restriction as a solution:

```
list[Message] danglingCauses(Tree x, Tree y) {
  if (appl(p,/appl(q,_)) := x, appl(q,/appl(p,_)) := y) {
    return danglingOffsideSolutions(x, y)
         + danglingFollowSolutions(x, y);
  }
  return [];
}
list[Message] danglingFollowSolutions(Tree x, Tree y) {
  if (prod(_, rhs, _) := x.prod,
      prod(_, [prefix*, _, l:lit(_), more*], _) := y.prod,
      rhs == prefix) {
    return [error("You␣might␣add␣a␣follow␣restriction␣for␣<l>␣on:␣<x.prod>")];
  }
  return [];
}
```

The function `danglingCauses` detects re-orderings of arbitrary depth, after which the outermost productions are compared by `danglingFollowSolutions` to see if one production is a prefix of the other.

DR. AMBIGUITY currently contains 10 such functions, and we will probably add more. Since they all employ the same style —(a) simultaneous deep match, (b) production comparison and (c) construction of a feedback message— we have not included more source code[5].

[5] The source code is available at
`http://svn.rascal-mpl.org/rascal/trunk/src/org/rascalmpl/library/Ambiguity.rsc`

4.3 Discussion on Correctness

These diagnostics algorithms are typically wrong if one of the following four errors is made:

- no suggestion is given, even though the ambiguity is of a quite common kind;
- the given suggestion does not resolve any ambiguity;
- the given suggestion removes both alternatives from the forest, resulting in an empty forest (i.e., it removes the sentence from the language and is thus not language preserving);
- the given suggestion removes the proper derivation, but also unintentionally removes sentences from the language.

We address the first threat by demonstrating DR. AMBIGUITY on Java in Section 5. However, we do believe that the number of detection algorithms is open in principle. For instance, for any disambiguation method that characterizes a specific way of solving ambiguity we may have a function to analyze the characteristic kind of difference. As an "expert tool", automating proposals for common solutions in language design, we feel that an open-ended solution is warranted. More disambiguation suggestion algorithms will be added as more language designs are made. Still, in the next section we will demonstrate that the current set of algorithms is complete for all disambiguations applied to a scannerless definition of Java 5 [11], which actually uses all disambiguations offered by SDF2.

For the second and third threats, we claim that no currently proposed solution removes both alternatives and all proposed solutions remove at least one. This is the case because each suggestion is solely deduced from a *difference* between two alternatives, and each disambiguation removes an artifact that is only present in one of the alternatives. We are considering to actually prove this, but only after more usability studies.

The final threat is an important weakness of DR. AMBIGUITY, inherited from the strength of the given disambiguation solutions. In principle and in practice, the application of rejects, follow restrictions, or semantic actions in general renders the entire parsing process stronger than context-free. For example, using context-free grammars with additional disambiguations we may decide language membership of many non-context-free languages. On the one hand, this property is beneficial, because we want to parse programming languages that have no or awkward context-free grammars. On the other hand, this property is cumbersome, since we can not easily predict or characterize the effect of a disambiguation filter on the accepted set of sentences.

Only in the SWAPS class, and its sub-classes we may be (fairly) confident that we do not remove unforeseen sentences from a language by introducing a disambiguation. The reason is that if one of the alternatives is present in the forest, the other is guaranteed to be also there. The running assumption is that the other derivation has not been filtered by some other disambiguation. We might validate this assumption automatically in many cases. So, application of priority and associativity rules suggested by DR. AMBIGUITY are safe if no other disambiguations are applied.

Table 1. Disambiguations applied in the Java 5 grammar [11]

Disambiguations	**Grammar snippet (Rascal notation)**
7 levels of expression priority	`Expr = Expr "++"`
	`     > "++" Expr`
1 father/child removal	`MethodSpec = Expr callee "." TypeArgs? Id {`
	`                if (callee is ExprName) filter; }`
9 associativity groups	`Expr = left ( Expr "+" Expr`
	`             | Expr "-" Expr )`
10 rejects	`ID = ( [$A-Z_a-z] [$0-9A-Z_a-z]* ) \ Keywords`
30 follow restrictions	`"+" = [\+] !>> [\+]`
4 vertical preferences	`Stm = @prefer "if" "(" Expr ")" Stm`
	`    | "if" "(" Expr ")" Stm "else" Stm`

5 Demonstration

In this section we evaluate the effectiveness of DR. AMBIGUITY as a tool. We applied DR. AMBIGUITY to a scannerless (character level) grammar for Java [10,11]. This well tested grammar was written in SDF2 by Bravenboer et al. and makes ample use of its disambiguation facilities. For the experiment here we automatically transformed the SDF2 grammar to RASCAL's EBNF-like form.

Table 1 summarizes which disambiguations were applied in this grammar. RASCAL supports all disambiguation features of SDF2, but some disambiguation filters are implemented as libraries rather than built-in features. The `@prefer` attribute is interpreted by a library function for example. Also, in SDF2 one can (mis)use a non-transitive priority to remove a direct father/child relation from the grammar. In Rascal we use a semantic action for this.

5.1 Evaluation Method

DR. AMBIGUITY is effective if it can explain the existence of a significant amount of choice nodes in parse forests and proposes the right fixes. We measure this effectiveness in terms of precision and recall. DR. AMBIGUITY has high precision if it does not propose too many solutions that are useless or meaningless to the language engineer. It has high recall if it finds all the solutions that the language engineer deems necessary.

Our evaluation method is as follows:

- The set of disambiguations that Bravenboer applied to his Java grammar is our "golden standard".
- The disambiguations in the grammar are selectively removed, which results in different ambiguous versions of the grammar. New parsers are generated for each version.
- An example Java program is parsed with each newly generated parser. The program is unambiguous for the original grammar, but becomes ambiguous for each altered version of the grammar.

Table 2. Precision/Recall results for each experiment, including (P)riority, (A)ssociativity, (R)eject, (F)ollow restrictions, A(c)tions filtering edges, A(v)oid/prefer suggestions, and (O)ffside rule. For each experiment, the figures of the removed disambiguation are highlighted.

	Diagnoses								
Experiment	**P**	**A**	**R**	**F**	**c**	**v**	**O**	**Precision**	**Recall**
1. Remove priority between `"*"` and `"+"`	**1**	1	0	0	0	1	0	33%	100%
2. Remove associativity for `"+"`	0	**1**	0	0	0	0	0	100%	100%
3. Remove reservation of `true` keyword from `ID`	0	0	**1**	0	0	1	0	50%	100%
4. Remove longest match for identifiers	0	0	0	**6**	0	0	0	16%	100%
5. Remove package name vs. field access priority	0	0	0	0	**6**	1	0	14%	100%
6. Remove vertical preference for dangling else	0	0	0	1	14	**1**	1	7%	100%
7. *All the above changes at the same time*	**1**	**2**	**1**	**7**	**20**	**4**	1	17%	100%

– We measure the total amount and which kinds of suggestions are made by DR. AMBIGUITY for the parse forests of each grammar version, and compute the precision and recall.

Precision is computed by $\frac{|\text{FOUNDDISAMBIGUATIONS} \cap \text{REMOVEDDISAMBIGUATIONS}|}{|\text{FOUNDDISAMBIGUATIONS}|} \times 100\%$. We expect low precision, around 50%, because each particular ambiguity often has many different solution types. Low precision is not necessarily a bad thing, provided the total amount of disambiguation suggestions remains human-checkable.

Recall is computed by $\frac{|\text{FOUNDDISAMBIGUATIONS} \cap \text{REMOVEDDISAMBIGUATIONS}|}{|\text{REMOVEDDISAMBIGUATIONS}|} \times 100\%$. From this number we see how much we have missed. We expect the recall to be 100% in our experiments, since we designed our detection methods specifically for the disambiguation techniques of SDF2.

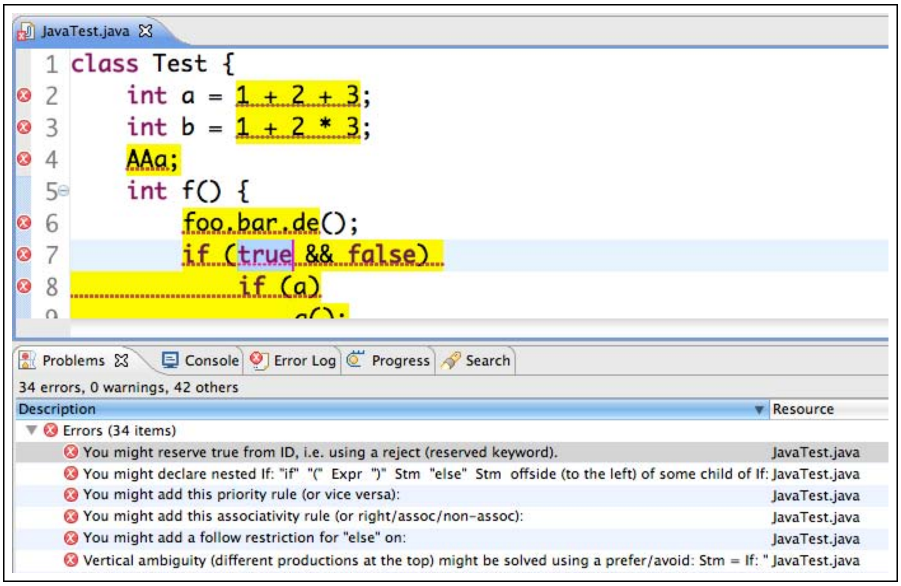
5.2 Results

Table 2 contains the results of measuring the precision and recall on a number of experiments. Each experiment corresponds to a removal of one or more disambiguation constructs and the parsing of a single Java program file that triggers the introduced ambiguity.

Table 2 shows that we indeed always find the removed disambiguation among the suggestions. Also, we always find more than one suggestion (the second experiment is the only exception).

The dangling-else ambiguity of experiment 6 introduces many small differences between two alternatives, which is why many (arbitrary) semantic actions are proposed to solve these. We may learn from this that semantic actions need to be presented to the language engineer as a last resort. For these disambiguations the risk of collateral damage (a non-language preserving disambiguation) is also quite high.

The final experiment tests whether the simultaneous analysis of different choice nodes that are present in a parse forest may lead to a loss of precision or recall. The results show that we find exactly the same suggestions. Also, as expected the precision



JavaTest.java

```
class Test {
    int a = 1 + 2 + 3;
    int b = 1 + 2 * 3;
    AAa;
    int f() {
        foo.bar.de();
        if (true && false)
            if (a)
```

Problems | Console | Error Log | Progress | Search

34 errors, 0 warnings, 42 others

Description	Resource
Errors (34 items)	
You might reserve true from ID, i.e. using a reject (reserved keyword).	JavaTest.java
You might declare nested If: "if" "(" Expr ")" Stm "else" Stm offside (to the left) of some child of If:	JavaTest.java
You might add this priority rule (or vice versa):	JavaTest.java
You might add this associativity rule (or right/assoc/non-assoc):	JavaTest.java
You might add a follow restriction for "else" on:	JavaTest.java
Vertical ambiguity (different productions at the top) might be solved using a prefer/avoid: Stm = If: "	JavaTest.java

Fig. 5. DR. AMBIGUITY reports diagnostics in the RASCAL language workbench

of such an experiment is very low. Note however, that DR. AMBIGUITY reports each disambiguation suggestion per choice node, and thus the precision is usually perceived per choice node and never as an aggregated value over an entire source file. Figure 5 depicts how DR. AMBIGUITY may report its output.

5.3 Discussion

We have demonstrated the effectiveness of DR. AMBIGUITY for only one grammar. Moreover this grammar already contained disambiguations that we have removed, simultaneously creating a representative case and a golden standard.

We may question whether DR. AMBIGUITY would do well on grammars that have not been written with any disambiguation construct in mind. We may also question whether DR. AMBIGUITY works well on completely different grammars, such as for COBOL or PL/I. More experimental evaluation is warranted. Nevertheless, this initial evaluation based on Java looks promising and does not invalidate our approach.

Regarding the relatively low precision, we claimed that this is indeed wanted in many cases. The actual resolution of an ambiguity is a language design question. DR. AMBIGUITY should not a priori promote a particular disambiguation over another well known disambiguation. For example, reverse engineers have a general dislike of the offside rule because it complicates the construction of a parser, while the users of a domain specific language may applaud the sparing use of bracket literals.

6 Conclusions

We have presented theory and practice of automatically diagnosing the causes of ambiguity in context-free grammars for programming languages and of proposing disambiguation solutions. We have evaluated our prototype implementation on an actively used and mature grammar for Java 5, to show that DR. AMBIGUITY can indeed propose the proper disambiguations.

Future work on this subject includes further extension, further usability study and finally proofs of correctness. To support development of front-ends for many programming languages and domain specific languages, we will include DR. AMBIGUITY in releases of the RASCAL IDE (a software language workbench).

References

1. Aho, A., Sethi, R., Ullman, J.: Compilers. Principles, Techniques and Tools. Addison-Wesley (1986)
2. Altman, T., Logothetis, G.: A note on ambiguity in context-free grammars. Inf. Process. Lett. 35(3), 111–114 (1990)
3. Aycock, J., Horspool, R.N.: Faster Generalized LR Parsing. In: Jähnichen, S. (ed.) CC 1999. LNCS, vol. 1575, pp. 32–46. Springer, Heidelberg (1999)
4. Basten, H.J.S.: Tracking Down the Origins of Ambiguity in Context-Free Grammars. In: Cavalcanti, A., Deharbe, D., Gaudel, M.-C., Woodcock, J. (eds.) ICTAC 2010. LNCS, vol. 6255, pp. 76–90. Springer, Heidelberg (2010)
5. Basten, H.J.S., Vinju, J.J.: Faster ambiguity detection by grammar filtering. In: Brabrand, C., Moreau, P.E. (eds.) Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications (LDTA 2010), pp. 5:1–5:9. ACM (2010)
6. Begel, A., Graham, S.L.: XGLR–an algorithm for ambiguity in programming languages. Science of Computer Programming 61(3), 211–227 (2006); Special Issue on The Fourth Workshop on Language Descriptions, Tools, and Applications (LDTA 2004)
7. Bouwers, E., Bravenboer, M., Visser, E.: Grammar engineering support for precedence rule recovery and compatibility checking. ENTCS 203(2), 85–101 (2008); Proceedings of the Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA 2007)
8. Brabrand, C., Giegerich, R., Møller, A.: Analyzing ambiguity of context-free grammars. Sci. Comput. Program. 75(3), 176–191 (2010)
9. van den Brand, M., Scheerder, J., Vinju, J.J., Visser, E.: Disambiguation Filters for Scannerless Generalized LR Parsers. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 143–158. Springer, Heidelberg (2002)
10. Bravenboer, M., Tanter, E., Visser, E.: Declarative, formal, and extensible syntax definition for AspectJ. SIGPLAN Not. 41, 209–228 (2006)
11. Bravenboer, M., Vermaas, R., de Groot, R., Dolstra, E.: Java-front: Java syntax definition, parser, and pretty-printer. Tech. rep., http://www.program-transformation.org (2011), http://www.program-transformation.org/Stratego/JavaFront
12. 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.R. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 157–172. Springer, Heidelberg (2005)

13. Cantor, D.G.: On the ambiguity problem of Backus systems. Journal of the ACM 9(4), 477–479 (1962)
14. Charles, P., Fuhrer, R.M., Sutton Jr., S.M., Duesterwald, E., Vinju, J.: Accelerating the creation of customized, language-specific IDEs in eclipse. In: Arora, S., Leavens, G.T. (eds.) Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009 (2009)
15. Chomsky, N., Schützenberger, M.: The algebraic theory of context-free languages. In: Braffort, P. (ed.) Computer Programming and Formal Systems, pp. 118–161. North-Holland, Amsterdam (1963)
16. Earley, J.: An efficient context-free parsing algorithm. Commun. ACM 13, 94–102 (1970)
17. Economopoulos, G.R.: Generalised LR parsing algorithms. Ph.D. thesis, Royal Holloway, University of London (August 2006)
18. Floyd, R.W.: On ambiguity in phrase structure languages. Communications of the ACM 5(10), 526–534 (1962)
19. Ford, B.: Parsing expression grammars: a recognition-based syntactic foundation. SIGPLAN Not. 39, 111–122 (2004)
20. Ginsburg, S., Harrison, M.A.: Bracketed context-free languages. Journal of Computer and System Sciences 1(1), 1–23 (1967)
21. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The syntax definition formalism SDF - reference manual. SIGPLAN Notices 24(11), 43–75 (1989)
22. 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)
23. Klint, P., van der Storm, T., Vinju, J.: EASY Meta-Programming with Rascal. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) Generative and Transformational Techniques in Software Engineering III. LNCS, vol. 6491, pp. 222–289. Springer, Heidelberg (2011)
24. Klint, P., Visser, E.: Using filters for the disambiguation of context-free grammars. In: Pighizzini, G., San Pietro, P. (eds.) Proc. ASMICS Workshop on Parsing Theory, pp. 1–20. Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano, Milano, Italy (1994)
25. Lämmel, R., Verhoef, C.: Semi-automatic grammar recovery. Softw. Pract. Exper. 31, 1395–1448 (2001)
26. Landin, P.J.: The next 700 programming languages. Commun. ACM 9, 157–166 (1966)
27. Moonen, L.: Generating robust parsers using island grammars. In: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE 2001), p. 13. IEEE Computer Society, Washington, DC (2001)
28. Parr, T., Fisher, K.: LL(*): the foundation of the ANTLR parser generator. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, pp. 425–436. ACM, New York (2011)
29. Rekers, J.: Parser Generation for Interactive Environments. Ph.D. thesis, University of Amsterdam (1992)
30. Salomon, D.J., Cormack, G.V.: Scannerless NSLR(1) parsing of programming languages. In: Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation, PLDI 1989, pp. 170–178. ACM (1989)
31. Schröer, F.W.: AMBER, an ambiguity checker for context-free grammars. Tech. rep., compilertools.net (2001), http://accent.compilertools.net/Amber.html
32. Schröer, F.W.: ACCENT, a compiler compiler for the entire class of context-free grammars, 2nd edn. Tech. rep., compilertools.net (2006), http://accent.compilertools.net/Accent.html

33. Scott, E.: SPPF-style parsing from earley recognisers. ENTCS 203, 53–67 (2008)
34. Scott, E., Johnstone, A.: GLL parsing. ENTCS 253(7), 177–189 (2010); Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009)
35. Tomita, M.: Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems. Kluwer Academic Publishers (1985)
36. Vinju, J.J.: SDF disambiguation medkit for programming languages. Tech. Rep. SEN-1107, Centrum Wiskunde & Informatica (2011), `http://oai.cwi.nl/oai/asset/18080/18080D.pdf`
37. Visser, E.: Syntax Definition for Language Prototyping. Ph.D. thesis, Universiteit van Amsterdam (1997)

Ambiguity Detection: Scaling to Scannerless

Hendrikus J.S. Basten, Paul Klint, and Jurgen J. Vinju

Centrum Wiskunde & Informatica
Science Park 123, 1098 XG Amsterdam, The Netherlands

Abstract. Static ambiguity detection would be an important aspect of language workbenches for textual software languages. However, the challenge is that automatic ambiguity detection in context-free grammars is undecidable in general. Sophisticated approximations and optimizations do exist, but these do not scale to grammars for so-called "scannerless parsers", as of yet. We extend previous work on ambiguity detection for context-free grammars to cover disambiguation techniques that are typical for scannerless parsing, such as longest match and reserved keywords. This paper contributes a new algorithm for ambiguity detection in character-level grammars, a prototype implementation of this algorithm and validation on several real grammars. The total run-time of ambiguity detection for character-level grammars for languages such as C and Java is significantly reduced, without loss of precision. The result is that efficient ambiguity detection in realistic grammars is possible and may therefore become a tool in language workbenches.

1 Introduction

1.1 Background

Scannerless generalized parsers [7], generated from character-level context-free grammars, serve two particular goals in textual language engineering: parsing legacy languages and parsing language embeddings. We want to parse legacy languages when we construct reverse engineering and reengineering tools to help mitigating cost-of-ownership of legacy source code. The syntax of legacy programming languages frequently does not fit the standard scanner-parser dichotomy. This is witnessed by languages that do not reserve keywords from identifiers (PL/I) or do not always apply "longest match" when selecting a token class (Pascal). For such languages we may generate a scannerless generalized parser that will deal with such idiosyncrasies correctly.

Language embeddings need different lexical syntax for different parts of a composed language. Examples are COBOL with embedded SQL, or Aspect/J with embedded Java. The comment conventions may differ, different sets of identifiers may be reserved as keywords and indeed identifiers may be comprised of different sets of characters, depending on whether the current context is the "host language" or the embedded "guest language". Language embeddings are becoming increasingly popular, possibly due to the belief that one should select the right tool for each job. A character-level grammar can be very convenient to implement a parser for such a combined language [8]. The reason is that the particular nesting of non-terminals between the host language and the guest language defines where the different lexical syntaxes are applicable. The lexical ambiguity introduced by the language embedding is therefore a non-issue for a scannerless parser. There is no need to program state switches in a scanner [15], to use scanner non-determinism [2], or to use any other kind of (ad-hoc) programming solution.

Using a character-level grammar and a generated scannerless parser results in more declarative BNF grammars which may be maintained more easily than partially handwritten parsers [11]. It is, however, undeniable that character-level grammars are more complex than classical grammars since all lexical aspects of a language have to be specified in full detail. The character-level grammar contains more production rules, which may contain errors or introduce ambiguity. In the absence of lexical disambiguation heuristics, such as "prefer keywords" and "longest match", a character-level grammar may contain many ambiguities that need resolving. Ergo, character-level grammars lead to more declarative grammar specifications but increase the risk of ambiguities and makes automated ambiguity detection harder.

1.2 Contributions and Roadmap

We introduce new techniques for scaling ambiguity detection methods to the complexity that is present in character-level grammars for real programming languages. Our point of departure is a fast ambiguity detection framework that combines a grammar approximation stage with a sentence generation stage [5]. The approximation is used to split a grammar into a set of rules that certainly do not contribute to ambiguity and a set that might. The latter is then fed to a sentence generator to obtain a clear and precise ambiguity report. We sketch this global framework (Section 2) and then describe our baseline algorithm (Section 4). The correctness of this framework has been established in [4] and is not further discussed here.

We present several extensions to the baseline algorithm to make it suitable for character-level grammars (Section 5). First, we consider character classes as shiftable symbols, instead of treating every character as a separate token. This is necessary to deal with the increased lexical complexity of character-level grammars. Second, we make use of disambiguation filters [7] to deal with issues such as keyword reservation and longest match. These filters are used for precision improvements at the approximation stage, and also improve the run-time efficiency of the sentence generation stage by preventing spurious explorations of the grammar. Third, we use grammar unfolding as a general optimization technique (Section 6). This is necessary for certain character-level grammars but is also generally applicable. At a certain cost, it allows us to more effectively identify the parts of a grammar that do not contribute to ambiguity.

We have selected a set of real character-level grammars and measure the speed, footprint and accuracy of the various algorithms (Section 7). The result is that the total cost of ambiguity detection is dramatically reduced for these real grammars.

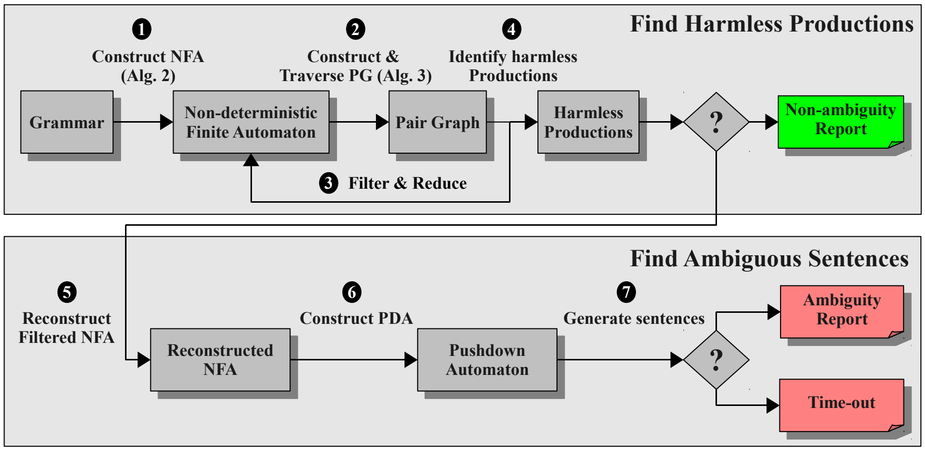


Fig. 1. Baseline architecture for fast ambiguity detection

2 The Ambiguity Detection Framework

2.1 The Framework

Our starting point is an ambiguity detection framework called AMBIDEXTER [5], which combines an extension of the approximative Noncanonical Unambiguity Test [13] with an exhaustive sentence generator comparable to [14]. The former is used to split a grammar into a set of harmless rules and a set of rules that may contribute to ambiguity. The latter is used to generate derivations based on the potentially ambiguous rules and produce understandable ambiguity reports.

Figure 1 displays the architecture of the baseline algorithm which consists of seven steps, ultimately resulting in a non-ambiguity report, an ambiguity report, or a time-out.

1. In step ❶ the grammar is bracketed, starting and ending each rule with a unique terminal. The language of the bracketed grammar represents all parse trees of the original grammar. In this same step an NFA is constructed that over-approximates the language of the bracketed grammar. This NFA allows us to find strings with multiple parse trees, by approximation, but in finite time.
2. In step ❷ a data-structure called a Pair Graph (PG) is constructed from the NFA. This PG represents all pairs of two different paths through the NFA that produce the same sentence, i.e., potentially ambiguous derivations. During construction, the PG is immediately traversed to identify the part of the NFA that is covered by the potentially ambiguous derivations.
3. In step ❸ we filter the uncovered parts from the NFA and clean up dead ends. This might filter potentially ambiguous derivations from the NFA that are actually false positives, so we reconstruct the PG again to find more uncovered parts. This process is repeated until the NFA cannot be reduced any further.
4. In step ❹ we use the filtered NFA to identify harmless productions. These are the productions that are not used anymore in the NFA. If the NFA is completely filtered then all productions are harmless and the grammar is unambiguous.

5. In step ❺ we prepare the filtered NFA to be used for sentence generation. Due to the removal of states not all paths produce terminal only sentences anymore. We therefore reconstruct the NFA by adding new terminal producing paths.
   In our original approach we generated sentences based on the remaining potentially harmful productions. However, by immediately using the filtered NFA we retain more precision, because the NFA is a more precise description of the potentially ambiguous derivations than a reconstructed grammar.
6. In step ❻ we convert the NFA into a pushdown automaton (PDA) which enables faster sentence generation in the next step.
7. The final step (❼) produces ambiguous strings, including their derivations, to report to the user. This may not terminate, since most context-free grammars generate infinite languages; we need to stop after a certain limited time. All ambiguity that was detected before the time limit is reported to the user.

It was shown in [4] that the calculations employed in this architecture are correct, and in [5] that indeed the efficiency of ambiguity detection can be improved considerably by first filtering harmless productions. However, the baseline algorithm is not suitable for character-level grammars since it is unable to handle their increased complexity and it will still find ambiguities that are already solved. It can even lead to incorrect results because it cannot deal with the non-context-free behaviour of follow restrictions. In this paper we identify several opportunities for optimization and correction:

– We filter nodes and edges in the NFA and PG representations in order to make use of disambiguation information that is found in character-level grammars (Section 5).
– We "unfold" selected parts of a grammar to handle the increased lexical complexity of character-level grammars (Section 6).

For the sake of presentation we have separated the discussion of the baseline algorithm (Section 4), the filtering (Section 5), and the unfolding (Section 6), but it is important to note that these optimizations are not orthogonal.

2.2 Notational Preliminaries

A *context-free grammar* G is a four-tuple (N, T, P, S) where N is the set of non-terminals, T the set of terminals, P the set of productions over $N \times (N \cup T)^*$, and S is the start symbol. V is defined as $N \cup T$. We use $A, B, C, \ldots$ to denote non-terminals, $X, Y, Z, \ldots$ for either terminals or non-terminals, $u, v, w, \ldots$ for sentences: strings of T^*, and $\alpha, \beta, \gamma, \ldots$ for sentential forms: strings over V^*.

A production (A, α) in P is written as $A \to \alpha$. A grammar is augmented by adding an extra non-terminal symbol S', a terminal symbol $\$$ and a production $S' \to S\$$, and making S' the start symbol. We use the function $\text{pid} : P \to \mathbb{N}$ to relate each production to a unique number. An *item* indicates a position in a production rule with a dot, for instance as $S \to A{\bullet}BC$. We use I to denote the set of all items of G.

The relation $\Longrightarrow$ denotes derivation. We say $\alpha B\gamma$ directly derives $\alpha\beta\gamma$, written as $\alpha B\gamma \Longrightarrow \alpha\beta\gamma$ if a production rule $B \to \beta$ exists in P. The symbol $\Longrightarrow^*$ means "derives in zero or more steps". The *language* of G, denoted $\mathcal{L}(G)$, is the set of all sentences derivable from S. We use $\mathcal{S}(G)$ to denote the *sentential language* of G: the set of all sentential forms derivable from S.

From a grammar G we can create a *bracketed grammar* G_b by surrounding each production rule with unique bracket terminals [9]. The bracketed grammar of G is defined as $G_b = (N, T_b, P_b, S)$ where T_b is the set of terminals and brackets, defined as $T_b = T \cup T_{\langle} \cup T_{\rangle}$, $T_{\langle} = \{\langle_i \mid i \in \mathbb{N}\}$, $T_{\rangle} = \{\rangle_i \mid i \in \mathbb{N}\}$, and $P_b = \{A \to \langle_i \alpha \rangle_i \mid A \to \alpha \in P, i = \mathsf{pid}(A \to \alpha)\}$. V_b is defined as $N \cup T_b$. We use the function $\mathsf{bracketP}$ to map a bracket to its corresponding production, and $\mathsf{bracketN}$ to map a bracket to its production's left hand side non-terminal. They are defined as $\mathsf{bracketP}(\langle_i) = \mathsf{bracketP}(\rangle_i) = A \to \alpha$ iff $\mathsf{pid}(A \to \alpha) = i$, and $\mathsf{bracketN}(\langle_i) = \mathsf{bracketN}(\rangle_i) = A$ iff $\exists A \to \alpha \in P$, $\mathsf{pid}(A \to \alpha) = i$. A string in the language of G_b describes a parse tree of G. Therefore, if two unique strings exist in $\mathcal{L}(G_b)$ that become identical after removing their brackets, G is ambiguous.

3 Character-Level Grammars

Now we introduce character-level grammars as used for scannerless parsing. Character-level grammars differ from conventional grammars in various ways. They define their syntax all the way down to the character level, without separate token definitions. For convenience, sets of characters are used in the production rules, so-called *character classes*. Regular list constructs can be used to handle repetition, like in EBNF. Also, additional constructs are needed to specify the disambiguation that is normally done by the scanner, so called disambiguation filters [12]. Typical disambiguation filters for character-level grammars are follow restrictions and rejects [7]. Follow restrictions are used to enforce longest match of non-terminals such as identifiers and comments. Rejects are typically used for keyword reservation. Other commonly used disambiguation filters are declarations to specify operator priority and associativity, so these do not have to be encoded manually into the production rules.

3.1 Example

Figure 2 shows an excerpt of a character-level grammar, written in SDF [10,16]. The excerpt describes syntax for C-style variable declarations. A `Declaration` statement consists of a list of `Specifiers` followed by an `Identifier` and a semicolon, separated by whitespace (Rule 1). A `Specifier` is either a predefined type like `int` or `float`, or a user-defined type represented by an `Identifier` (Rule 4). At Rule 5 we see the use of the character class `[a-z]` to specify the syntax of `Identifier`.

The grammar contains both rejects and follow restrictions to disambiguate the lexical syntax. The `{reject}` annotation at Rule 6 declares that reserved keywords of the language cannot be recognized as an `Identifier`. The follow restriction statements at Rules 9–11 declare that any substring that is followed by a character in the range `[a-z]` cannot be recognized as an `Identifier` or keyword. This prevents the situation where a single `Specifier`, for instance an `Identifier` of two or more characters, can also be recognized as a list of multiple shorter `Specifiers`. Basically, the follow restrictions enforce that `Specifiers` should be separated by whitespace.

```
Declaration ::= Specifiers Ws? Identifier Ws? ";"      (1)
Specifiers  ::= Specifiers Ws? Specifier               (2)
Specifiers  ::= Specifier                              (3)
Specifier   ::= Identifier | "int" | "float" | ...     (4)

Identifier  ::= [a-z]+                                 (5)
Identifier  ::= Keyword  { reject }                    (6)
Keyword     ::= "int" | "float" | ...                  (7)
Ws          ::= [\ \t\n]+                              (8)

Identifier  -/- [a-z]                                  (9)
"int"       -/- [a-z]                                  (10)
"float"     -/- [a-z]                                  (11)
```

Fig. 2. Example character-level grammar for C-style declarations

3.2 Definition

We define a *character-level context-free grammar* $G^{\mathcal{C}}$ as the eight-tuple $(N, T, \mathcal{C}, P^{\mathcal{C}}, S, R_D, R_F, R_R)$ where $\mathcal{C} \subset N$ is the set of character classes over $\mathcal{P}(T)$, $P^{\mathcal{C}}$ the set of production rules over $N \times N^*$, R_D the set of derivation restrictions, R_F the set of follow restrictions, R_R the set of rejects.

A *character class* non-terminal is a finite set of terminals in T. For each of its elements it has an implicit production with a single terminal right hand side. We can write $\alpha C\beta \Longrightarrow \alpha c\beta$ iff $C \in \mathcal{C}$ and $c \in C$.

The *derivation restrictions* R_D restrict the application of productions in the context of others. They can be used to express priority and associativity of operators. We define R_D as a relation over $I \times P^{\mathcal{C}}$. Recall that we have defined I as the set of all items of a grammar. An element $(A \to \alpha{\bullet}B\gamma, B \to \beta)$ in R_D means that we are not allowed to derive a B non-terminal with production $B \to \beta$, if it originated from the B following the dot in the production $A \to \alpha B\gamma$.

The *follow restrictions* R_F restrict the derivation of substrings following a certain non-terminal. We define them as a relation over $N \times T^+$. An element (A, u) in this relation means that during the derivation of a string $\beta A\gamma$, γ can not be derived into a string of form $u\delta$.

The *rejects* R_R restrict the language of a certain non-terminal, by subtracting the language of another non-terminal from it. We define them as a relation over $N \times N$. An element (A, B) means that during the derivation of a string $\alpha A\beta$, A cannot be derived to a string that is also derivable from B.

4 Baseline Algorithm

In this section we explain the baseline algorithm for finding harmless production rules and ambiguous counter-examples. The presentation follows the steps shown in Figure 1. We will mainly focus on the parts that require extensions for character-level grammars, and refer to [4,5] for a complete description of the baseline algorithm. Algorithm 1 gives an overview of the first stage of finding harmless productions. All functions operate on a fixed input grammar $G = (N, T, P, S)$ to which they have global read access.

Algorithm 1. Base algorithm for filtering the NFA and finding harmless productions.

```
function FIND-HARMLESS-PRODUCTIONS() =
  $(Q, R)$ = BUILD-NFA()
  do
    nfasize = $|Q|$
    $Q_a$ = TRAVERSE-PATH-PAIRS$(Q, R)$ // returns items used on conflicting path pairs
    $(Q, R)$ = FILTER-NFA$(Q, R, Q_a)$ // removes unused items and prunes dead ends
  while nfasize $\neq |Q|$
  return $P \setminus$ USED-PRODUCTIONS$(Q)$
```

4.1 Step 1: NFA Construction

The first step of the baseline algorithm is to construct the NFA from the grammar. It is defined by the tuple (Q, R) where Q is the set of states and R is the transition relation over $Q \times V_b \times Q$. Edges in R are denoted by $Q \xmapsto{V_b} Q$. The states of the NFA are the items of G. The start state is $S' \to \bullet S\$$ and the end state is $S' \to S\$\bullet$. There are three types of transitions:

- *Shifts* of (non-)terminal symbols to advance to a production's next item,
- *Derives* from items with the dot before a non-terminal to the first item of one of the non-terminal's productions, labeled over $T_\langle$,
- *Reduces* from items with the dot at the end, to items with the dot after the non-terminal that is at the first item's production's left hand side, labeled over $T_\rangle$.

Algorithm 2 describes the construction of the NFA from G. First, the set of states Q is composed from the items of G. Then the transitions in R are constructed, assuming only items in Q are used. Lines 2–4 respectively build the shift, derive and reduce transitions between the items of G.

Intuitively, the NFA resembles an LR(0) parse automaton before the item closure. The major differences are that also shifts of non-terminals are allowed, and that the NFA has — by definition — no stack. The LR(0) pushdown automaton uses its stack to determine the next reduce action, but in the NFA all possible reductions are allowed. Its language is therefore an overapproximation of the set of parse trees of G. However, the shape of the NFA does allow us to turn it into a pushdown automaton that only generates valid parse trees of G. We will do this later on in the sentence generation stage.

Without a stack the NFA can be searched for ambiguity in finite time. Two paths through it that shift the same sequence of symbols in V, but different bracket symbols in T_b, represent a possible ambiguity. If no conflicting paths can be found then G is

Algorithm 2. Computing the NFA from a grammar.

```
function BUILD-NFA() =
1 $Q = I$ // the items of G
2 $R = \{A \to \alpha \bullet X \beta \xmapsto{X} A \to \alpha X \bullet \beta \mid \}$ // shifts
3   $\cup\ \{A \to \alpha \bullet B \gamma \xmapsto{\langle_i} B \to \bullet \beta \mid i = \mathsf{pid}(B \to \beta)\}$ // derives
4   $\cup\ \{B \to \beta \bullet \xmapsto{\rangle_i} A \to \alpha B \bullet \gamma \mid i = \mathsf{pid}(B \to \beta)\}$ // reduces
5 return $(Q, R)$
```

unambiguous, but otherwise it is uncertain whether or not all conflicting paths represent ambiguous strings in $\mathcal{L}(G)$. However, the conflicting paths can be used to find harmless production rules. These are the rules that are not or incompletely used on these paths. If not all items of a production are used in the overapproximated set of ambiguous parse trees of G, then the production can certainly not be used to create a real ambiguous string in $\mathcal{L}(G)$.

4.2 Step 2: Construct and Traverse Pair Graph

NFA Traversal. To collect the items used on all conflicting path pairs we can traverse the NFA with two cursors at the same time. The traversal starts with both cursors at the start item $S' \to \bullet S\$$. From there they can progress through the NFA either independently or synchronized, depending on the type of transition that is being followed. Because we are looking for conflicting paths that represent different parse trees of the same string, the cursors should shift the same symbols in V. To enforce this we only allow synchronized shifts of equal symbols. The derive and reduce transitions are followed asynchronously, because the number of brackets on each path may vary.

During the traversal we wish to avoid the derivation of unambiguous substrings, i.e. an identical sequence of one derive, zero or more shifts, and one reduce on both paths, and prefer non-terminal shifts instead. This enables us to filter more items and edges from the NFA. Identical reduce transitions on both paths are therefore not allowed if no conflicts have occurred yet since their corresponding derives. A path can thus only reduce if the other path can be continued with a different reduce or a shift. This puts the path in conflict with the other. After the paths are conflicting we do allow identical reductions (synchronously), because otherwise it would be impossible to reach the end item $S' \to S\$\bullet$. To register whether a path is in conflict with the other we use boolean flags, one for each path. For a more detailed description of these flags we refer to [13,4].

Algorithm 3 describes the traversal of path pairs through the NFA. It contains gaps ①–④ that we will fill in later on (Algorithm 7), when extending it to handle character-level grammars. To model the state of the cursors during the traversal we use an *item pair* datatype with four fields: two items q_1 and q_2 in Q, and two conflict flags c_1 and c_2 in $\mathbb{B}$. We use Π to denote the set of all possible item pairs.

The function TRAVERSE-EDGES explores all possible continuations from a given item pair. We assume it has access to the global NFA variables Q and R. To traverse each pair graph edge it calls the function TRAVERSE-EDGE — not explained here — which in turn calls TRAVERSE-EDGES again on the next pair. The function SHIFTABLE determines the symbol that can be shifted on both paths. In the baseline setting we can only shift if the next symbols of both paths are identical. Later on we will extend this function to handle character-level grammars. The function CONFLICT determines whether a reduce transition of a certain path leads to a conflict. This is the case if the other path can be continued with a shift or reduce that differs from the first path's reduce.

Pair Graph. There can be infinitely many path pairs through the NFA, which can not all be traversed one-by-one. We therefore model all conflicting path pairs with a finite structure, called a *pair graph*, which nodes are item pairs. The function TRAVERSE-EDGES describes the edges of this graph. An infinite amount of path pairs translates to

Algorithm 3. Traversing NFA edge pairs.

```
function TRAVERSE-EDGES($p \in \Pi$) =
  for each $(p.q_1 \xmapsto{\langle_i} q_1') \in R$ do // derive $q_1$
    $p' = p,\ p'.q_1 = q_1',\ p'.c_1 = 0$
    TRAVERSE-EDGE$(p, p')$
  od
  for each $(p.q_2 \xmapsto{\langle_i} q_2') \in R$ do // derive $q_2$
    $p' = p,\ p'.q_2 = q_2',\ p'.c_2 = 0$
    TRAVERSE-EDGE$(p, p')$
  od
  for each $(p.q_1 \xmapsto{X} q_1'), (p.q_2 \xmapsto{Y} q_2') \in R$
  do // synchronized shift
    if SHIFTABLE$(X, Y) \neq \emptyset$ then
      $p' = p,\ p'.q_1 = q_1',\ p'.q_2 = q_2'$
      // ... ①
      TRAVERSE-EDGE$(p, p')$
    fi
  for each $(p.q_1 \xmapsto{\rangle_i} q_1') \in R$ do
    if CONFLICT$(p.q_2, \rangle_i)$ then
      // conflicting reduction of $q_1$
      $p' = p,\ p'.q_1 = q_1',\ p'.c_1 = 1$
      // ... ②
      TRAVERSE-EDGE$(p, p')$
    fi
  for each $(p.q_2 \xmapsto{\rangle_i} q_2') \in R$ do
    if CONFLICT$(p.q_1, \rangle_i)$ then
      // conflicting reduction of $q_2$
      $p' = p,\ p'.q_2 = q_2',\ p'.c_2 = 1$
      // ... ③
      TRAVERSE-EDGE$(p, p')$
    fi
  if $p.c_1 \vee p.c_2$ then
    for each $(p.q_1 \xmapsto{\rangle_i} q_1'), (p.q_2 \xmapsto{\rangle_i} q_2') \in R$
    do // synchronized reduction
      $p' = p,\ p'.q_1 = q_1',\ p'.q_2 = q_2'$
      $p'.c_1 = p'.c_2 = 1$
      // ... ④
      TRAVERSE-EDGE$(p, p')$
    od

function SHIFTABLE($X \in V, Y \in V$) =
  if $X = Y$ then return $X$ else return $\emptyset$

function CONFLICT($q \in Q,\ \rangle_i \in T_{\rangle}$) =
  return $\exists q' \in Q, u \in T_{\langle}^* : (\exists X : q \xmapsto{uX}{}^{+} q') \vee (\exists \rangle_j \neq \rangle_i : q \xmapsto{u\rangle_j}{}^{+} q')$
```

cycles in this finite pair graph. To find the items used on all conflicting paths it suffices to do a depth first traversal of the pair graph that visits each edge pair only once.

4.3 Steps 3–4: NFA Filtering and Harmless Rules Identification

After the items used on conflicting path pairs are collected we can identify harmless production rules from them. As said, these are the productions of which not all items are used. All other productions of G are potentially harmful, because it is uncertain if they can really be used to derive ambiguous strings.

We filter the harmless production rules from the NFA by removing all their items and pruning dead ends. If there are productions of which some but not all items were used, we actually remove a number of conflicting paths that do not represent valid parse trees of G. After filtering there might thus be even more unused items in the NFA. We therefore repeat the traversing and filtering process until no more items can be removed. Then, all productions that are not used in the NFA are harmless. This step concludes the first stage of our framework (*Find Harmless Productions* in Figure 1).

4.4 Steps 5–7: NFA Reconstruction and Sentence Generation

In the second part of our framework we use an inverted SGLR parser [7] as a sentence generator to find real ambiguous sentences in the remainder of the NFA. However, certain states in the NFA might not lead to the generation of terminal-only sentences anymore, due to the removal of terminal shift transitions during filtering. These are the states with outgoing non-terminal shift transitions that have no corresponding derive and reduce transitions anymore. To make such a non-terminal productive again we introduce a new terminal-only production for it that produces a shortest string from its original language. Then we add a new chain of derive, shift, and reduce transitions for this production to the states before and after the unproductive non-terminal shift.

After the NFA is reconstructed we generate an LR(0) pushdown automaton from it to generate sentences with. In contrast to the first stage, we now do need a stack because we only want to generate proper derivations of the grammar. Also, because of the item closure that is applied in LR automata, all derivations are unfolded statically, which saves generation steps at run-time.

The inverted parser generates all sentences of the grammar, together with their parse trees. If it finds a sentence with multiple trees then these are reported to the user. They are the most precise ambiguity reports possible, and are also very descriptive because they show the productions involved [3]. Because the number of derivations of a grammar can be infinite, we continue searching strings of increasing length until a certain time limit is reached. The number of strings to generate can grow exponentially with increasing length, but filtering unambiguous derivations beforehand can also greatly reduce the time needed to reach a certain length as Section 7 will show.

5 Ambiguity Detection for Character-Level Grammars

After sketching the baseline algorithm we can extend it to find ambiguities in character-level grammars. We take disambiguation filters into account during ambiguity detection, so we do not report ambiguities that are already solved by the grammar developer. Furthermore, we explain and fix the issue that the baseline harmless rules filtering is unable to properly deal with follow restrictions.

5.1 Application of Baseline Algorithm on Example Grammar

Before explaining our extensions we first show that the baseline algorithm can lead to incorrect results on character-level grammars. If we apply it to the example grammar of Figure 2, the harmless production rule filter will actually remove ambiguities from the grammar. Since the filtering is supposed to be conservative, this behaviour is incorrect.

The baseline algorithm will ignore the reject rule and follow restrictions in the grammar (Rules 6, 7, 9–11), and will therefore find the ambiguities that these filters meant to solve. Ambiguous strings are, among others, "`float f;`" (`float` can be a keyword or identifier) and "`intList l;`" (`intList` can be one or more specifiers). Rules 1–5 will therefore be recognized as potentially harmful. However, in all ambiguous strings, the substrings containing whitespace will always be unambiguous. This is detected by the PG traversal and Rule 8 (`Ws ::= [\ \t\n]+`) will therefore become harmless.

Rule 8 will be filtered from the grammar, and during reconstruction `Ws?` will be terminalized with the shortest string from its language, in this case ε. This effectively removes all whitespace from the language of the grammar. In the baseline setting the grammar would still be ambiguous after this, but in the character-level setting the language of the grammar would now be empty! The follow restriction of line 9 namely dictates that valid `Declaration` strings should contain at least one whitespace character to separate specifiers and identifiers.

This shows that our baseline grammar filtering algorithm is not suitable for character-level grammars as is, because it might remove ambiguous sentences. In addition, it might even introduce ambiguities in certain situations. This can happen when non-terminals are removed that have follow restrictions that prohibit a second derivation of a certain string. In short, follow restrictions have a non-context-free influence on sentence derivation, and the baseline algorithm assumes only context-free derivation steps. In the extensions presented in the next section we repair this flaw and make sure that the resulting algorithm does not introduce or lose ambiguous sentences.

5.2 Changes to the Baseline Algorithm

The differences between character-level grammars and conventional grammars result in several modifications of our baseline algorithm. These modifications deal with the definitions of both the NFA and the pair graph. We reuse the NFA construction of Algorithm 2 because it is compliant with character-level productions, and apply several modifications to the NFA afterwards to make it respect a grammar's derivation restrictions and follow restrictions. An advantage of this is that we do not have to modify the pair graph construction. To keep the test practical and conservative we have to make sure that the NFA remains finite, while its paths describe an overapproximation of $\mathcal{S}(G_b)$.

Character Classes. Because of the new shape of the productions, we now shift entire character classes at once, instead of individual terminal symbols. This avoids adding derives, shifts and reduces for the terminals in all character classes, which would bloat the NFA, and thus also the pair graph. In the PG we allow a synchronized shift of two character classes if their intersection is non-empty. To enforce this behaviour we only need to change the SHIFTABLE function as shown in Algorithm 4.

Derivation Restrictions and Follow Restrictions. After the initial NFA is constructed we remove derive and reduce edges that are disallowed by the derivation restrictions. This is described in function FILTER-DERIVE-RESTRICTIONS in Algorithm 5. Then we propagate the follow restrictions through the NFA to make it only generate strings that comply with them. This is described in function PROPAGATE-FOLLOW-

Algorithm 4. SHIFTABLE function for character-level pair graph.

```
function SHIFTABLE($X \in N, Y \in N$) =
  // returns the symbol that can be shifted from $X$ and $Y$
  if $X \in \mathcal{C} \land Y \in \mathcal{C}$ then return $X \cap Y$      // $X$ and $Y$ are character classes
  else if $X = Y$ then return $X$      // $X$ and $Y$ are the same non-terminal
  else return $\emptyset$      // no shift possible
```

Algorithm 5. Filtering derive restrictions from the NFA.

```
function FILTER-DERIVE-RESTRICTIONS($R$) =
    return $R \backslash \{A \to \alpha \bullet B\gamma \overset{\langle_i}{\longmapsto} B \to \bullet\beta \mid i = \mathsf{pid}(B \to \beta),\ (A \to \alpha \bullet B\gamma, B \to \beta) \in R_D\}$
             $\backslash \{B \to \beta \bullet \overset{\rangle_i}{\longmapsto} A \to \alpha B \bullet \gamma \mid i = \mathsf{pid}(B \to \beta),\ (A \to \alpha \bullet B\gamma, B \to \beta) \in R_D\}$
```

RESTRICTIONS in Algorithm 6. The operation will result in a new NFA with states that are tuples containing a state of the original NFA and a set of follow restrictions over $\mathcal{P}(T^+)$. A new state cannot be followed by strings that have a prefix in the state's follow restrictions. To enforce this we constrain character class shift edges according to the follow restrictions of their source states.

The process starts at $(S' \to \bullet S\$,\ \emptyset)$ and creates new states while propagating a state's follow restrictions over the edges of its old NFA item. In contrast to the original NFA, which had at most one shift edge per state, states in the new NFA can have multiple. This is because non-terminal or character class edges actually represent the shift of multiple sentences, which can each result in different follow restrictions. Lines 6–9 show the reconstruction of character-class shift edges from a state $(A \to \alpha \bullet B\beta, f)$. Shift edges are added for characters in B that are allowed by f. All characters in B that will result in the same new set of follow restrictions are combined into a single shift edge, to not bloat the new NFA unneccesarily. The restrictions after a shift of a are the tails of the strings in f beginning with a, and are calculated by the function NEXT-FOLLOW.

Line 12 describes how a state's restrictions are passed on unchanged over derive edges. Lines 13–20 show how new non-terminal shift edges are added from a state $(A \to \alpha \bullet B\beta, f)$ once their corresponding reduce edges are known. This is convenient because we can let the propagation calculate the different follow restrictions that can reach $A \to \alpha B \bullet \beta$. Once the restrictions that were passed to the derive have reached a state $B \to \gamma \bullet$, we propagate them upwards again over a reduce edge to $A \to \alpha B \bullet \beta$. If B has follow restrictions — in R_F — these are added to the new state as well. Note that multiple follow restriction sets might occur at the end of a production, so we might have to reduce a production multiple times. For a given state $B \to \bullet\gamma$, the function SHIFT-ENDS returns all states that are at $B \to \gamma \bullet$ and that are reachable by shifting.

If the reduced production is of form $B \to \varepsilon$ we create a special non-terminal symbol B^ε and make it the label of the shift edge instead of B. This is a small precision improvement of the PG traversal. It prevents the situation where a specific non-terminal shift that —because of its follow restriction context— only represents the empty string, is traversed together with another instance of the same non-terminal that cannot derive ε.

The propagation ends when no new edges can be added to the new NFA. In theory the new NFA can now be exponentially larger than the original, but since follow restrictions are usually only used sparingly in the definition of lexical syntax this will hardly happen in practice. In Section 7 we will see average increases in NFA size of a factor 2–3.

Rejects. Instead of encoding a grammar's rejects in the NFA, we choose to handle them during the PG traversal. Consider an element (A, B) in R_R, which effectively subtracts the language of B from that of A. If the language of B is regular then we could, for instance, subtract it from the NFA part that overapproximates the language of A. This

Algorithm 6. Propagating follow restrictions through the NFA.

```
function PROPAGATE-FOLLOW-RESTRICTIONS$(Q, R)$ =
// propagate follow restrictions through NFA $(Q, R)$ and return a new NFA $(Q', R')$
1  $Q' = \{(S' \to \bullet S\$, \emptyset)\}$, $R' = \emptyset$
2  repeat
3    add all states used in $R'$ to $Q'$
4    for $q_f = (A \to \alpha \bullet B\beta, f) \in Q'$ do
5      if $B \in \mathcal{C}$ then // $B$ is a character class
6        for $a \in B$, $a \notin f$ do // all shiftable characters in $C$
7          let $B' = \{b \mid b \in B, b \notin f$, NEXT-FOLLOW$(a, f)$ = NEXT-FOLLOW$(b, f)\}$
8          add $q_f \xmapsto{B'} (A \to \alpha B \bullet \beta$, NEXT-FOLLOW$(a, f))$ to $R'$
9        od
10     else // $B$ is a normal non-terminal
11       for $A \to \alpha \bullet B\beta \xmapsto{\langle_i} q' \in R$ do
12         add $q_f \xmapsto{\langle_i} (q', f)$ to $R'$ // propagate $f$ over derivation
13         for $q_f^r = (q^r, f^r) \in$ SHIFT-ENDS$((q', f))$
14           let $q_f^s = (A \to \alpha B \bullet \beta, f^r \cup R_F(B))$ // shift target
15           add $q_f^r \xmapsto{\rangle_i} q_f^s$ to $R'$ // reduction to shift target
16           if $\mathsf{bracketP}(\langle_i) = B \to \varepsilon$ then
17             add $q_f \xmapsto{B^\varepsilon} q_f^s$ to $R'$ // non-terminal shift representing empty string
18           else
19             add $q_f \xmapsto{B} q_f^s$ to $R'$ // non-terminal shift of non-empty strings
20         od
21       od
22 until no more edges can be added to $R'$
23 return $(Q', R')$

function SHIFT-ENDS$((A \to \bullet\alpha, f) \in Q')$ =
1  // return the states at the end of $A \to \alpha$, reachable from q using only shifts
2  let $\dashrightarrow = \{q \dashrightarrow q' \mid q \xmapsto{B} q' \in R'\}$ // the shift transitions of $R'$
3  return $\{(A \to \alpha\bullet, f') \mid (A \to \bullet\alpha, f) \dashrightarrow^* (A \to \alpha\bullet, f')\}$

function NEXT-FOLLOW$(a \in T, f \in \mathcal{P}(T^+))$
1  return $\{\alpha \mid a\alpha \in f, \alpha \neq \varepsilon\}$ // the next follow restrictions of $f$ after a shift of $a$
```

would not violate the finiteness and overapproximation requirements. However, if the language of B is context-free we have to *under*approximate it to finite form first, to keep the NFA an overapproximation and finite. A possible representation for this would be a second NFA, which we could subtract from the first NFA beforehand, or traverse alongside the first NFA in the PG.

Instead, we present a simpler approach that works well for the main use of rejects: keyword reservation. We make use of the fact that keywords are usually specified as a set of non-terminals that represent literal strings — like Rules 6 and 7 in Figure 2. The production rules for `"int"`, `"float"`, etc. are not affected by the approximation, and appear in the NFA in their original form. We can thus recognize that, during the PG traversal, a path has completely shifted a reserved keyword if it reduces `"int"`. After

Algorithm 7. Extensions to TRAVERSE-EDGES for avoiding rejected keywords.

```
// at ① (shift) insert:
    $p'.r_1 = p'.r_2 = \emptyset$ // clear reduced sets

// at ② and ④ (conflicting and pairwise reduce) insert:
    if not CHK-REJECT($\rangle_i, p.r_2$) then continue
    $p'.r_1$ = NEXT-REJECT($\rangle_i, p.r_1$)

// similarly, insert at ③ and ④:
    if not CHK-REJECT($\rangle_i, p.r_1$) then continue
    $p'.r_2$ = NEXT-REJECT($\rangle_i, p.r_2$)

function CHK-REJECT($\rangle_i \in T_\rangle, r \in \mathcal{P}(N)$) =
    // returns whether a reduction with $\rangle_i$ is possible after reductions $r$ on other path
    let $A = \mathsf{bracketN}(\rangle_i)$
    return $\neg\exists\ B \in r : (A, B) \in R_R \vee (B, A) \in R_R$

function NEXT-REJECT($\rangle_i \in T_\rangle, r \in \mathcal{P}(N)$) =
    // adds non-terminal reduced with $\rangle_i$ to $r$ if it is involved in a reject
    let $A = \mathsf{bracketN}(\rangle_i)$
    if $\exists B \in r : (A, B) \in R_R \vee (B, A) \in R_R$ then
        return $r \cup \{A\}$
    else return $r$
```

that, we can prevent the other path from reducing `Identifier` before the next shift. This does not restrict the language of `Identifier` in the NFA — it is kept overapproximated —, but it does prevent the ambiguous situation where "`int`" is recognized as an `Identifier` on one path and as an `"int"` on the other path.

Of course, `Identifier` could also be reduced before `"int"`, so we need to register the reductions of both non-terminals. During the PG traversal, we keep track of all reduced non-terminals that appear in R_R, in two sets r_1 and r_2, one for each path. Then, if a path reduces a non-terminal that appears in a pair in R_R, together with a previously reduced non-terminal in the other path's set, we prevent this reduction. The sets are cleared again after each pairwise shift transition. Algorithm 7 shows this PG extension.

5.3 NFA Reconstruction

In Section 5.1 we saw that follow restrictions should be handled with care when filtering and reconstructing a grammar, because of their non-context-free behaviour. By removing productions from a grammar certain follow restrictions can become unavoidable, which removes sentences from the language. On the other hand, by removing follow restrictions new sentences can be introduced that were previously restricted. When reconstructing a character-level grammar we thus need to terminalize filtered productions depending on the possible follow-restrictions they might generate or that might apply to them.

Instead, by directly reusing the filtered NFA for sentence generation, we can avoid this problem. The follow restrictions that are propagated over the states already describe the follow restriction context of each item. For each distinct restriction context of an item a separate state exists. We can just terminalize each unproductive non-terminal shift edge with an arbitrary string from the language of its previously underlying automaton.

Furthermore, the filtered NFA is a more detailed description of the potentially ambiguous derivations than a filtered grammar, and therefore describes less sentences. For instance, if derive and reduce edges of a production $B \to \beta$ are filtered out at a specific item $A \to \alpha \bullet B\gamma$, but not at other items, we know $B \to \beta$ is harmless in the context of $A \to \alpha \bullet B\gamma$. The propagated follow restrictions also provide contexts in which certain productions can be harmless. We could encode this information in a reconstructed grammar by duplicating non-terminals and productions of course, but this could really bloat the grammar. Instead, we just reuse the baseline NFA reconstruction algorithm.

6 Grammar Unfolding

In Section 7 we will see that the precision of the algorithm described above is not always sufficient for some real life grammars. The reason for this is that the overapproximation in the NFA is too aggressive for character-level grammars. By applying grammar unfoldings we can limit the approximation, which improves the precision of our algorithm.

The problem with the overapproximation is that it becomes too aggressive when certain non-terminals are used very frequently. Remember that due to the absence of a stack, the derive and reduce transitions do not have to be followed in a balanced way. Therefore, after deriving from an item $A \to \alpha \bullet B\beta$ and shifting a string in the language of B, the NFA allows reductions to any item of form $C \to \gamma B \bullet \delta$. This way, a path can jump to another production while being in the middle of a first production. Of course, a little overapproximation is intended, but the precision can be affected seriously if certain non-terminals are used very frequently. Typical non-terminals like that in character-level grammars are those for whitespace and comments, which can appear in between almost all language constructs. Since these non-terminals can usually derive to ε, we can thus jump from almost any item to almost any other item by deriving and reducing them.

To restrict the overapproximation we can unfold the frequently used non-terminals in the grammar, with a technique similar to one used in [6]. A non-terminal is unfolded by creating a unique copy of it for every place that it occurs in the right-hand sides of the production rules. For each of these copies we then also duplicate the entire sub-grammar of the non-terminal. The NFA thus gets a separate isolated sub-automaton for each occurence of an unfolded non-terminal. After the derivation from an item $A \to \alpha \bullet B\beta$ a path can now only reduce back to $A \to \alpha B \bullet \beta$, considering B is unfolded. After unfolding, the NFA contains more states, but has less paths through it because it is more deterministic. In the current implementation we unfold all non-terminals that describe whitespace, comments, or literal strings like keywords, brackets and operators. Later on we will refer to this unfolding extension as CHAR+UNF.

7 Experimental Results

We have evaluated our ambiguity detection algorithm for character-level grammars on the grammar collection shown in Table 1. All grammars are specified in SDF [10,16]. The selection of this set is important for external validity. We have opted for grammars of

Table 1. Character-level grammars used for validation

Name	Prods.	SLOC	Non-terms.	Derive rest.	Follow restr.	Reserved keywords
C[1]	324	415	168	332	10	32
C++[2]	807	4172[a]	430	1	87	74
ECMAScript[3]	403	522	232	1	27	25
Oberon0[4]	189	202	120	132	31	27
SQL-92[5]	419	495	266	23	5	30
Java 1.5[6]	698	1629	387	297	78	56

[1] SDF grammar library, revision 27501, `http://www.meta-environment.org`

[2] Transformers 0.4, `http://www.lrde.epita.fr/cgi-bin/twiki/view/Transformers/Transformers`

[3] ECMAScript-front, revision 200, `http://strategoxt.org/Stratego/EcmaScriptFront`

[4] RASCAL Oberon0 project (converted to SDF), rev. 34580, `http://svn.rascal-mpl.org/oberon0/`

[5] SQL-front, revision 20713, `http://strategoxt.org/Stratego/SqlFront`

[6] Java-front, revision 17503, `http://strategoxt.org/Stratego/JavaFront`

[a] After removal of additional attribute code

general purpose programming languages, which makes it easier for others to validate our results. For each grammar we give its name, number of productions, number of source lines (SLOC), number of non-terminals, number of priorities and associativities (derivation restrictions), number of follow restrictions and number of reserved keywords.

7.1 Experiment Setup

We have run both our NFA filtering and sentence generation algorithms on each of these grammars. Most measurements were carried out on an Intel Core2 Quad Q6600 2.40GHz with 8GB DDR2 memory, running Fedora 14. A few memory intensive runs were done on an Amazon computing cloud EC2 High-Memory Extra Large Instance with 17.1GB memory. The algorithms have been implemented in Java and are available for download at `http://homepages.cwi.nl/~basten/ambiguity`. In order to identify the effects of the various extensions, we present our empirical findings for the following combinations:

- **BASE**: the baseline algorithm for token-level grammars as described in Section 4, with the only addition that whole character-classes are shifted instead of individual tokens. Even though this configuration can lead to incorrect results, it is included as a baseline for comparison.
- **CHAR**: the baseline algorithm extended for handling character-level grammars as described in Section 5, including extensions for follow restrictions, derive restrictions and rejects.
- **CHAR+UNF**: the **CHAR** algorithm combined with grammar unfolding (Section 6).

7.2 Results and Analysis

In Table 2 we summarize our measurements of the NFA filtering and harmless production rule detection. For each grammar and extension configuration we give the number of harmless productions found versus total number of productions, number of edges filtered from the NFA, execution time (in seconds) and memory usage (in MB).

Table 2. Timing and precision results of filtering harmless productions

Grammar	**Method**	**Harmless productions**	**NFA edges filtered**	**Time** (sec)	**Memory** (MB)
C	BASE	48 / 324	343 / 14359	64	2128
	CHAR	62 / 324	2283 / 24565	120	3345
	CHAR+UNF	75 / 324	8637 / 30653	97	2616
C++	BASE	0 / 807	0 / 8644	32	1408
	CHAR	0 / 807	0 / 39339	527	7189
	CHAR+UNF[a]	–	–	>9594	>17.3G
ECMAScript	BASE	44 / 403	414 / 4872	12	547
	CHAR	46 / 403	1183 / 10240	46	1388
	CHAR+UNF	88 / 403	9887 / 19890	31	1127
Oberon0	BASE	0 / 189	0 / 3701	4.2	256
	CHAR	70 / 189	925 / 6162	9.0	349
	CHAR+UNF	73 / 189	10837 / 20531	14	631
SQL-92	BASE	13 / 419	98 / 4944	16	709
	CHAR	20 / 419	239 / 9031	83	2093
	CHAR+UNF	65 / 419	7285 / 14862	37	1371
Java 1.5	BASE	0 / 698	0 / 16844	60	2942
	CHAR	0 / 698	0 / 45578	407	7382
	CHAR+UNF[a]	189 / 698	180456 / 262030	1681	15568

[a]Run on Amazon EC2 High-Memory Extra Large Instance

Every configuration was able to filter an increasing number of productions and edges for each of the grammars. For C and ECMAScript **BASE** could already filter a small number rules and edges, although it remains unsure whether these are all harmless because the baseline algorithm cannot handle follow restrictions properly. For C and Oberon0 our character-level extensions of **CHAR** improved substantially upon **BASE**, without the risk of missing ambiguous sentences.

Of all three configurations **CHAR+UNF** was the most precise. For the grammar order of the table, it filtered respectively 23%, 0%, 22%, 39%, 16% and 27% of the production rules, and 28%, 0%, 50%, 53%, 49% and 69% of the NFA edges. Unfolding grammars leads to larger but more deterministic NFAs, which in turn can lead to smaller pair graphs and thus faster traversal times. This was the case for most grammars except the larger ones. ECMAScript, SQL-92 and Oberon0 were checkable in under 1 minute, and C in under 2 minutes, all requiring less than 3GB of memory. Java 1.5 was checkable in just under 16GB in 30 minutes, but for the C++ grammar — which is highly ambiguous — the pair graph became too large. However, the additional cost of unfolding was apparently necessary to deal with the complexity of Java 1.5.

Table 3 allows us to compare the sentence generation times for the unfiltered and filtered NFAs. For each grammar and NFA it shows the number of sentences of a certain length in the language of the NFA, and the times required to search them for ambiguities. The unfiltered sentence generation also takes disambiguation filters into account. C++ is not included because its NFA could not be filtered in the previous experiments.

For all grammars we see that filtering with **CHAR** and **CHAR+UNF** lead to reductions in search space and generation times. To indicate whether the investments in

Table 3. Timing results of sentence generation. Times are in seconds. For each sentence length, the run-time of the fastest configuration (also taking filtering time into account) is highlighted. Speedup is calculated as $\frac{\text{unfiltered sentence gen. time}}{\text{filtering time+sentence gen. time}}$.

Grammar	Len	Ambig	Unfiltered		CHAR		CHAR+UNF		Maximum
		NTs	Sentences	Time	Sentences	Time	Sentences	Time	speedup
C	5	6	345K	**7.9**	273K	5.9	267K	5.9	0.08x
	6	8	5.06M	**35**	3.77M	25	3.66M	25	0.29x
	7	8	75.5M	398	53.4M	270	51.6M	**259**	1.1x
	8	9	1.13G	5442	756M	3466	727M	**3362**	1.6x
	9	10	17.0G	78987	10.8G	47833	10.3G	**47018**	1.7x
ECMAScript	3	6	14.2K	**4.5**	11.7K	3.5	9.29K	3.3	0.13x
	4	8	274K	**11**	217K	8.9	159K	6.7	0.29x
	5	10	5.17M	149	3.92M	120	2.64M	**69**	1.5x
	6	11	96.8M	2805	70.5M	2186	43.8M	**1184**	2.3x
	7	12	1.80G	54175	1.26G	41091	719M	**20264**	2.7x
Oberon0	22	0	21.7M	60	320	**1.0**	182	1.0	6.0x
	23	0	62.7M	186	571	**1.0**	248	1.0	19x
	24	0	247M	815	1269	**1.0**	468	1.0	82x
	25	0	1.39G	4951	3173	**1.1**	1343	1.1	490x
	26	0	9.56G	35007	9807	**1.3**	3985	1.3	3399x
	32	0			108M	172	13.8M	**28**	
	33	0			549M	885	55.6M	**101**	
	34	0			2.80G	4524	224M	**393**	
	35	0			14.3G	22530	906M	**1591**	
	36	0					3.66G	**6270**	
SQL-92	11	5	2.65M	**16**	1.54M	9.4	321K	4.2	0.39x
	12	6	15.8M	102	7.36M	47	1.66M	**14**	2.0x
	13	6	139M	1018	51.3M	379	11.5M	**90**	8.0x
	14	6	1.49G	11369	453M	3572	90.8M	**711**	15x
	15	7			4.39G	35024	742M	**5781**	
	16	8					6.13G	**47211**	
Java 1.5	7	0	187K	**33**			39.1K	6.8	0.02x
	8	1	3.15M	**115**			482K	20	0.07x
	9	1	54.7M	**1727**			6.05M	212	0.91x
	10	1	959M	39965			76.2M	**4745**	6.2x

filtering time actually pay off, the last column contains the maximum speedup gained by either **CHAR** or **CHAR+UNF**. For sentence lengths that are already relatively cheap to generate, filtering beforehand has no added value. However, the longer the sentences get the greater the pay-off. We witnessed speedup factors ranging from a small 1.1 (C length 7) to a highly significant 3399 (Oberon0 length 26). Filtering Oberon0 with **CHAR+UNF** was so effective that it increased the sentence length checkable in around 15 minutes from 24 to 35.

For most grammars filtering already becomes beneficial after around 15 seconds to 6 minutes. For Java 1.5 this boundary lies around 35 minutes, because of its high filtering time. However, after that we see an immediate speedup of a factor 6.2. In all cases **CHAR+UNF** was superior to **CHAR**, due to its higher precision and lower run-times.

The third column of Table 3 contains the number of ambiguous non-terminals found at each length. Because of filtering, ambiguous non-terminals at larger lengths were found earlier in multiple grammars. There were 2 ambiguous non-terminals in C that were found faster, and 4 non-terminals in ECMAScript and 3 in SQL-92.

Concluding, we see that our character-level NFA filtering approach was very beneficial on the small to medium grammars. A relatively low investment in filtering time — under 2 minutes — lead to significant speedups in sentence generation. This enabled the earlier detection of ambiguities in these grammars. For the larger Java 1.5 grammar the filtering became beneficial only after 32 minutes, and for the highly ambiguous C++ grammar the filtering had no effect at all. Nevertheless, ambiguity detection for character-level grammars is ready to be used in interactive language workbenches.

7.3 Validation

In [4] we proved the correctness of our baseline algorithm. To further validate our character-level extensions and their implementations we applied them on a series of toy grammars and grammars of real world programming languages. We ran various combinations of our algorithms on the grammars and automatically compared the ambiguous sentences produced, to make sure that only those ambiguities that exist in a grammar were found, so not more and not less. For the version of our implementation that we used for the experiments above, we found no differences in the ambiguous strings generated. The validation was done in the following stages:

- First we built confidence in our baseline sentence generator by comparing it to the external sentence generators AMBER [14] and CFGAnalyzer [1]. For this we used a grammar collection also used in [3], which contains 87 small toy grammars and 25 large grammars of real-world programming languages.
- Then we validated the character-level extension of the baseline sentence generator by comparing it to a combination of our baseline sentence generator and the SGLR [7] parser used for SDF. By running the baseline sentence generator on character-level grammars it will report more strings as ambiguous than actually exist in a grammar, because it does not regard disambiguation filters. We therefore filter out the truly ambiguous sentences by using the SGLR parser as an oracle, and test whether our character-level sentence generator finds exactly the same ambiguous sentences. In some situations SGLR will produce non-optimal parse trees, so we had to verify these by hand. In this step and the following we used the SDF grammars in Table 1.
- Third, we validated our NFA filtering algorithms by running the character-level sentence generator on both filtered and unfiltered NFAs. Because a filtered NFA contains only one reconstructed sentence for non-terminals with only harmless productions, it might produce less variations of ambiguous sentences. We therefore reduced all ambiguous sentences to their *core* ambiguous sentential forms [4] before comparison. This is done by removing the unambiguous substrings from an ambiguous sentence, and replacing them with their deriving non-terminal.

8 Conclusion

We have presented new algorithms for ambiguity detection for character-level grammars and by experimental validation we have found an affirmative answer to the question whether ambiguity detection can be scaled to this kind of grammars. We have achieved significant speedups of up to three orders of magnitude for ambiguity checking of real programming language grammars. Ambiguity detection for character-level grammars is ready to be used in interactive language workbenches, which is good news for the main application areas of these grammars: software renovation, language embedding and domain-specific languages.

References

1. Axelsson, R., Heljanko, K., Lange, M.: Analyzing Context-Free Grammars Using an Incremental SAT Solver. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 410–422. Springer, Heidelberg (2008)
2. Aycock, J., Horspool, R.N.: Schrödinger's token. Software: Practice & Experience 31(8), 803–814 (2001)
3. Basten, H.J.S.: The usability of ambiguity detection methods for context-free grammars. In: Johnstone, A., Vinju, J.J. (eds.) Proceedings of the Eigth Workshop on Language Descriptions, Tools and Applications (LDTA 2008). ENTCS, vol. 238 (2009)
4. Basten, H.J.S.: Tracking Down the Origins of Ambiguity in Context-Free Grammars. In: Cavalcanti, A., Deharbe, D., Gaudel, M.-C., Woodcock, J. (eds.) ICTAC 2010. LNCS, vol. 6255, pp. 76–90. Springer, Heidelberg (2010)
5. Basten, H.J.S., Vinju, J.J.: Faster ambiguity detection by grammar filtering. In: Brabrand, C., Moreau, P.-E. (eds.) Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications (LDTA 2010). ACM (2010)
6. Brabrand, C., Giegerich, R., Møller, A.: Analyzing ambiguity of context-free grammars. Sci. Comput. Program. 75(3), 176–191 (2010)
7. van den Brand, M.G.J., Scheerder, J., Vinju, J.J., Visser, E.: Disambiguation Filters for Scannerless Generalized LR Parsers. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 143–158. Springer, Heidelberg (2002)
8. Bravenboer, M., Visser, E.: Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In: Vlissides, J.M., Schmidt, D.C. (eds.) OOPSLA, pp. 365–383. ACM (2004)
9. Ginsburg, S., Harrison, M.A.: Bracketed context-free languages. Journal of Computer and System Sciences 1(1), 1–23 (1967)
10. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The syntax definition formalism SDF - reference manual. SIGPLAN Notices 24(11), 43–75 (1989)
11. 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.) OOPSLA, pp. 918–932. ACM (2010)
12. Klint, P., Visser, E.: Using filters for the disambiguation of context-free grammars. In: Proceedings of the ASMICS Workshop on Parsing Theory. pp. 1–20. Technical Report 126-1994, Università di Milano (1994)
13. Schmitz, S.: Conservative Ambiguity Detection in Context-Free Grammars. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 692–703. Springer, Heidelberg (2007)

14. Schröer, F.W.: AMBER, an ambiguity checker for context-free grammars. Tech. rep., compilertools.net (2001), `http://accent.compilertools.net/Amber.html`
15. Van Wyk, E., Schwerdfeger, A.: Context-aware scanning for parsing extensible languages. In: Consel, C., Lawall, J.L. (eds.) GPCE, pp. 63–72. ACM (2007)
16. Visser, E.: Syntax Definition for Language Prototyping. Ph.D. thesis, University of Amsterdam (September 1997)

Comparison of Context-Free Grammars Based on Parsing Generated Test Data

Bernd Fischer[1], Ralf Lämmel[2], and Vadim Zaytsev[3]

[1] Electronics and Computer Science, University of Southampton, Southampton, United Kingdom
[2] Software Languages Team, Universität Koblenz-Landau, Koblenz, Germany
[3] Software Analysis and Transformation Team, Centrum Wiskunde en Informatica, Amsterdam, The Netherlands

Abstract. There exist a number of software engineering scenarios that essentially involve equivalence or correspondence assertions for some of the context-free grammars in the scenarios. For instance, when applying grammar transformations during parser development—be it for the sake of disambiguation or grammar-class compliance—one would like to preserve the generated language. Even though equivalence is generally undecidable for context-free grammars, we have developed an automated approach that is practically useful in revealing evidence of nonequivalence of grammars and discovering correspondence mappings for grammar nonterminals. Our approach is based on systematic test data generation and parsing. We discuss two studies that show how the approach is used in comparing grammars of open source Java parsers as well as grammars from the course work for a compiler construction class.

Keywords: grammar-based testing, test data generation, coverage criteria, grammar equivalence, parsing, compiler construction, course work.

1 Introduction

The paper is concerned with the automated comparison of context-free grammars based on test data generated from a grammar. The goal is here to reveal evidence, if any, for grammar nonequivalence, and to suggest a correspondence mapping between the nonterminals of the compared grammars. If no evidence of grammar nonequivalence is found, then this status may support an assertion of grammar equivalence (against the odds of undecidability). We develop a corresponding approach for grammar comparison which we demonstrate with two studies. The resulting infrastructure and both studies in grammar comparison are available online.[1]

The following grammar comparison scenarios exemplify the relevance of the presented work.

[1] http://slps.sourceforge.net/testmatch

Grammar comparison scenarios

- ◇ *Parser implementation*: The implementor of a parser may start from the "readable" grammar in a language manual and then transform it so that ambiguities or inefficiencies or grammar class violations are addressed. For instance, the (recovered) Cobol grammar from IBM's standard [9,13] requires substantial transformations before a quality parser is obtained. Grammar comparison can be used to shield this laborious process against errors.
- ◇ *Language documentation*: The documenter is supposed to provide a readable grammar for which it may be hard to establish though that it precisely represents the intended language. For instance, each version of the Java Language Specification contains a "more readable" and a "more implementable" grammar [4], and a substantial number of deviations have been identified by a complex and laborious process of grammar convergence [17]. Grammar comparison can be used to improve automation of this process.
- ◇ *Interoperability testing*: Suppose that there exist multiple grammars (in fact, front-ends) for the same (intended) language. Interoperability testing may be based on code reviews or manually developed test suites. Grammar comparison techniques can be used to test for interoperability more automatically and systematically even during mapping preparation phase.
- ◇ *Teaching language processing*: Compiler construction is a very established subject in computer science education and there are continuous efforts to improve and update corresponding courses [1,5,24,27]. However, the typical course involves laborious efforts—on the educator's side—some of which can be reduced with grammar comparison. For instance, the nonterminal names of student solutions can be automatically connected with a reference solution. Differences between the generated languages can be automatically identified.

Contributions of the paper

- ◇ We develop a framework for grammar-based test data generation and various related coverage criteria with associated and modularized generation algorithms. This results in a simple and integrated framework—when compared to previous work.
- ◇ We develop a grammar matching algorithm which uses a systematic classification scheme for the nonterminal correspondences between two grammars starting from accept/reject results obtained by "combinatorial" parsing: all mappings between nonterminals of the grammars are evaluated.
- ◇ We produce empirical evidence for the power of grammar-based test data generation in practical situations based on two complementary studies. Different coverage criteria are shown to make a contribution in this context.

Roadmap of this paper: §2 presents a methodology for grammar comparison. §3 describes a set of coverage criteria and test data generation algorithms for use in grammar comparison. §4 reports on a grammar comparison study for Java grammars which concludes with a nonequivalence result in particular. §5 develops a matching algorithm for nonterminals based on parser applications to test data. §6 reports on a grammar comparison study for a compiler construction class managing to match grammars of the course work. §7 discusses related work. §8 concludes the paper.

2 Methodology

Overall, the idea of test-based comparison of grammars may appear relatively straightforward. Nevertheless, a suitable methodology has to be set up.

Asymmetric comparison. Given are two grammars G and G' which have been extracted from or can be turned uniformly into parsers (acceptors) A and A'. Here we call G the *reference grammar* and G' the *grammar under test*. Accordingly, G represents the intended language, and we want to support assertions of correctness and completeness for G' relative to G. We say that G' is *complete*, if A' accepts all strings that A accepts. We say that G' is *correct*, if A' rejects all strings that A rejects. With test-based comparison we can attempt to find counterexamples. That is, we generate (positive) test cases from G and apply A' to them; rejection provides evidence of incompleteness of G'. We also generate (positive) test cases from G' and apply A to them; rejection provides evidence of incorrectness of G'.

Symmetric comparison. In practice, we cannot always assume that one grammar is clearly a reference grammar. Instead, both grammars may simply compete with each other to appropriately capture an intended language. In this case, it does not make sense any longer to speak of correctness and completeness. One can still exercise both of the above-mentioned directions of test data generation and parser application, but what was called evidence of incompleteness or incorrectness previously simply reduces to evidence of nonequivalence. (A)symmetric comparison, as discussed here, is a form of *differential testing* [20].

Non-context-free effects. When discussing (a)symmetric grammar comparison so far, we stipulated that a parser A should precisely accept the language generated by the grammar G. Obviously, this is not necessarily true in practice. For instance, grammar-class restrictions or built-in ambiguity resolution strategies may imply that a generated parser rejects some part of the formal language. Also, parser descriptions may provide additional control that also goes beyond plain context-free grammars; see, for example, syntactic and semantic predicates in ANTLR. Further, a parser may rely on a designated lexer whose description may be incorporated into the grammar, but some aspects may be hard to model explicitly, e.g., whitespace handing. These and other differences between grammar and parser challenge the soundness of any grammar comparison approach. We encounter such effects in the case studies, but we defer a more general investigation of these effects to future work.

Nonterminal matching. When discussing (a)symmetric grammar comparison so far, we focused on confidence for equivalence or evidence for nonequivalence. As some of the introductory scenarios indicated, one may want to go beyond (non)equivalence and aim at nonterminal matching. This generalization is useful for *understanding* grammars and for preparing an effective *mapping* between derivation trees of the compared grammars, if needed. The key idea here is to use data sets indexed by nonterminals so that acceptance/rejection can be tested per nonterminal which eventually allows to match nonterminals from the two grammars when they accept each other test data sets better than for any other combination of nonterminals. For practicality's sake, it is important to support nonterminal matching even for grammars that are not fully equivalent.

Stochastic vs. systematic test data generation. As we discuss in [15], prior art in grammar-based testing focuses on *stochastic test data generation* (e.g., [19,25]). The canonical approach is to annotate a grammar with probabilistic weights on the productions and other hints. A test data set is then generated using probabilistic production selection and potentially further heuristics. Stochastic approaches have been successfully applied to practical problems. One conceptual challenge with stochastic approaches is that they require some amount of configuration to achieve coverage. For instance, recursive nonterminals in grammars imply a need for appropriate probabilistic weights so that divergence is avoided. This needs to be done carefully to avoid, in turn, insufficient coverage. In the present paper, we leverage systematic test data generation, by which we mean that test data sets are generated by effective enumeration methods for the coverage criteria of interest. These methods do not require any configuration. Also, these methods imply minimality of the test data sets in both an intuitive and a formal sense.

Larger sets of smaller test data items. Starting with Purdom's seminal work [22], there is the question of how to trade off size of test data *set* vs. size of test data *items*. For instance, when attempting to cover all productions of a grammar, one may generate a smaller test data set with each item covering as many additional productions as possible (thereby implying larger items); instead, one may also generate a larger test data set with each item covering as few individual productions as possible (thereby implying smaller items). In the present paper, without loss of generality, we adopt the latter principle which is well in line with general (unit) testing advice. We also refer to [20] for support of this principle.

3 Test Data Generation

Based on previous work on grammar-based test data generation [7,14,15,18,22,25], we develop a generation framework which accumulates a number of coverage criteria and associated generation algorithms in a modular manner. We have specified all ingredients in a declarative logic program of which we show excerpts below. (The complete specification, which also includes some optimizations, is available online; see the footnote on the first page.)

3.1 Grammars and Trees

Generation algorithms process a grammar and generate trees. We represent grammars as lists of productions. A production is a triple $p(L, N, X)$ consisting of an optional label L, a left-hand side nonterminal N, and a right-hand side expression X. *expr/1* specifies the allowed expression forms for BNF and EBNF, using functors *true* for ε, t for terminals, n for nonterminals, ',' for sequences, ';' for choices, '?' for optional parts, '*' and '+' for repetitions. The structure of trees follows exactly the one of grammars, and hence all functors are overloaded to represent trees as well as grammars. We refer to Figure 1 for details.[2] Grammar fragments are included into trees for origin tracking; see n and ';' on the right of the figure.

```
grammar(Ps)                          prod(p(L,N,X))
  ⇐ maplist(prod,Ps).                  ⇐ mapopt(atom,L), atom(N), expr(X).

expr(true).                          tree(true).
expr(t(T)) ⇐ atom(T).                tree(t(T)) ⇐ atom(T).
expr(n(N)) ⇐ atom(N).                tree(n(P,T)) ⇐ prod(P), tree(T).
expr(','(Xs)) ⇐ maplist(expr,Xs).    tree(','(Ts)) ⇐ maplist(tree,Ts).
expr(';'(Xs)) ⇐ maplist(expr,Xs).    tree(';'(X,T)) ⇐ expr(X), tree(T).
expr('?'(X)) ⇐ expr(X).              tree('?'(Ts)) ⇐ mapopt(tree,Ts).
expr('*'(X)) ⇐ expr(X).              tree('*'(Ts)) ⇐ maplist(tree,Ts).
expr('+'(X)) ⇐ expr(X).              tree('+'(Ts)) ⇐ maplist1(tree,Ts).
```

Fig. 1. Logic programming-based specification of grammars and trees

3.2 Coverage Criteria

Suppose that S is a set of derivation trees for a given grammar G. We say that S achieves *trivial coverage* (TC), if S is not empty; S achieves *nonterminal coverage* (NC), if S exercises each nonterminal of G at least once; S achieves *production coverage* (PC), if S exercises each production of G at least once; S achieves *branch coverage* (BC), if S exercises each branch for each occurrence of ';', '?', '*', '+' at least once; S achieves *unfolding coverage* (UC), if S exercises each production of each right-hand side nonterminal occurrence at least once.

[2] The definitions leverage higher-order predicates *maplist/2*, *maplist1/2*, *mapopt/2* for applying unary predicates to arbitrary lists, to lists with at least one element, or to lists of zero or one elements, respectively.

For backward compatibility with preexisting terminology [14], we also give the name *context-dependent branch coverage* (CDBC) to the combination of BC and UC.

Trivial, nonterminal and production coverage presumably do not require further formal clarification. BC and UC require the notion of a *focus*, i.e., the right-hand side non-terminal that is expanded ("varied") next. The predicate *mark/3* in Figure 2 precisely enumerates all possible foci for branch and unfolding coverage in an expression (or an entire production). In the case of BC, all expressions that involve a form of choice are foci. In the case of UC, all expressions that denote a nonterminal occurrence are foci. In *mark(C,X1,X2)*, *C* is the name of the coverage criterion (*bc* or *uc*), *X1* is the original expression, *X2* is *X1* updated so that one subterm contains a focus that is marked by enclosing it in {...}.

```
mark(C,p(L,N,X1),p(L,N,X2)) ⇐
    mark(C,X1,X2).
```

Marked productions are essentially marked expressions.

```
mark(uc,n(N),{n(N)}).
mark(bc,';'(Xs),{';'(Xs)}).
mark(bc,'?'(X),{'?'(X)}).
mark(bc,'*'(X),{'*'(X)}).
mark(bc,'+'(X),{'+'(X)}).
```

A nonterminal occurrence provides a focus for unfolding coverage. The EBNF forms ';', '?', '*', '+' provide foci for branch coverage.

```
mark(C,'?'(X1),'?'(X2)) ⇐
    mark(C,X1,X2).
mark(C,'*'(X1),'*'(X2)) ⇐
    mark(C,X1,X2).
mark(C,'+'(X1),'+'(X2)) ⇐
    mark(C,X1,X2).
```

Foci for BC and UC may also be found by recursing into subexpressions.

```
mark(C,','(Xs1),','(Xs2)) ⇐
    append(Xs1a,[X1|Xs1b],Xs1),
    append(Xs1a,[X2|Xs1b],Xs2),
    mark(C,X1,X2).

mark(C,';'(Xs1),';'(Xs2)) ⇐
    append(Xs1a,[X1|Xs1b],Xs1),
    append(Xs1a,[X2|Xs1b],Xs2),
    mark(C,X1,X2).
```

Sequences and choices combine multiple expressions, and foci are found by considering one subexpression at the time. (Marking is designed to be nondeterministic here.)

Fig. 2. Marking foci for branch and unfolding coverage

The remarkable property of this uniform specification is that it facilitates effectively systematic test data generation for the coverage criteria *BC* and *UC* in the sense that a generation algorithm may simple iterate over the extension of the predicate and exercise all options for any marked focus.

3.3 Generation Primitives

Generation algorithms for the five coverage criteria can be composed from a small set of primitives; one of which is the predicate *mark/3* described above. These are the remaining ones; we include mode annotations for the intended direction of usage.[3]

- ⋄ *complete(+G,+X,−T)*: we follow the standard definition [18,22]; the tree T is the shortest completion of expression X according to grammar G.
- ⋄ *mindepth(+G,+N,−D)*: the natural number D is the minimum depth of derivation trees rooted by nonterminal N according to grammar G in terms of the nonterminal nodes on paths—this is the essential relationship for shortest completion and possibly further generation algorithms; it can be computed by a simple fixed point computation.
- ⋄ *hole(+G,+N,+H,−T,−V)*: the tree T is rooted in nonterminal N with a "hole" for a derivation tree for nonterminal H where the hole is accessible through the place holder (logical variable) V—the tree is the smallest one in the sense of the shortest path from N to H (in terms of nonterminal nodes) while using shortest completion everywhere else.
- ⋄ *dist(+G,+N_1,+N_2,−D)*: the natural number D is the (minimum) distance between nonterminals N_1 and N_2 in the sense of nonterminal nodes on paths in derivation trees from N_1 to N_2—this is the essential relationship for smallest trees with holes; it can be computed by a simple fixed point computation similar to *mindepth/3*.
- ⋄ *vary(+G,+X,−T)*: the expression X contains exactly one focus ($\{\ldots\}$) and trees T are enumerated such that they are shortest completions overall, but all "immediate options" for the focus are exercised.

Figure 3 lists the specification of *vary/3*; it uses the primitive *complete/3* and a trivial relation *def(+G,?N,−Ps)* between grammars G, nonterminals N, and productions Ps such that N is a nonterminal defined by the grammar G according to the productions Ps.

3.4 Generation Algorithms

We are ready to define algorithms for the coverage criteria *TC*, *NC*, *PC*, *BC*, and *UC*. We leverage the primitives mentioned above. See Figure 4 for the specification of the algorithms. The simple specifications generate larger sets of smaller trees that achieve coverage in the intended manner, due to the focus of the primitives on minimality (i.e., shortest completion, smallest tree, etc) and the individual expansion of the foci via *mark/3*. For instance *uc/3* with arguments *uc(+G,?R,−T)* generates (by backtracking) derivation trees T for nonterminals R from grammar G. It is important to notice that the predicates of Figure 4

[3] The modes "+" and "−" are used for (instantiated) input or (uninstantiated) output arguments, respectively. In principle, there is also the mode "?" for unconstrained arguments.

```
vary(G,{n(N)},n(P,T)) ⇐
    def(G,N,Ps),
    member(P,Ps),
    P = p(_,_,X),
    complete(G,X,T).
```

A nonterminal occurrence in focus is varied so that all productions are exercised. (The complete spec also deals with chain productions and top-level choices in a manner that increases variation in a reasonable sense.)

```
vary(G,{';'(Xs)},';'(X,T)) ⇐
    member(X,Xs),
    complete(G,X,T).
```

A choice in focus is varied so that all branches are exercised.

```
vary(_,{'?'(_)},'?'([])).
vary(G,{'?'(X)},'?'([T])) ⇐
    complete(G,X,T).
vary(_,{'*'(_)},'*'([])).
vary(G,{'*'(X)},'*'([T])) ⇐
    complete(G,X,T).
vary(G,{'+'(X)},'+'([T])) ⇐
    complete(G,X,T).
vary(G,{'+'(X)},'+'([T1,T2])) ⇐
    complete(G,X,T1),
    complete(G,X,T2).
```

An optional expression and a '*' repetition in focus are varied so that the cases for no tree and one tree are exercised. A '+' repetition is varied so that the cases for sequences of length 1 and 2 are exercised.

We omit all clauses for recursing into compound expressions; they mimic shortest completion but they are directed in a way that they reach the focus.

Fig. 3. Varying foci for branch and unfolding coverage

```
tc(G,R,T)
  ⇐ def(G,R,_), complete(G,n(R),T).
nc(G,R,T)
  ⇐ def(G,R,_), dist(G,R,H,_), hole(G,n(R),H,T,V), complete(G,n(H),V).
pc(G,R,T)
  ⇐ def(G,R,Ps), member(P,Ps), complete(G,P,T).
pc(G,R,T)
  ⇐ def(G,R,_), dist(G,R,H,_), hole(G,n(R),H,T,V), pc(G,H,V).
bc(G,R,T)
  ⇐ cdbc(bc,G,R,T).
uc(G,R,T)
  ⇐ cdbc(uc,G,R,T).
cdbc(C,G,R,T)
  ⇐ def(G,R,Ps), member(P,Ps), mark(C,P,F), vary(G,F,T).
cdbc(C,G,R,T)
  ⇐ def(G,R,_), dist(G,R,H,_), hole(G,n(R),H,T,V), cdbc(C,G,H,V).
```

Fig. 4. Enumeration of test data achieving coverage

iterate over all possible nonterminals for the root R of the generated trees (assuming R is left uninstantiated). This implies that we can generate test data sets that are indexed by the nonterminals of the grammar; see again §2.

Let us pick one generation algorithm for discussion. For instance, predicate *pc/3* enumerates trees achieving PC as follows. The first clause of *pc/3* models the case that we want to cover a production P of the rooting nonterminal R, in which case we simply apply shortest completion to P. The second clause of *pc/3* models the case that we want to cover a production of some nonterminal H that is only reachable through a nonempty path starting from the rooting nonterminal R, in which case we create a tree with a hole for nonterminal H to be filled by recursive invocation of *pc/3*.

4 Grammar Nonequivalence Study: Java 5

In this study, we apply symmetric grammar comparison to four different grammars, in fact, parsers of the Java programming language. That is, we generate test data for all the grammars, and each test case from each of the test sets is then fed into each of the parsers. In this manner, we discover differences between the languages generated by the four grammars. (All involved grammars and tools are available online; see the footnote on the first page.)

4.1 Grammar Sources

In previous work, we have extracted Java grammars from the Java Language Specification (JLS) [4], with many inconsistencies and irregularities reported in [17]. These Java grammars appear to be a good target for grammar comparison, yet a significant grammar recovery effort would be needed to make those grammars executable. In fact, this recovery process would involve judgment calls that possibly bring the executable grammar further away from the JLS.

It turns out though that several handmade, executable adaptations of the JLS already exist and are deployed in practice. Thus, in the current work we acquired four operational grammars for J2SE 5.0 ("Java 5") from four widely used ANTLR sources, distributed under the BSD license. The underlying ANTLR-based parser descriptions strive to cover the same language; they were developed independently from one another by different grammar engineers, based on their experience, style and understanding of the JLS [4]:

	Technology	Author	Year	PROD	VAR	TERM
Habelitz	ANTLR3[4]	Dieter Habelitz[5]	2008	397	226	166
Parr	ANTLR3	Terence Parr[6]	2006	425	151	157
Stahl	ANTLR2[7]	Michael Stahl[8]	2004	262	155	167
Studman	ANTLR2	Michael Studman[9]	2004	267	161	168

[4] `http://www.antlr.org`

[5] `http://www.antlr.org/grammar/1207932239307/Java1_5Grammars/Java.g`

[6] `http://www.antlr.org/grammar/1152141644268/Java.g`

[7] `http://www.antlr2.org`

[8] `http://www.antlr.org/grammar/1093454600181/java15-grammar.zip`

[9] `http://www.antlr.org/grammar/1090713067533/java15.g`

PROD, VAR and TERM values in the table refer to simple grammar metrics [21] of the number of top alternatives in grammar production rules, the number of nonterminal symbols, and the number of terminal symbols. We have developed a simple infrastructure for driving a set of ANTLR-based parsers including aspects of parser generation and selecting the appropriate ANTLR version.

4.2 Grammar Extraction

Based on previous work on grammar convergence [16], we were able to extract the context-free grammars from the ANTLR-based parser description. That is, we developed a designated extractor, using the Rascal [11] meta-programming language, so that the following ANTLR constructs are abstracted away:

- ◇ Semantic actions — `{...}`
- ◇ Rule arguments — `[...]`
- ◇ Semantic predicates — `{...}?`
- ◇ Syntactic predicates — `(...)=>`
- ◇ Rewriting rules — `-> ^(...)`
- ◇ Return types of the rules — `returns ...`
- ◇ Specific sections — `options`, `@header`, `@members`, `@rulecatch`, ...
- ◇ Rule modifiers — `options`, `scope`, `@after`, `@init`, ...

Also some minor notational features like character class negation (~) or range operator (`..`) needed to be translated into basic context-free grammar notation. Tokens defined as terminals were merged with the normal grammar rules. By doing so, we are able to fit most of the grammar knowledge in our infrastructure without focusing on idiosyncratic details. An abstracted grammar differs from the original in terms of the accepted language, and these effects are yet to be fully studied; see §2.

4.3 Test Set Generation

Using the algorithm and the infrastructure described in §3, we generated test data for (only) the start symbols of each of the Java grammars. Figure 5 reports on the amount of test data. As an exercise in studying the effectiveness of the different coverage criteria, we explicitly divided test data based on the coverage criteria, and ultimately found out that the CDBC set contains the largest number of test cases and usually includes TC, PC, NC and BC sets.

Trivial coverage only involves one test case (rooted in the start symbol). One may expect that the shortest completions of all grammars are mutually accepted by the parsers. The test sets for production and nonterminal coverage yield the same test sets because of ANTLR-implied[10] and author-specific grammar style. The way *BC* and *UC* (and hence *CDBC*) are defined, the corresponding test sets need not to imply *PC* and *NC*, but, in practice, the implication holds. Hence, for the rest of the paper, we use test sets of CDBC for drawing actual conclusions on grammar comparison.

[10] For instance, definitions of nonterminals in ANTLR have exactly one production because choices are used instead of multiple productions.

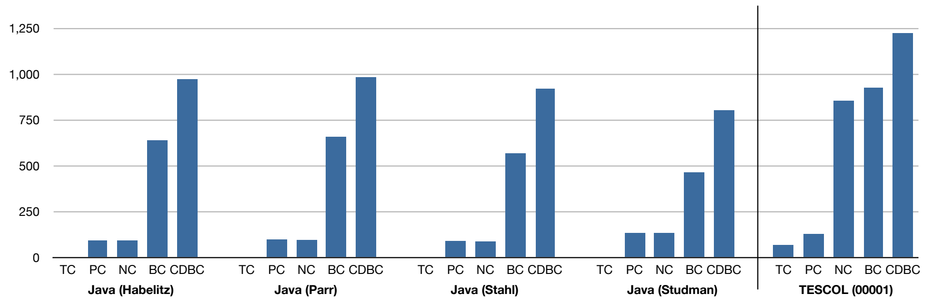


Fig. 5. Test set sizes. Amount of test data generated to satisfy trivial, production, nonterminal, branch and context-dependent branch coverage criteria. For comparison, we also show test set sizes for a much smaller grammar of the study in §6 in which case test sets were generated for all nonterminals as opposed to only the start symbol of the Java grammars.

4.4 Results

Figure 6 reports on the degree of observed nonequivalence during testing. The blue dots represent acceptance rate for each of the criteria-driven subsets, while the green block behind them reports on all test data together. Let us first examine the diagonal plots which are expected to be equal to 100%, not just close to it. Namely, consider one of the test cases generated from but not parseable with the Habelitz grammar:

```
    class a { { switch ( ++ this ) { } } }
```

According to the extracted grammar, switch block labels are defined by a nillable nonterminal aptly called `switchBlockLabels`:

```
switchBlockLabels:
        switchCaseLabels switchDefaultLabel? switchCaseLabels
switchDefaultLabel:
        DEFAULT COLON blockStatement*
switchCaseLabels:
        switchCaseLabel*
```

However, the original parser specification contained an AST rewriting rule:

```
switchBlockLabels
    :   switchCaseLabels switchDefaultLabel? switchCaseLabels
        ->  ^(SWITCH_BLOCK_LABEL_LIST switchCaseLabels
              switchDefaultLabel? switchCaseLabels) ;
```

This rule raises an exception if an attempt is made to rewrite an empty tree, and the unhandled exception is then treated as a failure to parse code. Since the context-free part allows `switchBlockLabels` to be ε, generated test data

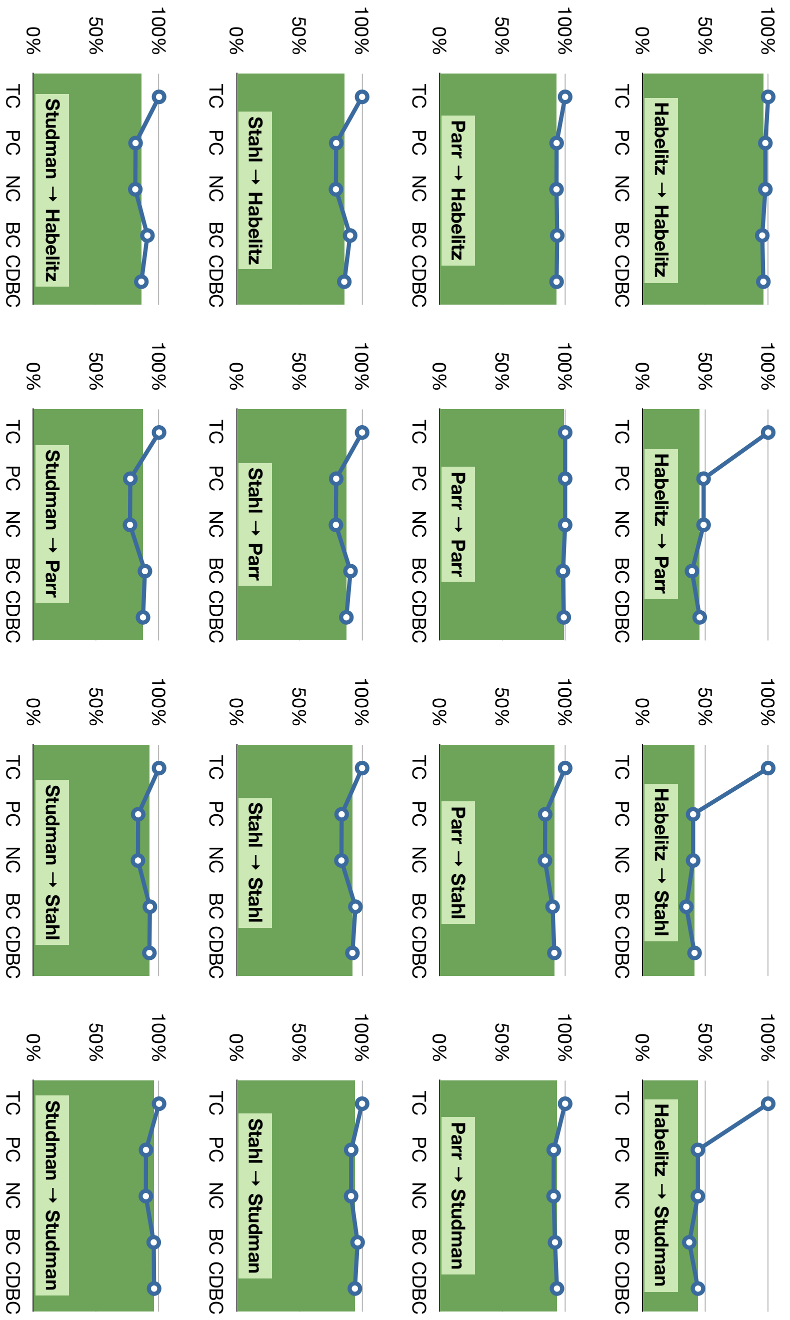


Fig. 6. Testing Java grammars and parsers. The Habelitz grammar is apparently much more permissive than the rest. All parsers accept almost all test cases generated from their corresponding grammars (diagonal plots).

explores the option, but the idiosyncrasy with which its structure was originally defined, leads to false nonequivalence reports. It is also worth mentioning that the grammar with the highest self-acceptance rate (99%) is Parr, which was designed by the creator of the ANTLR notation.

From the non-diagonal plots of Figure 6 one can see that the Parr, Stahl and Studman grammars are rather close to one another, but the Habelitz grammar is much more permissive. Indeed, manual cursory examination of the failing test cases shows that the Habelitz parser accepts, among other things:

- ◇ `class a < a extends a {}`, `class a < a >> {}`, `class a < a >>> {}` (the piece of grammar dealing with angle brackets is annotated with a "dirty trick" comment)
- ◇ `native class a { }` ("native" is a modifier for a method, not for a class)
- ◇ `@ a ( ++ 0 )` (annotation followed by neither class nor package declaration)

The last mentioned example is responsible for most of the failures. In fact, the only place we were able to spot where the Habelitz grammar is more restrictive than the rest is enumeration definitions (it does not allow for empty enumerations).

5 Matching Algorithm

Nonterminals of two given grammars are to be matched. We assume that the grammars are executable in that corresponding parsers are available. In fact, the grammars may have been extracted from the parsers—as discussed above. We start from a test set indexed by nonterminals of one grammar. We apply the parser of the other grammar to the indexed test set while also varying the start symbol so that all nonterminals are exercised. For each parser run with one test case, we get a positive response (meaning that this particular test case has been accepted as valid according to a particular nonterminal) or a negative one (meaning that a parse error occurred, AST building failed, a syntactic or semantic predicate did not hold, etc.).

We can group these results into triples {reference nonterminal, nonterminal under test, percentage of successfully parsed test data}. Such a relation, when displayed in table form with reference nonterminals as rows and nonterminals under test as columns, and when sorted alphabetically, looks like Figure 7 (left). Cells with 0% successes are left blank, up to 25% are yellow, below 75% are blue, up to 99% are green and exactly 100% successes are red.

The results are processed further by making actual matches between nonterminals. First, **universal**$(\cdot, y)$ matches are made by removing nonterminals under test that accept all test data generated by more than 75% of the reference nonterminals. Then, different rules for matching are attempted exhaustively. Each single match is recorded and the matched nonterminals are removed from further checks for the rest of the matching loop. There are the following rules for matches; these options are attempted in the given order for each matching step:

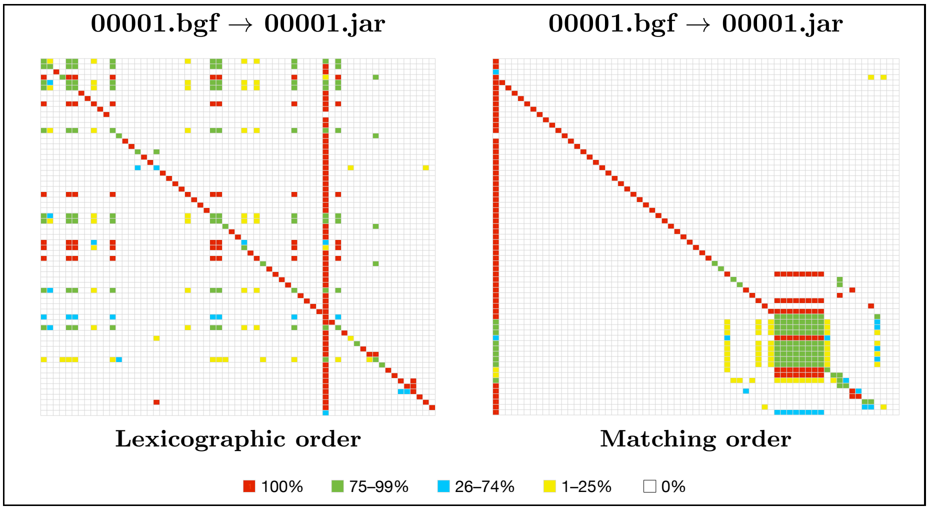


Fig. 7. Visualized nonterminal matching. In every color matrix, each row represents a producing nonterminal and each column denotes an accepting nonterminal. On the left color matrix, nonterminals (i.e., rows and columns) are sorted alphabetically; on the right one, in the order of matching.

void$(x, \cdot)$ all nonterminals under test accept less than 25% for x's test data.
perfect(x, y) x generates test data which can always be parsed by y and never by any other nonterminal, and y also exclusively accepts only x's test data;
nearlyPerfect(x, y) x generates test data of which more than 75% can be parsed by y and never by any other nonterminal, and y also exclusively accepts only x's test data;
exclusive(x, y) x generates test data which is best parsed by y at more than 75%, and y exclusively accepts only x's test data;
probable(x, y) x generates test data which is parsed only by y, and acceptance rate is at least 25%;
block(x_i, y_i) all x_i yield test data that is well accepted ($> 75\%$) by all y_i;
probableBlock(x_i, y_i) all x_i yield test data accepted at $> 25\%$ by all y_i;
maximum(x, y) of all candidates, y has the highest acceptance rate.

If any nonterminals are left once the above rules have been exhausted, then that rest is assumed to match **none**$(x, \cdot)$. If rows and columns of the relation are resorted in the order of matching, we can see a picture like the one on Figure 7 (right). There we see a **universal** match being made, followed by a long series of **perfect** and then **nearly perfect** matches, several **exclusive** matches, a big **block** match and some less reliable matches at the end of the process.

6 Nonterminal Matching Study: Course Work

TESCOL (TESt COmpiler Language) is an artificial small programming language used by the first coauthor in a compiler engineering course. A TESCOL program contains a list of semicolon-separated declarations and a single statement. The program starts with the keyword `trolley`, followed by a constant identifier, the keyword `contains`, and the declarations. The statement is separated from the declarations by the keyword `checkout` and followed by a semicolon, the mandatory `done` and another semicolon. There are also some contextual restrictions: global naming scheme, non-recursive procedures, declarations preceding uses, etc.

A class of students was asked to implement TESCOL in ANTLR, resulting in a codebase of 83 grammars claiming to conform to the same language specification. The following actions were part of the preparations of the TESCOL grammar base:

- ◇ ANTLR3 grammars were recovered from the submitted tarballs;
- ◇ The grammars were extracted as described in §4.2;
- ◇ We generated boilerplate Java code for passing a file name and a nonterminal name as parameters for parser runs;
- ◇ The code produced by ANTLR from the grammar was compiled together with the boilerplate code to form a JAR;
- ◇ The filenames were obfuscated to avoid disclosing students' identities.

In this way we were able to obtain 32 pairs, each consisting of a valid context-free grammar and a runnable JAR with a parser. Each of the remaining 51 grammars contained small errors in the ANTLR productions or the expected interaction protocol which prevented their automated processing in the study of this paper. Each of the working grammars was used to generate test data for all nonterminals it contained. Such a test data set for one grammar consisted of around 1000 test cases (min. 599, max. 1354), distributed among coverage criteria as shown in Figure 5 (right). One test data set took around 5 hours to test against all 2300 nonterminals of available 32 candidate grammars on an Intel Core i7 machine with a 2.80GHz CPU; see also Table 3. The results reported in this paper refer specifically to one test data set for the reference grammar nicknamed 00001, fed into all of the available parsers. The choice of 00001 over other TESCOL grammars was purely incidental.

TESCOL grammars are considerably smaller than Java grammars, having on average four times less top alternatives, three times less nonterminal symbols and almost half less terminals (compare with the values in the table on page 332):

	PROD	VAR	TERM
Minimum	69	54	101
Average	85	67	104
Maximum	126	83	120

Let us return to Figure 7, which we already used for illustration of nonterminal matching. In fact, the two matrices in the figure represent matches of the reference grammar against its own parser. The only universal match is with a non-

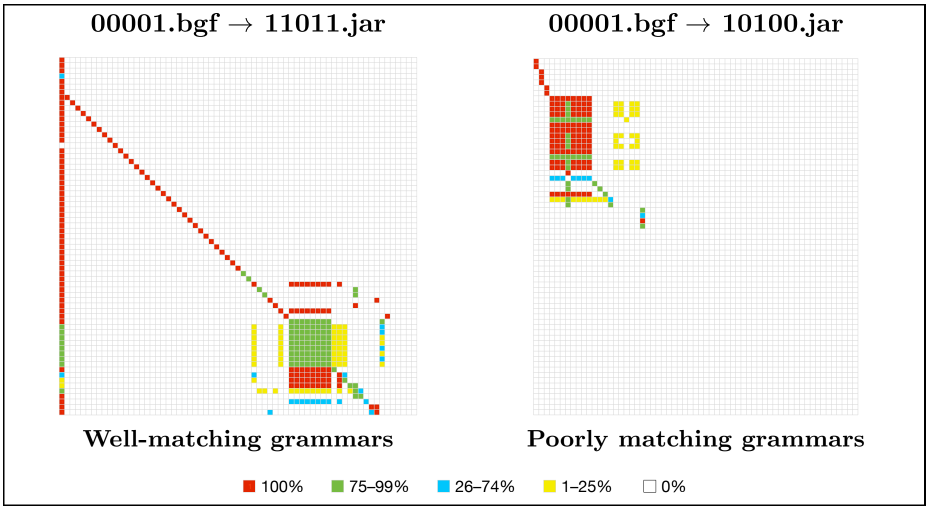


Fig. 8. Visualized nonterminal matching. A good match between languages can be seen on the left; a considerably worse one on the right.

terminal called `token`, which serves error handling. Void matches for `comment`, `COMMENT` and `WS` (whitespace) make sense because of the way how a parser handles, in fact, skips such lexical categories. However, a void match for `procDec` is suspicious; when investigated, we see the same problem encountered earlier in §4.4: a `RewriteEmptyStreamException`.

Nominal inspection of all 50 singular matches shows that they are correct. There are also two group matches: one correct (comprising `expr`, `multExpr`, `compExpr`, `andExpr`, etc, closely related nonterminals from one grammatical level) and one incorrect (`constDec` and `declarations` with themselves). The incorrectness of the latter is a direct consequence of the problem with `procDec`.

Figure 8 shows two more examples of nonterminal matching which we will discuss very briefly. The one on the left is well-matched, with a couple of groups and many perfect matches, most of which could not have been inferred from nominal matching: `MULTI` with `ARITH-MUL`, `NEQ` with `COND-NONEQUAL`, `grstatement` with `statement-group`, etc. The one on the right is matching rather poorly, with 41 nonterminals matching **void** or **none** and the rest being in **block**s.

We have condensed the results of matching all grammars with the reference grammar in Figure 9, where matches are counted based on their type. **Universal**, **void** and **none** belong to a group of usually unwanted matches since they fail to provide any information to the grammar engineer. On the other end, **block** and **probable block** matches give some information which requires more sophisticated heuristics or human interpretation. The remaining matches are singular: one reference nonterminal matches with one nonterminal under test. As it becomes apparent from the diagram, **perfect**, **nearly perfect**, **exclusive**,

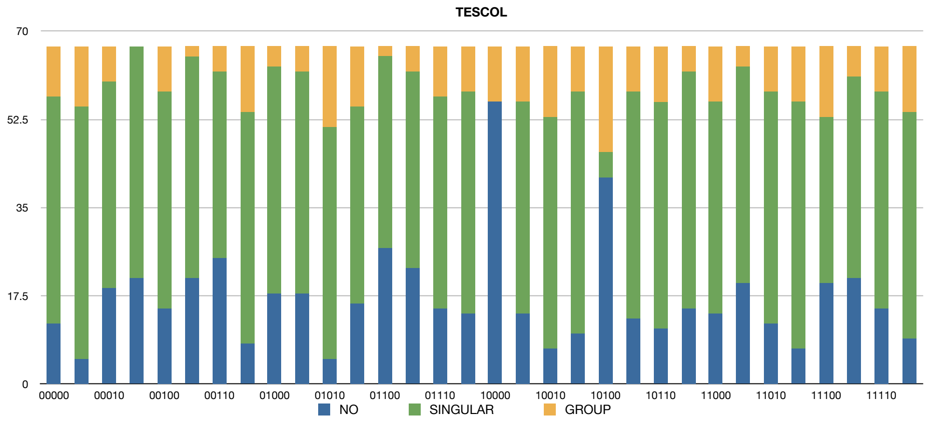


Fig. 9. TESCOL nonterminal matching. Blue (dark grey) bar parts denote nonterminals that did not match anything (universal, void, none); green (grey) denotes nonterminals for which a match was found (perfect, nearly perfect, exclusive, probable, maximum); yellow (light grey) is for nonterminals which were matched in a group (block, probable block).

probable and **maximum** matches cover the majority of reference nonterminals. Group matches also provide useful and adequate results. Hence, nonterminal matching is successful in the context of the study.

7 Related Work

§2 already provided some general background on the established topic of grammar-based testing; we refer to [3,7,8,12,14,15,18,19,20,25] for extensive discussion of methods and applications of grammar-based testing. Our work is original in so far that we are the first to actually use grammar-based testing for the comparison of grammars. Usually, grammar-based testing is used to test parsers or compilers.

In both studies in §4 and §6, we have noticed imperfect self-matching and explained reasons for it. One of the ways to improve on this issue would be to take into account the constraints expressed by the parser specification. There are related methods of extending grammar-based testing to attribute grammars [6,10].

In our current development, we do not yet leverage any sort of negative test data generation. There are grammar-based testing scenarios that clearly benefit from inclusion of negative test cases [29]. For instance, a parser for which no grammar-based parser description is available can only be tested for *completeness* with regard to reference grammar with positive test cases whereas testing for *correctness* would require negative test cases. In our comparison-based context of the present paper, negative test data is "less important" because evidence of both non-completeness and non-correctness can be found with the help of positive test cases that are obtained from the compared grammars; see again §2.

Grammar nonequivalence is a well-known undecidable problem. One related problem is the status of a grammar to be ambiguous (or not). Some sort of testing has been successfully applied though in this context [2]. Another related problem is grammar-class/non-ambiguity preservation under composition. While context-free grammars can always be combined together to form new context-free grammars, smaller subclasses related to specific parsing technology (or to the requirement of non-ambiguity) usually do not exhibit this property. Several attempts to provide painless language modularity are known, such as Kiama [26], Silver/Copper [28], language boxes methodology [23], etc. Grammar comparison-like methods may be potentially useful in supporting safe composition.

8 Conclusion

We have developed and demonstrated an approach to grammar comparison which relies on systematic grammar-based test data generation and parsing. We have shown, in particular, that the approach can be used for revealing differences between substantially large grammars and for matching many grammars. We conclude with a discussion of future work.

The results of nonterminal matching turn out to be useful based on our nominal inspection. Further research is needed to see how the information that is derived from nonterminal matching can be usefully consumed by grammar engineers for different scenarios. For instance, someone who likes to converge two grammars may need to turn the matches into appropriate transformations.

We already mentioned the possibility of generating negative test cases. In theory, more evidence can be found by applying parsers to negative test cases. Whether or not this evidence makes a difference in practical scenarios like ours is an open question.

There is also the related question whether we can improve precision of matching by generating larger test sets for more demanding coverage criteria. While it may lead to bad scalability to universally replace CDBC by a more demanding criterion, a more selective approach could be scalable enough: generate more test data when about to match a block; see §5.

Our implementation leaves much room for optimization. As apparent from Table 3, the generation phase is not a problem: it is required only once, and takes only a few minutes. However, our current infrastructure for parser execution loops over test cases such that the parser is run separately for each test case, causing excessive overhead with loading and unloading in the JVM. The computation of the results of the present paper relied on parallelism/distribution.[11]

[11] We used several machines at the CWI SWAT department. The estimated, sequential time to run all TESCOL-based test data against all parsers is 300 days.

Table 3. Performance. Time (in minutes, seconds and, if necessary, hours) to generate test data, unparse it (turn parse trees to source code), and run. Generation was measured separately for satisfying trivial, production, nonterminal, branch and context-dependent branch coverage criteria.

	generate					unparse	run			
Test set	TC	PC	NC	BC	CDBC		Habelitz	Parr	Stahl	Studman
Habelitz	00:21	00:58	00:59	02:14	04:46	00:30	02:29	02:02	01:23	01:20
Parr	00:08	00:29	00:29	02:10	03:51	00:34	02:50	02:21	01:33	01:34
Stahl	00:08	00:35	00:35	02:45	05:01	00:39	03:02	02:34	01:40	01:39
Studman	00:09	00:38	00:39	02:59	05:12	00:37	03:05	02:35	01:41	01:41

	TC	PC	NC	BC	CDBC	unparse	00000	00001
00000	00:31	00:47	00:50	00:59	01:27	00:57	5:08:48	4:40:23
00001	00:05	00:14	00:51	01:12	01:53	01:47	5:41:22	5:10:36
...						...		
All TESCOL	02:21	08:44	27:21	34:21	59:19	17:32	—	

Acknowledgments. Bernd Fischer was supported by EPSRC grant EP/F052669/1. Robert van Liere (CWI, Amsterdam) provided expert advice on visualization.

References

1. Aho, A.V.: Teaching the Compilers Course. SIGCSE Bull. 40, 6–8 (2008)
2. Basten, H.J.S.: Tracking Down the Origins of Ambiguity in Context-Free Grammars. In: Cavalcanti, A., Deharbe, D., Gaudel, M.-C., Woodcock, J. (eds.) ICTAC 2010. LNCS, vol. 6255, pp. 76–90. Springer, Heidelberg (2010)
3. Burgess, C.J.: The Automated Generation of Test Cases for Compilers. Software Testing, Verification and Reliability 4(2), 81–99 (1994)
4. Gosling, J., Joy, B., Steele, G.L., Bracha, G.: The Java Language Specification, 3rd edn. Addison-Wesley (2005), all versions of the JLS are available at `http://java.sun.com/docs/books/jls`
5. Griswold, W.G.: Teaching Software Engineering in a Compiler Project Course. Journal on Educational Resources in Computing 2 (December 2002)
6. Harm, J., Lämmel, R.: Two-dimensional Approximation Coverage. Informatica 24(3) (2000)
7. Hennessy, M., Power, J.F.: Analysing the Effectiveness of Rule-coverage as a Reduction Criterion for Test Suites of Grammar-based Software. Empirical Software Engineering 13, 343–368 (2008)
8. Hoffman, D., Wang, H.Y., Chang, M., Ly-Gagnon, D., Sobotkiewicz, L., Strooper, P.: Two Case Studies in Grammar-based Test Generation. Journal of Systems and Software 83, 2369–2378 (2010)
9. IBM Corporation: VS COBOL II Application Programming Language Reference, 4th edn. (1993), Publication number GC26-4047-07
10. Kastens, U.: Studie zur Erzeugung von Testprogrammen für Übersetzer. Bericht 12/80, Institut für Informatik II, University Karlsruhe (1980)

11. 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)
12. Kossatchev, A.S., Posypkin, M.A.: Survey of Compiler Testing Methods. Programming and Computing Software 31, 10–19 (2005)
13. Lämmel, R., Verhoef, C.: VS COBOL II grammar Version 1.0.4 (1999), http://www.cs.vu.nl/grammarware/browsable/vs-cobol-ii/
14. Lämmel, R.: Grammar Testing. In: Hussmann, H. (ed.) FASE 2001. LNCS, vol. 2029, pp. 201–216. Springer, Heidelberg (2001)
15. Lämmel, R., Schulte, W.: Controllable Combinatorial Coverage in Grammar-Based Testing. In: Uyar, M.Ü., Duale, A.Y., Fecko, M.A. (eds.) TestCom 2006. LNCS, vol. 3964, pp. 19–38. Springer, Heidelberg (2006)
16. 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)
17. Lämmel, R., Zaytsev, V.: Recovering Grammar Relationships for the Java Language Specification. Software Quality Journal 19(2), 333–378 (2011)
18. Malloy, B.A., Power, J.F.: An Interpretation of Purdom's Algorithm for Automatic Generation of Test Cases. In: 1st Annual International Conference on Computer and Information Science, pp. 3–5 (2001)
19. Maurer, P.: Generating Test Data with Enhanced Context-free Grammars. IEEE Software 7(4), 50–56 (1990)
20. McKeeman, W.M.: Differential Testing for Software. Digital Technical Journal of Digital Equipment Corporation 10(1), 100–107 (1998)
21. Power, J.F., Malloy, B.A.: A Metrics Suite for Grammar-based Software. Journal of Software Maintenance and Evolution: Research and Practice 16, 405–426 (2004)
22. Purdom, P.: A Sentence Generator for Testing Parsers. BIT 12(3), 366–375 (1972)
23. 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)
24. Schwartzbach, M.I.: Design Choices in a Compiler Course or How to Make Undergraduates Love Formal Notation. In: Hendren, L. (ed.) CC 2008. LNCS, vol. 4959, pp. 1–15. Springer, Heidelberg (2008)
25. Sirer, E.G., Bershad, B.N.: Using Production Grammars in Software Testing. SIGPLAN Notices 35, 1–13 (1999)
26. Sloane, A.M., Kats, L.C.L., Visser, E.: A Pure Object-Oriented Embedding of Attribute Grammars. In: Ekman, T., Vinju, J. (eds.) Proceedings of the Ninth Workshop on Language Descriptions, Tools, and Applications (LDTA 2009). Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers (2009)
27. Waite, W.M.: The Compiler Course in Today's Curriculum: Three Strategies. In: Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education, SIGCSE 2006, pp. 87–91. ACM (2006)
28. Van Wyk, E., Krishnan, L., Bodin, D., Schwerdfeger, A.: Attribute Grammar-Based Language Extensions for Java. In: Bateni, M. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 575–599. Springer, Heidelberg (2007)
29. Zelenov, S., Zelenova, S.: Automated Generation of Positive and Negative Tests for Parsers. In: Grieskamp, W., Weise, C. (eds.) FATES 2005. LNCS, vol. 3997, pp. 187–202. Springer, Heidelberg (2006)

RLSRunner: Linking Rascal with K for Program Analysis

Mark Hills[1,2], Paul Klint[1,2], and Jurgen J. Vinju[1,2]

[1] Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
[2] INRIA Lille Nord Europe, France

Abstract. The Rascal meta-programming language provides a number of features supporting the development of program analysis tools. However, sometimes the analysis to be developed is already implemented by another system. In this case, Rascal can provide a useful front-end for this system, handling the parsing of the input program, any transformation (if needed) of this program into individual analysis tasks, and the display of the results generated by the analysis. In this paper we describe a tool, RLSRunner, which provides this integration with static analysis tools defined using the K framework, a rewriting-based framework for defining the semantics of programming languages.

1 Introduction

The Rascal meta-programming language [13,12] provides a number of features supporting the development of program analysis tools. This includes support for processing the input program with a generalized parser and pattern matching over concrete syntax; developing the code for the analysis using flexible built-in types (e.g., sets, relations, lists, and tuples), pattern matching, user-defined algebraic data types, and higher-order functions; and displaying the analysis results interactively to the user through visualization libraries and through integration with the Eclipse IDE via IMP [4,5].

However, sometimes the analysis to be developed already exists, and there may be compelling reasons to use this analysis instead of rewriting it in Rascal. The existing analysis may be trusted, very complicated, or highly optimized, or may provide features not already available in Rascal. In these cases, instead of requiring the user to rewrite the analysis, Rascal can be used for its front-end capabilities (parsing, concrete syntax matching, Eclipse integration) while handing off analysis tasks to existing analysis tools.

This paper describes a tool, RLSRunner, which provides this integration with program analyses written using the K framework [10,16]. K is a rewriting-based, tool-supported notation for defining the semantics of programming languages. It is based on techniques developed as part of the rewriting logic semantics (RLS) project [15,14]. RLSRunner links languages defined in Rascal with formal language definitions written either directly in K and compiled to Maude [6], or with K-style definitions written directly in Maude.

Contributions. The direct contribution described here is the RLSRunner tool itself, providing a method to take advantage of K language definitions while building program analysis tools using Rascal. Indirectly, we believe that RLSRunner highlights the flexibility of Rascal, showing that there is no requirement for Rascal-only solutions. It also mitigates one of the limitations of the current K suite of tools, which are focused on abstract syntax-based program representations but provide only limited support for working with concrete syntax.

Roadmap. In Section 2 we describe the Rascal support created for interacting with external tools (in general) and K definitions in Maude (in particular). We then describe extensions to K language definitions needed to support interaction with Rascal in Section 3. This is put together in a case study in Section 4, presenting the pairing of Rascal with an existing analysis framework used to search for type and unit errors in programs written in a paradigmatic imperative language. Section 5 then briefly discusses related work, while Section 6 presents some thoughts for future work and concludes.

2 Supporting K Program Analysis in Rascal

Figure 1 provides an overview of the process of integrating languages defined in Rascal with program analysis semantics defined in K. The first step in this integration is to define the front-end components: a grammar for the language; a generated parser based on this grammar; and a program (the "Maude-ifier") that will take the generated parse tree for a program and transform it into a Maude-readable form. Section 4 provides a concrete example of this process. As a side-effect, defining the grammar also provides an IDE for the language, which can be further extended with additional features as needed.

Given the Maude-ified code, the Rascal RLSRunner library then provides the functionality, in conjunction with language-specific user code, to: generate the analysis tasks; send them to Maude; read the results; and parse these results to extract the overall results of the analysis. The main driver function for this process is the `runRLSTask` function, shown in Figure 2. `runRLSTask` provides the functionality needed to start, communicate with, and stop Maude. To do this it makes use of the ShellExec library. ShellExec includes a number of routines to start, read from, write to, and terminate processes running outside Rascal.

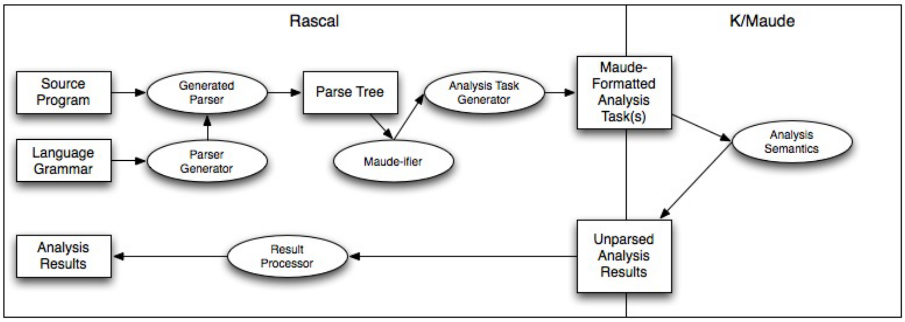


Fig. 1. Integrating Rascal and K

```
public RLSResult runRLSTask(loc ml, RLSRunner runner,
                            str input...)
{
  PID pid = startMaude(ml,runner.maudeFile);

  str inputStr = input[0];
  list[str] inputArgs = [ ];
  if (size(input) > 1)
    inputArgs = [ input[n] | n <- index(input)-0 ];

  str toRun = (runner.pre)(inputStr,inputArgs);

  writeTo(pid, toRun);
  str res = readFrom(pid);
  bool continueReading = true;
  while (continueReading) {
    if (/rewrites:\s*\d+/ !:= res && /Maude\>\s+$/ !:= res)
      res = res + readFrom(pid);
    else
      continueReading = false;
  }

  RLSResult rlsRes = (runner.post)(res);
  stopMaude(pid);
  return rlsRes;
}
```

Fig. 2. The `runRLSTask` Function, in Rascal

The execution of `runRLSTask` is customized for specific languages and analysis tasks using an `RLSRunner` value (in this case, `RLSRunner` is the name of a Rascal algebraic data type, not the library) passed as a parameter (`runner` in Figure 2). The `RLSRunner` includes the Maude definition of the K semantics to use for the analysis, and also includes pre- and post-processing functions, referred to in Figure 1 as "Analysis Task Generator" and "Result Processor", respectively (and in the code as `pre` and `post`). The pre-processing function takes the Maude-ified term, passed to `runRLSTask` as the first element of the `inputs` parameter, and pre-processes it. This generates the analysis task, or tasks, in the form of a Maude term (named `toRun` in the code). This term is written to the Maude process's input stream, where it will then be rewritten by Maude using the analysis semantics to perform the analysis. When Maude writes the final result of the analysis, this is read by `runRLSTask` on the process output stream.

Once the entire result is read, `runRLSTask` invokes the post-processing function. This function checks to see if the result format is one it can parse; if so, it returns the analysis results using a user-defined, analysis-specific constructor that extends type `RLSResult`. Two output formats are currently supported by the RLSRunner library, while more can be added as needed. Both default formats include performance information from Maude: total rewrites, rewrites per second, etc. In the first format, the result of the analysis is provided just as a string containing all the generated output (for instance, all error messages). This string is processed by the result processor (see Figure 1) to extract the analysis results. In the second format, error information is returned using a delimited format, made up of the location of the error, the severity of the error, and the error message. In conjunction with the `createMessages` function in the RLSRunner library, the result processor can extract the location, severity, and error message information, returning the analysis results as a set of items of Rascal type `Message`:

```
data Message = error(str msg, loc at)
             | warning(str msg, loc at)
             | info(str msg, loc at);
```

Using the `addMessageMarkers` function in the Rascal ResourceMarkers library, these messages can be added in Eclipse as `Problem` markers to the indicated locations, with a severity (`Error`, `Warning`, or `Info`) based on the results; these markers appear both in the editor (as red "squiggly" underlines for errors, for instance) and in the `Problems` view. Location information from the parse tree, passed in as part of the analysis task to Maude, is used in the messages to ensure that markers are added to the correct locations. Section 4 shows examples of this output marking in action.

If, instead, the post-processor cannot parse the result, it returns all the output from the analysis using a predefined `RLSResult` named `NoResultHandler`, indicating that some error occurred. Then, after any results (parsable or not) have been computed by `post`, `runRLSTask` stops the Maude process and returns the `RLSResult` item.

3 Rascal Support in K

To work with Rascal, K definitions need to meet two requirements. First, the output of a run of the semantics should be in a format parsable by the RLSRunner library. Second, the semantics should support Rascal source locations, allowing messages generated by the analysis to be tied back to specific source locations in the file being analyzed.

To support Rascal source locations in K semantics, an algebraic definition of Rascal locations is provided as an operator `sl` (for source location) that defines a value of sort `RLocation`. `sl` keeps track of the URI as well as the position information: offset, length, and starting and ending rows and columns (set to -1 if not available). `sl` is defined (using Maude notation) as follows:

```
fmod RASCAL-LOCATION is
  including STRING .
  including INT .
  sort RLocation .
  op sl : String Int Int Int Int Int Int -> RLocation .
endfm
```

These locations are then used by extending the abstract syntax for various language constructs, such as declarations, expressions, etc, adding new "located" versions of these constructs. For each construct `C` that we wish to extend, we define a new operator written as `locatedC`, taking both a `C` and an `RLocation`. For instance, assuming we have a sort for declarations named `Decl`, the located version is defined in Maude as:

```
op locatedDecl : Decl RLocation -> Decl .
```

To use these in the semantics, a new K cell, `currLoc`, is defined, holding the current source location. This cell is updated to the location given in a located construct when one is encountered during evaluation:

```
op currLoc : RLocation -> State .
eq k(decl(locatedDecl(D, RL)) -> K) currLoc(RL') =
   k(decl(D) -> rloc(RL') -> K) currLoc(RL) .
```

The equation shown above does the following: if we are in a state where a located declaration `D`, with source location `RL`, is the next evaluation step, and `RL'` is the current location, we change the current location to `RL`. We also set the next two computational steps, first evaluating the declaration `D` using whatever logic was already present, and then resetting the location back to `RL'`. This processes the declaration in the context of the source location given with the declaration, and then recovers the current location in case it is needed. The rule to handle `rloc`, which recovers a source location (while discarding the current location in the `currLoc` cell), is shown below:

```
op rloc : RLocation -> ComputationItem .
eq k(rloc(RL) -> K) currLoc(RL') = k(K) currLoc(RL) .
```

The need for, in essence, creating a location stack can be seen with constructs such as loops or conditionals, where the correctness of the (located) construct may depend on the correctness of (located) children. Without a mechanism to recover prior locations, the error message would instead have to be given in terms of the most recent location, which may not be the correct one.

4 Linking Rascal with the SILF Analysis Framework

SILF [10], the Simple Imperative Language with Functions, is a standard imperative language with functions, arrays, and global variables. Originally designed as a dynamically typed language, it has since been adapted to support experiments in defining program analysis frameworks, including type systems, using an abstract K semantics, with analysis information given in programs using function contracts and type annotations [11].

As mentioned in Section 2, the first step in linking Rascal with a K-defined analysis tool is to define the front-end components: a grammar for the language, the generated parser, and the Maude-ifier. As a running example, two rules in the grammar for SILF, defining the productions for addition (`Plus`) and for a function call (`CallExp`), are given as follows (`{Exp ","}*` represents a comma-separated list of expressions):

```
Exp = Plus: Exp "+" Exp
    | CallExp: Ident "(" {Exp ","}* ")";
```

In Maude, the abstract syntax is defined by defining algebraic operators, with each _ character representing a placeholder for a value of the given sort. The Maude versions of the two productions given above are shown below, with the first operator defining `Plus` and the second and third defining `CallExp`, one with parameters and one without:

```
op _+_ : Exp Exp -> Exp .
op _`(_`) : Id ExpList -> Exp .
op _`(`) : Id -> Exp .
```

The string generated by the Maude-ifier, created with Rascal code like the following, uses a prefix form of these operators, for instance using _+_ for the plus operation with the operands following in parens:

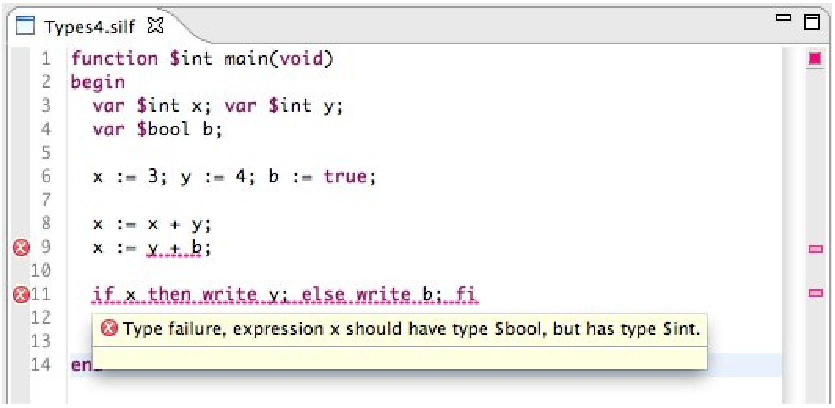
```
case (Exp)`<Exp el> + <Exp er>` :
  return located(exp,"Exp","_+_(<toMaude(el)>,<toMaude(er)>)");
```

Function `located` then builds the located version of the construct, discussed in Section 3, using the location associated with `exp` (the subtree representing the plus expression) to generate the correct values for the `sl` operator. An example of the Maude generated for the expression `1 + 2` is shown below (with the file URI replaced by `t.silf` for conciseness, and with the expression located on line 13 between columns 7 and 12):

```
locatedExp(_+_(
    locatedExp(#(1),sl("t.silf",166,1,13,7,13,8)),
    locatedExp(#(2),sl("t.silf",170,1,13,11,13,12))),
  sl("t.silf",166,5,13,7,13,12)))
```

Once we have the Maude-ified program with embedded location information, we can use the RLSRunner library, as described in Section 2, to run the analysis tasks generated for the language. In this case the pre-processing function is very simple, inserting the Maude-ified version of the program into a use of `eval` (with `nil` indicating that no input parameters are provided – they are only used in executions of the dynamic semantics). `red` is the Maude command to reduce a term, i.e., to perform the analysis:

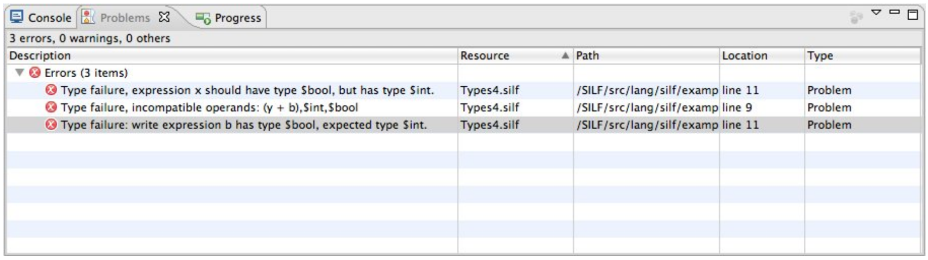
```
public str preCheckSILF(str pgm, list[str] params) {
  return "red␣eval((<pgm>),nil)␣.\n";
}
```



```
function $int main(void)
begin
  var $int x; var $int y;
  var $bool b;

  x := 3; y := 4; b := true;

  x := x + y;
  x := y + b;

  if x then write y; else write b; fi

en
```

Type failure, expression x should have type $bool, but has type $int.

Fig. 3. Type Error Markers in SILF



3 errors, 0 warnings, 0 others

Description	Resource	Path	Location	Type
Errors (3 items)				
Type failure, expression x should have type $bool, but has type $int.	Types4.silf	/SILF/src/lang/silf/examp	line 11	Problem
Type failure, incompatible operands: (y + b),$int,$bool	Types4.silf	/SILF/src/lang/silf/examp	line 9	Problem
Type failure: write expression b has type $bool, expected type $int.	Types4.silf	/SILF/src/lang/silf/examp	line 11	Problem

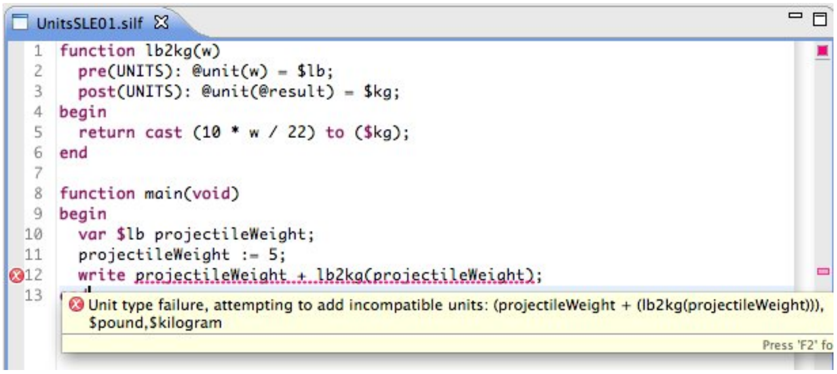
Fig. 4. Problems View with SILF Type Errors

The results are handled by the post-processing function in conjunction with functions in the RLSRunner library, extracting the analysis results as Rascal `Messages` as discussed in Section 3.

Two types of analysis are currently supported. The first is a standard type checking policy, with types provided using type annotations such as `$int`. Figure 3 provides an example of the results of a run of this analysis on a program with several type errors. As can be seen in the Eclipse Problems view, shown for the same file in Figure 4, there are three errors in this file. The first marked location in Figure 3 contains one of these errors, an attempt to add `y`, which is an integer, to `b`, which is a boolean. The second marked location actually has two errors. The guard of the `if` should be a boolean, not an integer; and the argument to `write` should be an integer, not a boolean. The actual output of Maude, parsed to extract this error information, is the following string (formatted to fit in the provided space, and with the actual URI replaced with `t.silf`):

```
Type checking found errors:
||1:::|t.silf::124::5::9::7::9::12|:::
  Type failure, incompatible operands: (y + b),$int,$bool||
||1:::|t.silf::134::35::11::2::11::37|:::
  Type failure, expression x should have type $bool,
  but has type $int.||
||1:::|t.silf::158::8::11::26::11::34|:::
  Type failure: write expression b has type $bool,
  expected type $int.||
```

The second analysis is a units analysis, which includes unit type annotations, assert and assume statements, loop invariants, and function contracts with preconditions, postconditions, and modifies clauses (useful because SILF supports global variables). An example of a run of this analysis, including a tooltip showing a detected error, is shown in Figure 5. The error is in the addition: variable `projectileWeight` is declared with pounds as the unit, while function `lb2kg` converts an input value in pounds (as specified in the precondition) to an output value in kilograms (as specified in the postcondition). When the units analysis examines the function call, it checks that the input satisfies the precondition (true) and then assumes the output result matches the postcondition. Because of this, the two operands have a different unit, so they cannot be added. Another example is shown in Figure 6, where the invariant fails to hold because of a mistake on line 10, where `y * x` was accidentally written instead of `y * y`, causing



UnitsSLE01.silf

```
function lb2kg(w)
  pre(UNITS): @unit(w) = $lb;
  post(UNITS): @unit(@result) = $kg;
begin
  return cast (10 * w / 22) to ($kg);
end

function main(void)
begin
  var $lb projectileWeight;
  projectileWeight := 5;
  write projectileWeight + lb2kg(projectileWeight);
```

Unit type failure, attempting to add incompatible units: (projectileWeight + (lb2kg(projectileWeight))), $pound,$kilogram

Press 'F2' fo

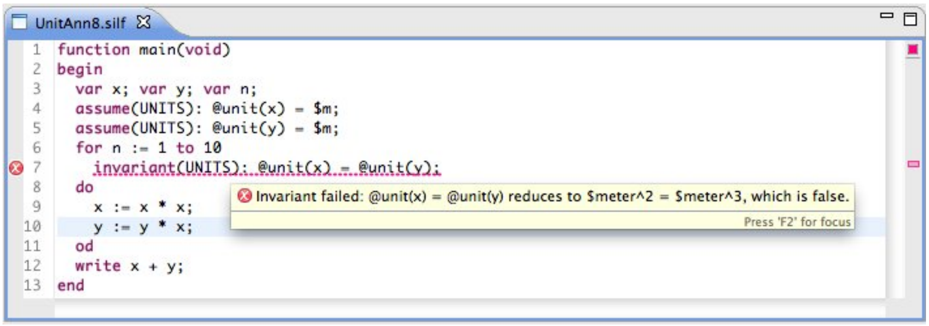
Fig. 5. Units Arithmetic Error in SILF Units Policy

the units of x and y to be out of sync. This error is detected after one abstract loop iteration, with the detected inconsistency between the units shown in the error message.

All the code for the integration between Rascal and K shown in this section is available in the Rascal subversion repository: links can be found at http://homepages.cwi.nl/~hills/RascalK .

5 Related Work

Many tools exist for working with, and potentially executing, formal definitions of programming languages. Generally these tools focus on defining a standard dynamic (evaluation) and maybe static (type) semantics, but, by choosing the appropriate domain, they could also be used for program analysis. Tools with support for some form of development environment include Centaur [2] (for operational semantics) and the Action Environment [17] (for action semantics). A number of approaches for defining semantics and program analyses using term rewriting have also been proposed [7]. For instance, the Meta-Environment [18] provides an open environment for designing and implementing term rewriting environments; one instantiation of this, the ASF+SDF Meta-Environment [19], provides a graphical environment for defining the syntax of a language in SDF, editing programs in the defined language, and performing reductions



UnitAnn8.silf

```
function main(void)
begin
  var x; var y; var n;
  assume(UNITS): @unit(x) = $m;
  assume(UNITS): @unit(y) = $m;
  for n := 1 to 10
    invariant(UNITS): @unit(x) = @unit(y);
  do
    x := x * x;
    y := y * x;
  od
  write x + y;
end
```

Invariant failed: @unit(x) = @unit(y) reduces to $meter^2 = $meter^3, which is false.

Press 'F2' for focus

Fig. 6. An Invariant Failure in SILF Units Policy

or transformations with ASF equations [20]. Much like in K, these equations can also be (and have been) used for program analysis.

Along with these, many static analysis tools, including those with some sort of graphical interface for the user, have also been developed, including Frama-C [1] and a number of tools that support JML notation [3].

The main distinctions between these tools and the work presented here are: the use of K for defining the program analysis semantics; the lack of a standard development environment for K (especially as compared to tools such as ASF+SDF); and the lack of a specific source language to analyze (as compared to tools such as JML, which focuses on Java). The work here attempts to address some of the limitations caused by the second point, providing a method to link IDEs developed in Rascal with an analysis semantics given in K, leading to a better experience for the analysis user. On the other hand, since we are focusing specifically on the second point, this work takes the use of K, and the interest in supporting multiple source languages, as a given. K has already been compared extensively to other semantic notations [8,16], and this approach towards program analysis has already been compared to other program analysis approaches [8,11]; due to space concerns, we do not repeat these comparison here.

6 Summary and Future Work

In this paper we have presented a library-based approach to integrating Rascal with analysis tools written using K and running in Maude. This approach allows standard K specifications to be written with only slight modifications to account for the use of source locations and the format of the output. In Rascal, most of the code needed has been encapsulated into a reusable library, requiring only the Maude-ifier and a small amount of "glue" code to be written. The RLSRunner library, including the ScriptExec and ResourceMarkers code, consists of[1] 156 lines of Java code, 114 lines of Rascal code, and 9 lines of Maude code. For SILF, the Maude-ifier is 244 lines of Rascal code – essentially 2 lines per language construct (1 for the match, 1 for the generated string), plus several lines to handle lists. The glue code is 42 lines of Rascal in a single module, plus two lines to add the required menu items to run the analyses. 12 equations were added to the SILF specification to handle the source locations, totaling 25 lines. The definition of source locations is reusable in other K analysis specifications, while all added equations are reusable across other SILF analyses (or analyses in other languages with the same abstract language constructs). As a point of comparison, the total size of the SILF specification, including both analyses used here, is 4428 lines.

In the future we would like to expand the RLSRunner to provide for more execution options, including those related to Maude's search and model checking features (useful for concurrent languages), with appropriate visualizations of the results. We would also like to investigate the automatic generation of the Maude-ifier, and potentially the Maude syntax operators, from the Rascal and Maude language specifications. Finally, once the Rascal C grammar is complete, we plan to use RLSRunner to integrate the generated C environment with CPF [9], an existing analysis framework for C defined using K.

[1] These are just counts of the total number of lines in the file exclusive of blank lines and comments.

References

1. Frama-C, http://frama-c.cea.fr
2. Borras, P., Clement, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B., Pascual, V.: CENTAUR: the system. In: Proceedings of SDE 3, pp. 14–24. ACM Press (1988)
3. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. In: Proceedings of FMICS 2003. ENTCS, vol. 80, pp. 75–91 (2003)
4. Charles, P., Fuhrer, R.M., Sutton Jr., S.M.: IMP: A Meta-Tooling Platform for Creating Language-Specific IDEs in Eclipse. In: Proceedings of ASE 2007, pp. 485–488. ACM Press, New York (2007)
5. Charles, P., Fuhrer, R.M., Sutton Jr., S.M., Duesterwald, E., Vinju, J.J.: Accelerating the Creation of Customized, Language-Specific IDEs in Eclipse. In: Proceedings of OOPSLA 2009, pp. 191–206. ACM (2009)
6. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L. (eds.): All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
7. Heering, J., Klint, P.: Rewriting-based Languages and Systems. Cambridge Tracts in Theoretical Computer Science, vol. 55, pp. 776–789. Cambridge University Press (2003)
8. Hills, M.: A Modular Rewriting Approach to Language Design, Evolution and Analysis. PhD thesis, University of Illinois at Urbana-Champaign (2009)
9. Hills, M., Chen, F., Roşu, G.: Pluggable Policies for C. Technical Report UIUCDCS-R-2008-2931, Department of Computer Science, University of Illinois at Urbana-Champaign (2008)
10. Hills, M., Şerbănuţă, T.F., Roşu, G.: A Rewrite Framework for Language Definitions and for Generation of Efficient Interpreters. In: Proceedings of WRLA 2006. ENTCS, vol. 176, pp. 215–231. Elsevier (2007)
11. Hills, M., Roşu, G.: A Rewriting Logic Semantics Approach To Modular Program Analysis. In: Proceedings of RTA 2010. Leibniz International Proceedings in Informatics, vol. 6, pp. 151–160. Schloss Dagstuhl - Leibniz Center of Informatics (2010)
12. Klint, P., van der Storm, T., Vinju, J.: RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In: Proceedings of SCAM 2009, pp. 168–177. IEEE Computer Society, Los Alamitos (2009)
13. 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 III. LNCS, vol. 6491, pp. 222–289. Springer, Heidelberg (2011)
14. Meseguer, J., Roşu, G.: The rewriting logic semantics project. In: Proceedings of SOS 2005. ENTCS, vol. 156, pp. 27–56. Elsevier (2006)
15. Meseguer, J., Rosu, G.: The rewriting logic semantics project. Theoretical Computer Science 373(3), 213–237 (2007)
16. Roşu, G., Şerbănuţă, T.F.: An Overview of the K Semantic Framework. Journal of Logic and Algebraic Programming 79(6), 397–434 (2010)
17. van den Brand, M., Iversen, J., Mosses, P.D.: An Action Environment. Science of Computer Programming 61(3), 245–264 (2006)
18. van den Brand, M., Moreau, P.-E., Vinju, J.J.: Environments for Term Rewriting Engines for Free! In: Nieuwenhuis, R. (ed.) RTA 2003. LNCS, vol. 2706, pp. 424–435. Springer, Heidelberg (2003)
19. van den Brand, M., van Deursen, A., Heering, J., de Jong, H.A., de Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P.A., Scheerder, J., Vinju, J.J., Visser, E., Visser, J.: The ASF+SDF Meta-Environment: A Component-Based Language Development Environment. In: Wilhelm, R. (ed.) CC 2001. LNCS, vol. 2027, pp. 365–370. Springer, Heidelberg (2001)
20. van Deursen, A., Heering, J., Klint, P. (eds.): Language Prototyping: An Algebraic Specification Approach. AMAST Series in Computing, vol. 5. World Scientific (1996)

Metacompiling OWL Ontologies

Anders Nilsson[1] and Görel Hedin[2]

[1] Department of Automatic Control
Lund University, Sweden
[2] Department of Computer Science
Lund University, Sweden

Abstract. Ontologies, formal knowledge representation, and reasoning are technologies that have begun to gain substantial interest in recent years. We present a high-level declarative approach to writing application programs for specific ontologies, based on viewing the ontology as a domain-specific language.

Our approach is based on declarative meta-compilation techniques. We have implemented a tool using this approach that allows typed front-ends to be generated for specific ontologies, and to which the desired functionality can be added as separate aspects. Our tool makes use of the JastAdd meta-compilation system which is based on reference attribute grammars. We describe the architecture of our tool and evaluate the approach on applications in industrial robotics.

1 Introduction

The *semantic web* [23] aims at formalizing large portions of knowledge in a form which enhances interoperability of usually distributed systems, and which introduces provisions for a common understanding of basic terms. The term *ontology* is normally used in this context to denote a logical formalization of a particular domain of knowledge, stored in a commonly understood format and accessible via the world wide web or a similar mechanism.

There are already many tools for handling ontologies in different formats. Ontology editors for the well known ontology notation OWL [15,16], for example Protégé [17] and OntoStudio [5], typically store the information in a knowledge database as RDF triplets (subject, predicate, and object).

Access and manipulation of such knowledge, can be done either at the *generic* level, i.e., in terms of the triplets, or at a *domain-specific level*, where the ontology is used for interpreting the knowledge as a domain-specific programming model. Examples of tools working at the generic level include semantic reasoners, such as FaCT++ [21] or Pellet [11], which can infer new facts from the knowledge base, and the standard query language SPARQL [19], which allows the knowledge base to be queried. The knowledge base can also be directly accessed programmatically, using a generic API, like the Jena Owl API [8].

While the generic level is appropriate for general reasoning over arbitrary ontologies, the domain-specific level is often more appropriate for applications tied to a specific ontology. For example, a hierarchical structure is easy and natural to represent as a tree at the domain-specific level, but needs to be represented as separate triplets of knowledge at the generic level, making generic information retrieval cumbersome and error-prone: Instead of simply traversing a tree structure, each level of nodes must be retrieved using new queries and the hierarchical structure must be maintained outside the knowledge database in the application logic.

There are many tools that can be used to support writing applications at the domain-specific level. These tools typically represent the knowledge as an object-oriented model, introducing classes for the concepts in the ontology, and interpreting predicates to model class and object relationships like inheritance and part-of relations. Typically, this domain-specific model is accessible through an API in an object-oriented programming language, like Java, and which is generated from the ontology schema. Examples of such tools include RDFReactor [22] and Owl2Java [24]. A related approach is that of providing mappings between RDF and object-oriented models, such as EMF, e.g., [7].

However, a plain generated API has its limitations. First, the application programmer might like to enrich the generated semantic model with application-specific computations, for example in the form of additional fields and methods. Second, such application code should be separated from the generated API: we do not want the application programmer to edit generated code. Third, for computing properties of the semantic model, it can be advantageous to use high-level declarative programming. Fourth, the ontology might change, and it is desirable that the application code can be reasonably robust to such changes.

In this paper, we provide a solution that supports these requirements. We note that the problem of writing the application program is similar to writing a compiler: we need to parse information, analyze it, and generate some kind of output as a result. By using an object-oriented language, we can map the triplets for a specific ontology dialect to a typed object-oriented abstract syntax tree that is easy to perform computations on. For example, a subclass triplet, like (PincerGripper, subclassof, Gripper), would be mapped to a subclass relation between the corresponding classes in the object-oriented language. And a composition restriction, like (Gripper, has, OpenSkill), would be mapped to a parent-child relation in the abstract syntax tree.

Implementing the classes for such an abstract syntax tree by hand would be awkward, however, since they will then be sensitive to future changes to the description specification. Even small changes to the structure could imply a lot of work to adjust the compiler to the changes.

As is not uncommon, such problems become easier to solve by moving up to the next abstraction level. By implementing a meta-compiler, a compiler for OWL that, as output, generates a compiler for the description language specified in OWL, the abstraction level is raised. Instead of having to handle the dependencies between description language and tools manually, there is now one single specification for both description language and tool generation. The fact that these description languages are XML-based helps in that the parsing syntax is given beforehand.

We have implemented such a meta-compiler for OWL, called *JastOwl*. JastOwl is implemented using the JastAdd meta-compilation system [3] which supports high-level declarative computations on the abstract syntax tree by means of reference attribute grammars [6], and aspect-oriented modularization using inter-type declarations [10].

The rest of this paper is structured as follows. In section 2 we describe the architecture of JastOwl. Section 3 gives an example application of using JastOwl in the area of industrial robotics, and section 4 evaluates the approach. Related work is discussed in section 5, and section 6 concludes the paper.

2 JastOwl, a Meta-compiler for OWL

Figure 1 shows the use of the JastOwl meta-compiler. Given an OWL ontology and hand-written application aspects, a dedicated compiler is generated that can parse an OWL knowledge database following the constraints defined by the ontology, and process that information according to the application aspects. For example, the dedicated compiler could generate vendor-specific configuration files for a particular robot, or interface classes for particular sensors and actuators. The dedicated compiler could also be a more advanced interactive application, communicating with an active robot, for example, to employ a skill server database to reason about what tools to attach to a robot to accomplish a specific task. These are just a few examples of possible applications.

In the middle part of the figure we see the generation of the dedicated compiler: JastOwl parses the ontology and generates specifications for the dedicated compiler, namely a parsing grammar, an abstract grammar, and a JastAdd aspect that contains methods for serialization. The parsing grammar is run through a parser generator, JavaCC in our case [13], to produce the parser for the dedicated compiler. The abstract grammar and the generated JastAdd aspects are combined with hand-written application aspects and run through JastAdd to generate the remaining part of the dedicated compiler.

The JastOwl tool is itself generated using JavaCC and JastAdd, as shown in the top part of the figure. The architecture is general, and we could use the same architecture to generate similar tools for other ontology notations than OWL, and for other file formats (there exists a sister tool for XML).

The JastOwl tool analyzes the ontology to find class declarations and restrictions on individuals of these classes in order to generate the JastAdd abstract grammar, see Fig. 2. The abstract grammar corresponds to an object-oriented API with a type hierarchy and traversal methods for abstract syntax trees following the grammar. The generated JastAdd aspect adds OWL/XML serialization methods to this API. The handwritten application aspects use the combined API to generate the desired robotics code for an input knowledge database. Examples of such generated code could be interface classes, communication protocol code, and skill server reasoning code.

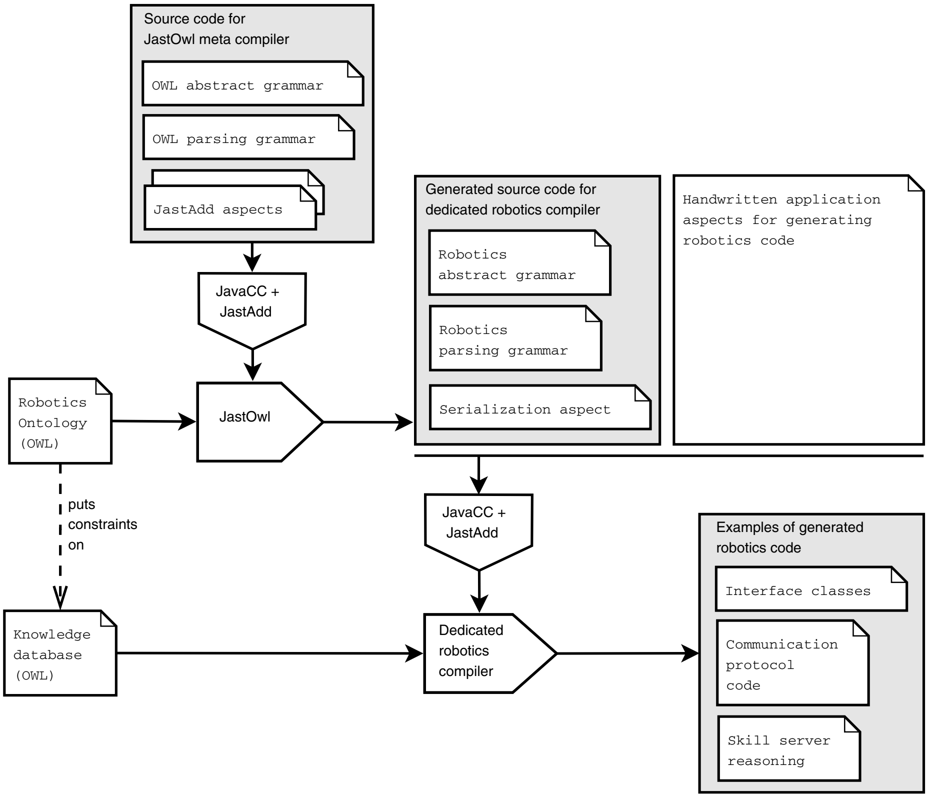

Source code for JastOwl meta compiler
OWL abstract grammar
OWL parsing grammar
JastAdd aspects
JavaCC + JastAdd
Robotics Ontology (OWL)
JastOwl
puts constraints on
Knowledge database (OWL)
Generated source code for dedicated robotics compiler
Robotics abstract grammar
Robotics parsing grammar
Serialization aspect
Handwritten application aspects for generating robotics code
JavaCC + JastAdd
Dedicated robotics compiler
Examples of generated robotics code
Interface classes
Communication protocol code
Skill server reasoning

Fig. 1. The JastOwl meta compiler. JastOwl generates a dedicated compiler description (abstract grammar, parsing grammar, and serialization code) for a given ontology. This description can be extended with application-specific handwritten aspects to generate the dedicated compiler. JastOwl is itself generated using JastAdd and JavaCC.

2.1 Generation Details and Limitations

The conceptual differences between Description Logic (DL) and Object Oriented (OO) systems as well as different ways of bridging the gap have been described in several papers. Kalyanpur et.al. [9] maps OWL classes to Java interfaces and properties to untyped Java lists while for example RDFReactor [22] uses a more elaborate approach where the OWL class hierarchy is being flattened to fit the Java single inheritance model. RDFReactor also handles OWL properties in a typed way in the generated Java code.

The development of JastOWL, on the other hand, has so far not been aimed towards a complete representation of DL in OO or a front-end to existing reasoners. Instead, the original idea was to implement a pragmatic toolkit in order to make it easier to write software that extracts knowledge from an ontological knowledge source and makes something out of it. For example, to generate access code or to generate communication protocol code.

The JastOWL translation of OWL concepts is similar to how RDFReactor does it, but with some limitations: The current version directly translates OWL classes into Java classes limiting us to ontologies where multiple inheritance is not used. OWL properties are handled similarly. For each property, child nodes will be generated for the corresponding domain class in the JastAdd abstract grammar, see Fig. 2. Multiple range properties are not yet supported.

It can be noted that the ontology and knowledge database are often stored in the same OWL file. Both the meta-compiler and the generated compiler will then operate on the same OWL file, but with very different goals. The meta-compiler looks for declarations of classes and restrictions, while the generated compiler is mainly interested in the instances of the aforementioned classes and restrictions.

To evaluate the approach, several prototype applications have been implemented, primarily in the area of industrial robotics.

3 SIARAS Skillserver Example

The example described here was developed as a part of the EU-project SIARAS *Skill-Based Inspection and Assembly for Reconfigurable Automation Systems* (FP6 - 017146) `http://www.siaras.org`. The main goal of the SIARAS project was to facilitate simple dynamic reconfiguration of complex production processes, by introducing the concepts of skill-based manufacturing and structured knowledge.

3.1 Ontology Structure

At the top level, see top left of Fig. 3, the ontology is split into six categories:

ObjectBase. Every physical object can be modeled as a simple *Part*, or as an *Assembly* consisting of parts or other assemblies.

Operation. The vocabulary needed for talking about operations[1] that are performed by a device.

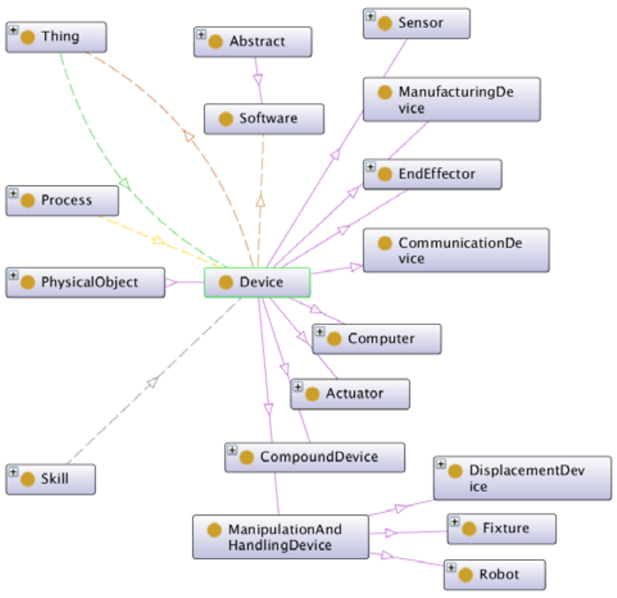
PhysicalObject. A work cell consists of *PhysicalObject*s. Some objects, *Devices*, are active and have skills, while other, *Workpieces*, are passive and are being manipulated by the devices.

Property. The *Property* hierarchy enumerates those properties of devices and skills which are interesting for the skill server to reason about.

Skill. A *Skill* represents an action that might be performed (by a device) in the context of a production process.

Task. The definition of a *Task* concept. It is not yet being used, but serves as a placeholder for possible future extension.

[1] An *operation* has been defined earlier as an instantiated skill. It is the basic element of task representation.



```
Start ::= Element*;
abstract Thing : ComplexElement ::=;
abstract Element;
ComplexElement : Element ::=  OwlIdentifier Attribute* Element*;
ValueElement : ComplexElement;
RdfDeclaration : ComplexElement;
abstract SimpleElement : Element ::= &lt;LITERAL&gt;;
Attribute ::= Value;
Value ::= &lt;STRING_LITERAL&gt;;
OwlIdentifier ::= &lt;IDENTIFIER&gt;;

PhysicalObject : Thing ::= hasProperty:Thing*;
Device : PhysicalObject ::= skill:Thing* subDevice:Thing* software:Thing*;
Abstract : Thing ::=;
Software : Abstract ::=;
Skill : Thing ::= hasProperty:Thing* isSkillOf:Thing*;
EndEffector : Device ::=;
Actuator : Device ::=;
CompoundDevice : Device ::=;
Sensor : Device ::=;
ManufacturingDevice : Device ::=;
CommunicationDevice : Device ::=;
Computer : Device ::=;
ManipulationAndHandlingDevice : Device ::=;
DisplacementDevice : ManipulationAndHandlingDevice ::=;
Fixture : ManipulationAndHandlingDevice ::=;
Robot : ManipulationAndHandlingDevice ::=;
```

Fig. 2. An OWL ontology and the corresponding generated abstract grammar. Solid edges indicate an *is superclass of* relation, and dashed edges an *is part of* relation.

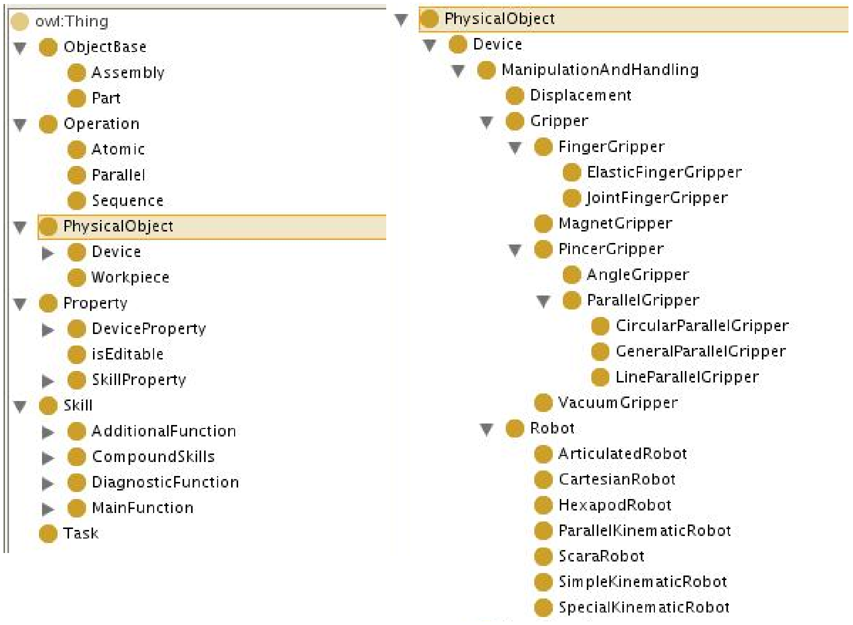


Fig. 3. Parts of the SIARAS robotics ontology

Most devices are not useful in isolation in a manufacturing cell, but must be combined with other devices to make a meaningful *compound device*. For example, consider a possible set of devices needed for an industrial robot to perform drilling in workpieces: robot controller, I/O board, robot arm, drilling machine, drill bit. None of these devices is by itself capable of drilling a hole at a specified location; a drilling machine can not position itself at the correct position, nor can it drill a hole without a drill bit attached to it. Only by connecting[2] together all the devices mentioned above may the resulting *compound device* perform a drilling operation.

Since the skill server is supposed to generate configurations for a robot cell, it must also be able to reason about how, and when, devices are connected to each other. We have therefore introduced a device relationship in the ontology, $hasSubDevice \leftrightarrow isSubDeviceOf$, in order to model compound devices. A difference compared to other relations in the ontology is that it is dynamic instead of static. Device instances in a device library will not typically be statically connected to any other device instance. Instead, a task description where a specific device instance is used, must also specify how it is connected to other devices listed in the task description. An example on specifying device relations is shown in Fig. 4.

[2] *Connect* should here not be taken literally but in a logical sense: controls/is controlled by.

```
controller_1: ABB_IRC5
ioboard_1: dig328
robot_1: ABB_IRB-140
clamp_1: AngleGripper
drillmachine_1: Bosch_GBM_10_RE
drillbit_1: DrillBit_HSS_8mm

SubDevice: controller_1,ioboard_1
SubDevice: controller_1,robot_1
SubDevice: drillmachine_1,drillbit_1
SubDevice: robot_1,drillmachine_1
```

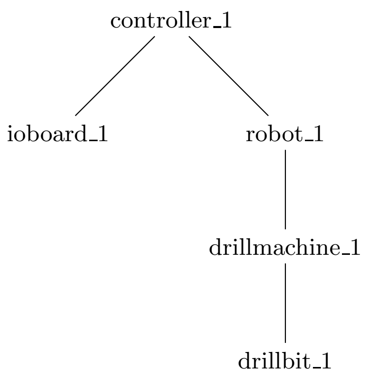


Fig. 4. Device specification from a task description on the left. Corresponding device tree on the right

We should also keep in mind that device relations may change during execution of a task description, for example by using a tool exchanger. Revisiting the example in Fig. 4 using a tool changer, we get a changing device tree such as one shown in Fig. 5. In the beginning of the task description, there is no device attached to the robot arm (if we do not consider the tool exchanger itself) — the middle tree in the figure. When the robot attaches a drilling machine, the device tree transforms to the left one. Finally replacing the drilling machine with a gripper results in the rightmost version of the device tree.

Yet another aspect of combining devices in compound ones is computing their properties out of the properties of their elements. In come cases this operation is obvious: e.g., a gripper can hold an object, thus a robot equipped with a gripper can also hold an object (simple inheritance). However, the allowed payload for such a compound device will not be inherited, but rather computed in a particular way. For example:

$$\min(\text{payload}(\mathit{robot}) - \text{weight}(\mathit{gripper}), \text{payload}(\mathit{gripper}))$$

There seems to be no obvious way to devise a generic inheritance mechanism for compounds; we currently assume that this will be specified by the user, although other possibilities are investigated.

3.2 Handling Knowledge

A lot of what the skill server is really about, is to transform information (knowledge) between different representations. First, the skill server needs to parse an ontology description, various local ontology extensions and a number of device descriptions from different device libraries, and build an internal representation of how the various parts of a manufacturing cell (devices, other physical objects, software, etc.) are interconnected, which is suitable for performing reasoning and feasibility analysis. In the other end of the skill server pipeline, it needs to be

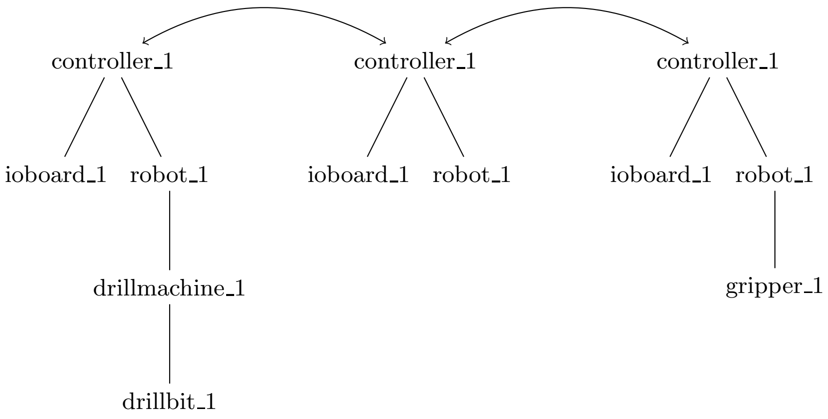


Fig. 5. Changing device tree when using a tool exchanger with the robot

able to generate configurations for the industrial robot cell. This is actually, at some level of abstraction, quite similar to what is done by any compiler for a programming language.

But, unlike a traditional compiler, the skill server is also used as a knowledge manager, keeping an abstract knowledge representation of the cell. As the manufacturing process executes. the cell state changes. For example, state changes occur when tools are replaced using tool changers or when work pieces are joined (glued, welded, screwed, etc.) together to finally form a product. As the cell state changes, the skill server internal representation of the device configurations must also change. As they are parts of a tree structure, we can simply model manufacturing cell state changes as moving branches of the syntax tree from one position to another.

Data inferred from the knowledge base is described as attributes of syntax tree nodes, declaratively defined by equations in a JastAdd application aspect. The declarative definition allows the information to be automatically updated whenever the syntax tree is changed. Currently, this is done simply by flushing all cached attribute values. New values are then computed on demand as needed. This works well as long as the syntax tree is not very large, and has not been a practical problem for our applications so far.

As an example of the use of JastAdd application aspects, consider the scenario shown in Fig. 5 where a robot is supposed to switch tools from a drilling machine to a gripper in order to be able to fulfill the operations mandated by the robot cell task. The skill server will then first check which ones of the available drilling machines, if any, could be used to perform the upcoming operations. Restrictions are given, for example, by the width and depth of the hole to drill, and by the workpiece material.

As a simplified version of this problem, consider the following grammar.

```
Start ::= Element*;
abstract Element;
Skill : Element ::= <Id> Property*;
Grasp : Skill;
Drill : Skill;
Device : Element ::= <Id> SkillUse*;
SkillUse ::= <Id>;
Property ::= <Value>;
```

To find the devices that can grasp, we need to look in each device, find which skills the device has by matching the `SkillUse` nodes to the appropriate `Skill` nodes, and finding out if one of those `Skills` is a `Grasp` object. This is accomplished by the following attributes and equations.

```
coll Set<Device> Start.devicesWithGrasp()
  [new HashSet<Device>()] with add;

Device contributes this
when canGrasp()
to Start.devicesWithGrasp() for root();

syn boolean Device.canGrasp() {
  for (SkillUse u : getSkillUses()) {
    if (u.decl() != null && u.decl().canGrasp()) return true;
  }
  return false;
}

syn Element SkillUse.decl() = lookup(getId());
inh Element SkillUse.lookup(String id);
eq  Start.getElement(int index).lookup(String id) {
  for (Element e : getElements()) {
    if (e.matches(id)) return e;
  }
  return null;
}

syn boolean Element.canGrasp() = false;
eq  Grasp.canGrasp() = true;

syn boolean Element.matches(String id) = false;
eq  Skill.matches(String id) = id == getId();

inh Start Device.root();
eq  Start.getElement(int index).root() = this;
```

This works as follows. The root of the abstract syntax tree, i.e., the `Start` node, has an attribute `devicesWithGrasp` that is a list of the devices we are looking for. It is defined as a so called *collection* attribute to which so called *contributions* contribute elements. In this case each `Device` contributes itself to this collection if it can grasp things.

To check if a `Device` can grasp things, it checks through its `SkillUses`. These are bound to `Skill` objects through a reference attribute `decl`, which is in turn defined through an inherited attribute `lookup`. The attributes `canGrasp`, `matches`, and `root`, are helper attributes.

Note that more code is needed to implement the actual reasoning, i.e., to match the set of restrictions imposed by the workpiece and operation to be carried out onto the set of properties of the retrieved devices. Based on some (given) optimization criteria the "best" device will be chosen.

4 Evaluation

The current JastOwl prototype, consisting of about 1500 lines of JastAdd code, can analyze a non-trivial OWL document and then generate a JastAdd abstract grammar, as well as a JavaCC parser description, for the description language as described by the OWL document. Regardless of which changes are done in the OWL-based specification, both the abstract and concrete grammars for the description language can be automatically generated.

In order to comprise a fully usable application, in the form of a dedicated compiler, application code, here in the form of JastAdd aspects, is needed. If the ontology changes, there is a possibility that the application code has to be changed as well. However, due to the use of high-level attribution mechanisms, the code is relatively insensitive towards changes in the syntax tree structure and to additions of new ontology classes or relations. In particular, equations for inherited attributes apply to complete subtrees and are therefore relatively insensitive to minor changes in the possible forms of the syntax tree. Likewise for contributions to collection attributes.

Recapitulating the four requirements from Section 1 we find that they are all satisfied. Aspect orientation in the form of static code weaving enables us to add desired functionality to the generated front-end in a modular fashion. We may re-generate the grammars and front-end code while not risking to accidentally delete any of the manually supplied code. Reference attribute grammars, as part of JastAdd, supplies a compact way to implement references to data stored in nodes in different parts of the tree, in effect transforming it to a directed graph.

Also performance-wise the proposed method of automatically generating a JastAdd based dedicated compiler front-end for ontological knowledge seems to be a good choice. The SIARAS skillserver could use any one of two different back-ends to access a library of device knowledge; either a backend based on Protégé with Pellet as reasoner, or a backend based on JastAdd, developed using the JastOwl meta compiler. The task of, for example, returning all grippers capable of lifting at least 0.5kg took around 3 seconds to execute using the JastOwl meta-compiler approach, and more than 20 seconds using the Protégé/Pellet back-end.

5 Related Work

The idea to take some kind of schema representation and generate a dedicated parser, model classes, and serialization code, is used in many other tools. In particular, there are many XML tools that employ this idea. Examples include JAXB [4] which is a part of the Java SE platform. Similar techniques also exist for OWL, for example [9], and is implemented by several tools, including Protege.

Whereas these tools generate high-level classes and APIs for particular schema or ontologies, JastOwl differs by basing the generation on a corresponding abstract grammar, and by supporting the modular addition of application-specific functionality to the generated classes. Furthermore, this added functionality can be specified at a high level, using declarative reference attribute grammars. Because the JastOwl tool is itself generated, it is also possible to add alternative serialization formats easily, that work for any ontology.

There are several other tools that provide high-level processing of schema-based formalisms by making use of grammarware, but that focus on term rewriting rather than analysis and computations on an AST [1,20].

An early approach to apply attribute grammars for schema-based notations was that of Psaila [18]. In this approach, it was suggested that the DTD schema for a class of XML documents was extended directly with attributes and equations to provide semantics to XML documents.

Cowan [2] presents an interesting way of connecting an OO Java model in the form of Javabeans with an RDF model using Java annotations and runtime reflection/introspection. However, no support for automatically generating Javabeans corresponding to a given RDF model has been found, and the developer is then left with the task of manually coding the needed Javabeans.

6 Conclusions

In this paper we have proposed how metacompilation based on reference attribute grammars can be used for developing tools for analyzing and manipulating ontological knowledge databases. By implementing a meta compiler, in this case a compiler parsing an ontology description in OWL, producing both abstract and concrete grammars for a dedicated compiler, we can get rid of the often tedious and error prone work of implementing such applications, as well as simplifying the maintenance of them as the ontology changes.

The application code can be modularized as aspects separate from the generated compiler source code, and can be programmed at a declarative high level using attributes. The separation of user submitted code from generated code result in fairly good robustness to changes in the ontology.

The JastOwl meta compiler has so far been used in several ontology related experiments and prototypes. Originally developed within the SIARAS project [12] (`http://www.siaras.org/`), JastOwl has also been used in industrial robotics ontology experiments within the european RoSta project [14] (`http://www.robot-standards.eu/`). Currently there is ongoing work within the ROSETTA project (`http://www.fp7rosetta.org/`) where we are investigating the possibilities of using ontologies in conjunction with self-describing communication protocols.

Experiences so far indicate that our method of using the JastOwl metacompiler with the JastAdd toolkit is an efficient way, both in lines of code as well as regarding performance, for analyzing and/or manipulating ontological knowledge.

References

1. Bravenboer, M.: Connecting XML processing and term rewriting with tree grammars. M. Sc. thesis. Utrecht University (November 2003)

2. Cowan, T.: Jenabean: Easily bind JavaBeans to RDF (April 2008), http://www.ibm.com/developerworks/java/library/j-jenabean/index.html
3. Ekman, T., Hedin, G.: The JastAdd System - modular extensible compiler construction. Science of Computer Programming 69, 14–26 (2007)
4. Fialli, J., Vajjhala, S.: The Java Architecture for XML Binding (JAXB). JSR Specification (2003)
5. Ontoprise GmBH: OntoStudio semantic modelling environment (2011), http://www.ontoprise.de/en/products/ontostudio/
6. Hedin, G.: Reference Attributed Grammars. Informatica (Slovenia) 24(3) (2000)
7. Hillairet, G., Bertrand, F., Lafaye, J.Y.: Bridging EMF applications and RDF data sources. In: Semantic Web Enabled Software Engineering (SWESE 2008), Karlsruhe (October 2008)
8. Jena: Jena – a semantic web framework for java (2009), http://jena.sourceforge.net/ (site accessed on November 5, 2009)
9. Kalyanpur, A., Pastor, D.J., Battle, S., Padget, J.A.: Automatic Mapping of OWL Ontologies into Java. In: Maurer, F., Ruhe, G. (eds.) SEKE, pp. 98–103 (2004)
10. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Lee, S.H. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
11. LLC, C.P.: Pellet: Owl2 reasoner for Java (2011), http://clarkparsia.com/pellet
12. Malec, J., Nilsson, A., Nilsson, K., Nowaczyk, S.: Knowledge-Based Reconfiguration of Automation Systems. In: International Conference on Automation Science and Engineering, CASE 2007, pp. 170–175. IEEE (2007)
13. Java-CC Parser Generator, metamata Inc., http://www.metamata.com
14. Nilsson, A., Muradore, R., Nilsson, K., Fiorini, P.: Ontology for robotics: A roadmap. In: International Conference on Advanced Robotics, ICAR 2009, pp. 1–6 (June 2009)
15. Web ontology language (2004), http://www.w3.org/2004/OWL/
16. Web ontology language, version 2 (2009), http://www.w3.org/TR/owl2-overview
17. The Protégé Ontology Editor and Knowledge Acquisition System (2009), http://protege.stanford.edu/
18. Psaila, G., Crespi-Reghizzi, S.: Adding semantics to XML. In: Workshop on Attribute Grammars, WAGA (1999)
19. SPARQL: SPARQL protocol and RDF query language (January 2008), http://www.w3.org/TR/rdf-sparql-query/
20. Stap, G.: XML document transformation processes using ASF+ SDF. M. Sc. thesis. University of Amsterdam (2007)
21. Tsarkov, D.: FaCT++ (2007), http://owl.man.ac.uk/factplusplus/
22. Völkel, M.: RDFReactor – From Ontologies to Programatic Data Access. In: Proc. of the Jena User Conference 2006. HP Bristol (May 2006)
23. W3C: Semantic web (2001), http://www.w3.org/2001/sw
24. Zimmermann, M.: Knowledge-Based Design Patterns for Detailed Ship Structural Design. Ph.D. thesis, University of Rostock (May 2010)

Towards Combinators for Bidirectional Model Transformations in Scala

Arif Wider

Humboldt-Universität zu Berlin
Unter den Linden 6, D-10099 Berlin, Germany
wider@informatik.hu-berlin.de

Abstract. In *model-driven engineering* (MDE), often models that conform to different metamodels have to be synchronized. Manually implemented model synchronizations that are not simple bijections are hard to maintain and to reason about. Special languages for expressing *bidirectional transformations* can help in this respect, but existing non-bijective languages are often hard to integrate with other MDE technologies or depend on up-to-date tool support. We embed *lenses* – a promising term-rewriting-based approach to bidirectional transformations – into the *Scala* programming language and use lenses for model synchronization. We discuss how this allows for static type-safety and for seamless integration with existing Java-based MDE technologies.

1 Introduction

Model-driven engineering (MDE) advocates the use of *domain-specific languages* (DSLs) for describing a system. Using multiple DSLs to describe different aspects of a system is called *multi-view modeling* or *domain-specific multimodeling*. In a metamodel-based context, this means that the system description consists of a heterogeneous set of models that conform to different metamodels. If these DSLs overlap semantically, the consistency of the system description has to be ensured, i.e., these models have to be synchronized.

Existing metamodel-based technologies like Xtext[1] provide good support for defining a DSL and for creating a corresponding *domain-specific workbench*, i.e., an integrated toolset that makes using this DSL comfortable. However, those technologies do not support multimodeling, yet, i.e., they do not provide means to specify (non-bijective) relations between DSLs, so that models that are created using these DSLs are synchronized automatically.

These synchronizations can be implemented manually as pairs of unidirectional model transformations. However, with this approach, the consistency of the forward and the backward transformation has to be ensured and both transformations have to be maintained separately. In order to avoid this maintenance overhead and to make defining such synchronizations more concise, there are special languages for defining *bidirectional transformations*. These languages provide

[1] http://www.eclipse.org/Xtext

means to specify a consistency relation which defines both the forward and the backward transformation. With *QVT Relations*[2], there is a language for defining bidirectional model transformations declaratively. Although standardization was finished in 2008, QVT Relations does not seem to have gained widespread use in current MDE practice. Stevens [1] points out semantic issues in QVT (especially regarding non-bijective transformations) that could be a reason for the limited acceptance. Additionally, we suppose that the lack of stable and up-to-date tool support for QVT Relations is responsible for the situation.

In this paper, we present a lightweight approach to bidirectional model transformations: We embed *lenses* – a combinator-based approach to bidirectional term-transformations – into the *Scala* programming language and explore how lenses have to be adapted for model transformations. Because of Scala's interoperability with Java, this approach integrates well with technologies that are based on the *Eclipse Modeling Framework* (EMF) as lenses implemented in Scala can directly process the Java-instances that represent an EMF model at runtime. Furthermore, one can benefit from existing tools for Scala – e.g., for editing, debugging and type-checking – and does not depend on special tool support.

A strength of lenses is their compositional notion: complex transformations are composed out of small and well-understood transformations using a set of combinators, which allows for compositional reasoning. This is possible because of lenses' asymmetric setting that differentiates lenses from QVT Relations and similar approaches: One of the two structures that are synchronized has to be an abstraction of the other. Although this restriction limits applications of lenses in MDE, we believe that lenses are well suited for multimodeling when used in conjunction with a common synchronization model as a shared abstraction.

Lenses are presented in detail in the next section. In Sect. 3, we explore how lenses can be adapted for model transformations and present a data model for that. In Sect. 4, we show how lenses can be embedded into Scala and how Scala's type system can be used to ensure static type-safety. After discussing related work in Sect. 5, Sect. 6 concludes the paper and presents future work.

2 Lenses

Lenses, as introduced by Pierce et al. [2], are asymmetric bidirectional transformations, i.e., one of the two structures that are synchronized has to be an abstraction of the other. This asymmetric approach is inspired by the *view-update problem* known in the database community, where a database view – the abstraction – has to be updated when the database changes and vice versa.

Given a set C of concrete structures and a set A of abstract structures, a lens comprises three functions:

$$\begin{array}{lll} get: & C & \to A \\ put: & A \times C & \to C \\ create: & A & \to C \end{array}$$

[2] being part of the QVT standard (http://www.omg.org/spec/QVT/)

The forward transformation *get* derives an abstract structure from a given concrete structure. The backward transformation *put* takes an updated abstract structure and the original concrete structure to yield an updated concrete structure. If there is no original concrete structure, the alternative backward transformation *create* creates a concrete structure using default values instead.

Lenses specify *well-behaved* [2] bidirectional transformations, which means that every lens must obey the following *lens laws*:

$$get(put(a, c)) = a \qquad (\textsc{PutGet})$$

$$get(create(a)) = a \qquad (\textsc{CreateGet})$$

$$put(get(c), c) = c \qquad (\textsc{GetPut})$$

These laws formalize some behaviour one would generally expect from a bidirectional transformation: The updated (or initially created) concrete structure c fully reflects changes made in the abstract structure a (PutGet and CreateGet) and data in the concrete structure that is hidden by the abstraction is preserved (GetPut). Now, the strength of lenses is their compositional notion: A set of *primitive lenses* whose well-behavedness was manually proved is provided together with a set of *lens combinators* for which it is proved that the resulting composed lens is well-behaved if all of its sublenses are well-behaved. These lenses and combinators then can be used as a vocabulary for bidirectional transformations from which arbitrarily complex lenses can be composed without having to prove the lens laws again. For example, a common combinator is the sequential composition *comp* which takes two lenses l and k as arguments and puts them in a row:

```
$comp(l : lens, k : lens) : lens$ {
    $get(c) = k.get(l.get(c))$
    $put(a, c) = l.put(k.put(a, l.get(c)), c)$
    $create(a) = l.create(k.create(a))$
}
```

The *get* direction is straightforward: first l's *get* function is called and the result is used as input for k's *get* function. The *put* direction is slightly more complicated: first, the original concrete input has to be abstracted by l's *get* function to be a proper input for k's *put* function. Pierce et al. show that with a small set of primitive lenses and combinators rich lens-libraries can be constructed.

3 A Data Model for Lenses for Model Transformations

In this section, we explore how lenses can be pragmatically adapted, so that they can be used in an EMF-based context for defining bidirectional model transformations. Pragmatic in the sense, that we take the characteristic properties of models into account and still stay as close as possible to the original semantics to be able to reuse some of the composed tree lenses presented by Pierce et al. [2].

Therefore, we have to look on the different data models: Lenses have been defined for transforming *unordered edge-labeled trees*. Taking the original lens example [2], the two trees shown in Fig. 1 can be kept in sync by composing a parametrized *focus* lens with the *map* lens combinator to *map*(*focus*(`Phone`,{`URL` ↦ `http://`}). Here, *focus* is parametrized to extract the phone number and – if there is no original model – to restore the lost URL with the default value `http://`.

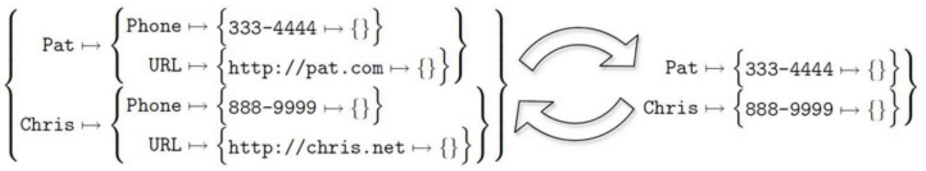


Fig. 1. A concrete tree and a derived abstract tree being kept in sync by a lens

In contrast, a model (at runtime) is a *graph of objects* that conforms to a *metamodel.* Those objects are instances of classes in the metamodel. Furthermore, a metamodel can contain constraints that restrict the set of valid models. An object is a triple of a unique identity by which it can be referenced, a state, and the implementing class defining valid operations on that object. The state of an object is defined by the values of a fixed number of fields. In a Java-based context, fields have a unique name and a static type. Fields containing multiple values can be expressed as a homogeneously typed collection, e.g., an indexed list or a key-value map. Thus, two models that are similar to the tree structures in Fig. 1 can be implemented in EMF as shown in the UML object diagram in Fig. 2.

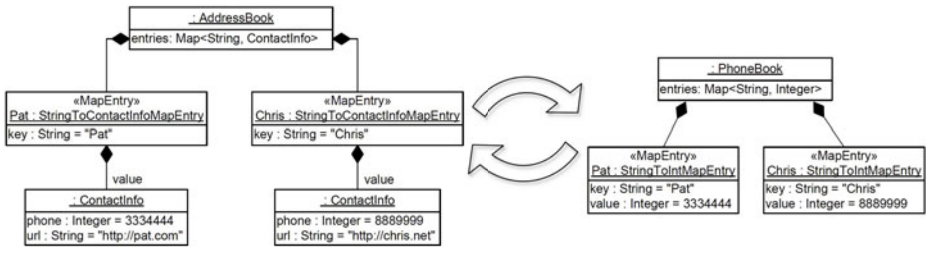
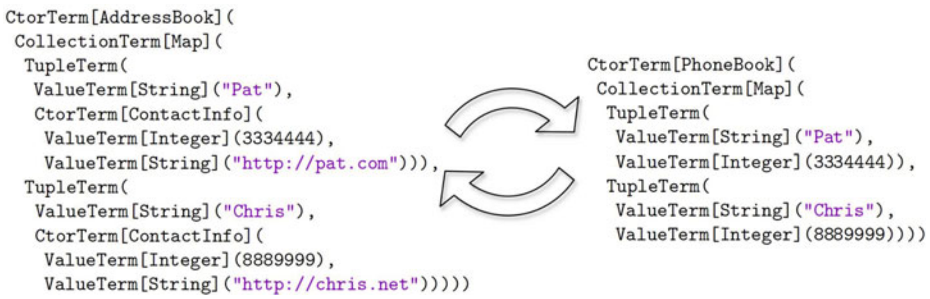

: AddressBook
entries: Map<String, ContactInfo>
«MapEntry»
Pat : StringToContactInfoMapEntry
key : String = "Pat"
«MapEntry»
Chris : StringToContactInfoMapEntry
key : String = "Chris"
value
: ContactInfo
phone : Integer = 3334444
url : String = "http://pat.com"
value
: ContactInfo
phone : Integer = 8889999
url : String = "http://chris.net"
: PhoneBook
entries: Map<String, Integer>
«MapEntry»
Pat : StringToIntMapEntry
key : String = "Pat"
value : Integer = 3334444
«MapEntry»
Chris : StringToIntMapEntry
key : String = "Chris"
value : Integer = 8889999

Fig. 2. A UML object diagram of an EMF implementation of the tree example in Fig. 1

Obviously, a fundamental difference is the fact that models in general are graphs. But if we look at MDE frameworks like EMF, it is characteristic that a spanning containment tree is enforced, i.e., models must have an explicitly marked root-object and objects can have at most one container. This constraint has been shown to be very useful for MDE tool implementations. If we adopt this constraint, we can describe a model as a tree of terms that have a type-annotation, a unique identity, and either a fixed number of subterms (the fields) or an arbitrary number of subterms of the same type (the contents of a collection). In the former case – we call it a *Constructor Term* (`CtorTerm`) – its arity and the order of its subterms is determined by its classtype, whereas in the latter case (`CollectionTerm`) it has an arbitrary arity. Terms that have no subterms either hold a single value-literal (`ValueTerm`) or the id of another term, thus, representing a non-containment reference (`RefTerm`). Finally, it is often helpful, to be able to express a tuple of terms (`TupleTerm`), that does not correspond to a classtype, e.g., for representing the key-value pairs in a map. The following grammar defines a notation for describing models in that way; type-annotations are surrounded by square brackets and, for now, we assume a unique identity to be implicitly carried by every term.

$$\begin{aligned} term ::= {} & CtorTerm[classtype](term_1, ..., term_n) \\ \mid {} & CollectionTerm[collectiontype](term_1, ..., term_n) \\ \mid {} & ValueTerm[valuetype](\textsc{value}) \\ \mid {} & RefTerm[classtype](\textsc{idref}) \\ \mid {} & TupleTerm(term_1, ..., term_n) \end{aligned}$$

This term notation has similarities with the *ATerm* format [3] and combines it with the data model of EMF. Fig. 3 shows the models from Fig. 2 being represented in our term notation.



```
CtorTerm[AddressBook](
 CollectionTerm[Map](
  TupleTerm(
   ValueTerm[String]("Pat"),
   CtorTerm[ContactInfo](
    ValueTerm[Integer](3334444),
    ValueTerm[String]("http://pat.com"))),
  TupleTerm(
   ValueTerm[String]("Chris"),
   CtorTerm[ContactInfo](
    ValueTerm[Integer](8889999),
    ValueTerm[String]("http://chris.net")))))
```

```
CtorTerm[PhoneBook](
 CollectionTerm[Map](
  TupleTerm(
   ValueTerm[String]("Pat"),
   ValueTerm[Integer](3334444)),
  TupleTerm(
   ValueTerm[String]("Chris"),
   ValueTerm[Integer](8889999))))
```

Fig. 3. The models from Fig. 2 represented in our type annotated term notation

Comparing this data model with the one of tree lenses, type-annotations and (implicit) object-ids were added and order of subterms now matters. This way, edge-labels are replaced by indices, but together with the type-annotation indices can be mapped to field names. Furthermore, we defined different types of terms, which, together with the type annotations, will later help us to express type constraints on the data that a lens can handle. On the one hand this data model allows us to implement most of the original tree lenses with similar semantics for model transformations and on the other hand allows for defining further lenses that are needed for applications in MDE, namely, lenses that handle non-containment references and make use of objects' identities.

4 Embedding Lenses in Scala

When composing lenses, it is desirable to have tool support that ensures that a composed lens will produce models that conform to the target metamodel and to the source metamodel, respectively. Therefore, we embed lenses into the Scala programming language and show how Scala's type system allows for static structural analysis and for expressing type constraints of specific lenses. This way, the corresponding error highlighting, syntax checks and code completion features can be provided by any Scala IDE plug-in and no further tooling is needed. Because of Scala's interoperability with Java, lenses implemented in Scala can directly process the Java-instances that represent an EMF model at runtime.

Statically Typed Lenses. In a first naive approach, the abstract type of a lens that synchronizes between concrete terms of type C and abstract terms of type A can be implemented as shown in the following code listing:

```
abstract class Lens[C <: Term, A <: Term] {
  def get(c: C): A
  def put(a: A, c: C): C
  def create(a: A): C           }
```

Both types C and A have to inherit (`<:`) from `Term` which is the root of the term type hierarchy presented in Sect. 3. Based on this lens type, we can define simple lenses and lens combinators like the sequential composition *Comp* (see Sect. 2) and we are able to express *Comp*'s type constraint that the abstract term of lens l has to be of the same type (`CA`) as the concrete term of lens k:

```
class Comp[C,CA,A](l: Lens[C,CA], k: Lens[CA,A]) extends Lens[C,A] {
  def get(c: C): A = k.get(l.get(c))
  def put(a: A, c: C): C = l.put(k.put(a, l.get(c)), c)
  def create(a: A): C = l.create(k.create(a))         }
```

Let the domain classes and the concrete model from the example in Sect. 3 be implemented like this (for brevity, we use Scala's *case class* syntax which allows for omitting the `new` keyword – we also could have defined the model using Java/EMF):

```
case class AddressBook(entries: Map[String, ContactInfo])
case class ContactInfo(phone: Int, url: String)
case class PhoneBook(entries: Map[String, Int])
val ab = AddressBook(Map("Pat" -> ContactInfo(3334444,"http://pat.com"),
                      "Chris" ->
                          ContactInfo(8889999,"http://chris.net"))
```

Now, the goal is to be able to parameterize and compose different pre-defined lenses (here, *Focus* and *Map*) and use the resulting composed lens to transform the model as shown in the following statically typed Scala code:

```
val ab2pb = Map(Focus(...)) // composing and parameterizing the lens
val pb: PhoneBook = ab2pb.get(ab) // derive abstract model as phonebook
pb.entries("Pat") = 3334321 // modify the contents of the phonebook
val abnew: AddressBook = ab2pb.put(pb, ab) // put the changes back
```

Converting Models to Typed Terms. In order to be able to implement such pre-defined lenses independently from the concrete domain classes, i.e., as lenses that work on term types, but on the other hand be able to use these lenses directly on domain objects as shown before, models have to be converted to terms. We use Scala's *implicit conversions* for transparently converting models to terms and vice versa. We want to preserve static type-safety throughout the whole transformation process, therefore, we have to keep track of the types of all of a term's subterms. This typing cannot be achieved only by annotating terms with a corresponding class type, because in the transformation process *intermediate term structures* can emerge that do not correspond to a class that is defined in the source metamodel or in the target metamodel. As Scala's type system (and other common type systems) only provide either a heterogeneously typed tuple construct with a fixed arity (e.g., `Tuple3[A,B,C]`) or a homogeneously typed collection (e.g., `List[A]`) we use *heterogeneously typed lists* (HLists) as introduced by Kiselyov et al. [4] as the underlying data structure. HLists are based on typed Cons-cells and can be implemented and used in Scala like this:

```
abstract class HList // base type with the following two subtypes:
case class HCons[H, T <: HList](head: H, tail: T) extends HList
case class HNil extends HList // type to express the end of a list
val hl: HCons[String, HCons[Int, HNil]] = HCons("str", HCons(42, HNil))
```

Some Scala implementations of HList – e.g., J. Nordenberg's [5] – define *typelist* types (TList) correspondingly and define the type alias `::[H,T]` for `TCons[H,T]`. Thus, using a TList as the type parameter of HList allows for concisely defining a list that contains objects of type A, B and C as `HList[A::B::C::TNil](a,b,c)`. A term type like `CtorTerm` that can have subterms of different types wraps an HList and has two type parameters: the corresponding class type C and TL, the typelist of its inner HList. Domain objects can now be converted back and forth implicitly. Therefore, pairs of conversions have to be provided for every class whose objects can be part of a model but can be generated from the corresponding metamodels.

```
class CtorTerm[C, TL <: TList](subterms: HList[TL])
// implicit conversions between ContactInfo and CtorTerm:
implicit def CI2Term(ci: ContactInfo):
  CtorTerm[ContactInfo, ValueTerm[Int] :: ValueTerm[String] :: TNil] = ..
implicit def Term2CI(t: CtorTerm[ContactInfo, ...]): ContactInfo = ...
```

Type-Parameterized Lenses. For a parameterized lens like *focus*, we need some type-level programming: Scala provides an alternative concept for type parameters, called *abstract type members* (accessible as `Type#TypeMember`). Like other abstract class members, abstract type members can be implemented by subclasses. As type members can have type parameters themselves, this can be used to realize *type functions* that are evaluated at compile-time. Together with recursively defined type-level number literals (`Nats`), this allows for defining type-safe methods of HList, e.g., a type-safe indexed accessor (`nth`):

```
abstract class Nat // a type representing natural numbers
class Succ[P <: Nat](...) extends Nat // recursively defined numbers
class _0 extends Nat // a type for expressing the bottom type
type _1 = Succ[_0] //type aliases for number literals; _2, _3, et cetera
abstract class TList{ type Nth[N <: Nat] } //TList's abstract type member
abstract class HList[TL <: TList] {
  def nth[N <: Nat](n: N): TL#Nth[N] // type-safe indexed accessor
```

With this framework of implicit conversions, term types and type-safe operations on HLists, we can define some parameterized lenses. In the following code listing, the *focus* lens is parameterized with the `_0` number literal to extract the phone number (0th member) of a ContactInfo object. The third parameter, a default object of type C, is only needed for the create function. As can be seen, the type parameters of `Focus` can be inferred by the compiler.

```
class Focus[C, TL <: TList, N <: Nat]
  (c: Class[C], n: N, dflt: C) extends Lens[CtorTerm[C,TL],TL#Nth[N]]{..}
// a parameterized instance of Focus to extract the phone number:
val focus = Focus(classOf[ContactInfo], _0, ContactInfo(42,"http://"))
// type inferred to: Lens[CtorTerm[ContactInfo,TCons[..]],ValueTerm[Int]]
val ciPat: ContactInfo = ab.entries("Pat") //here, the concrete structure
val phonePat: Int = focus.get(ciPat) // retrieving the abstract structure
val ciPatNew: ContactInfo = focus.put(3334321,ciPat) // put changes back
```

Generic Lenses. For some lenses the approach presented so far works well, but for every type that is to be transformed, a parameterized lens has to be instantiated. If, for instance, a lens is to be applied to one subterm and another lens is to be applied to all other subterms, different instances of the latter have to be provided for every type of subterm, which is not feasible. This applies already to very simple lenses like the identity lens *Id*. Using a common supertype (e.g., `Id extends Lens[Term,Term]`) does not help, because this way type information gets lost.

A more generic abstract lens type is needed, which we call `MetaLens`. The main difference is that the lens functions now have a type parameter themselve which is to be inferred from the passed function parameters. This way, the type C of the concrete term is not determined until – at compile-time – a lens' function is called. The type A of the abstract term is then determined by the type member `A[C]` which serves as a type function to express A in terms of C. Finally, a second abstract type member `Constraint` is provided to express constraints on C.

```
abstract class MetaLens {
  type Constraint <: Term // type member for defining constraints on C
  type A[C <: Constraint] <: Term // type function to derive A from C
  def get[C <: Constraint](c: C): A[C] // C is inferred and determines A
  def put[C <: Constraint](a: A[C], c: C): C // same with put
  def create[C <: Constraint](a: A[C]): C // inference via result type
}
```

The abstract `Term` type provides some of its properties as type members, e.g., its typelist `TL` and its concrete `TermType`. Therefore it is possible to define a lens that, for instance, only works on constructor terms by specifying `type Constraint = Term{ type TermType = CtorTerm[_,_] }`.

An example for a lens implementing this generic lens type is a generic version of the *focus* lens that does not have to be parametrized with a concrete type C:

```
class GenericFocus[N <: Nat] extends MetaLens {
  type Constraint = Term // no special constraint here
  type A[C <: Constraint] = C#TL#Nth[N] // implementing the type function
  def get[C <: Constraint](c: C): A[C] = c.subterms.nth[N]
  ...                       }
```

Ultimately, this gives us a rich framework for defining generic type-safe lenses and for expressing their type constraints. Still, these generic lenses can be used as concise as the lenses shown before because of Scala's rich type inference mechanisms. We implemented several of the original lenses with this generic lens type and we are confident that it serves as solid basis for a lens-library for MDE applications.

5 Related Work

Our approach to embed a term-rewriting- and combinator-based language into Scala was inspired by the work of Sloane, who embedded the term-rewriting language *Stratego* into Scala as part of the *kiama* project [6]. However, in kiama, Scala's type system is not leveraged to that extent as in our approach and terms are not typed. Admittedly, as Stratego is all about generic traversal, here, achieving static type-safety is much more involved. By embedding lenses into Scala, our approach to bidirectional model transformations is lightweight in the sense that it is easy to integrate with existing projects and tools. Furthermore, it is flexible in the sense that developers, who do not immediately see a way to solve their task using a special transformation language, can use Scala as a general-purpose language and can later gradually migrate to a bidirectional implementation in order to reduce the long-term maintenance overhead of unidirectional transformations.

Apart from symmetric or mainly bijective approaches like *Triple Graph Grammars* and QVT, respectively, other approaches that apply non-bijective asymmetric bidirectional transformations to heterogeneous model synchronization were presented by Xiong et al. [7] and Hidaka et al. [8]. These approaches are less lightweight but – in contrast to our term-based approach – are graph-based and, hence, are in general less limited regarding changes that affect non-containment references. In contrast to lenses' state-based nature, the approach of Xiong is update-based. This prevents seamless integration with existing technologies because updates have to be marked explicitly which results in a dirty metamodel. However, Diskin et al. argue that state-based lenses cannot decide whether an object was replaced by another or if its state was changed and, thus, show semantic issues with applying lenses to model transformations. Therefore, they introduce *delta-based lenses* that separate update-alignment from update-propagation [9]. This way, lenses can be used to synchronize graph-based models as long as a correct update-alignment is provided. Hidaka et al. combine concepts from lenses with the graph query language *UnQL* and presented promising results. However, compared to our embedding approach, their solution is not as tool-independent and easy to integrate, yet.

6 Conclusions and Future Work

We presented a lightweight approach to bidirectional model transformations by embedding lenses into Scala and showed how this allows for using them – with certain restrictions – in an MDE context. Furthermore, we showed that type-level programming techniques together with Scala's implicit conversions can be leveraged to ensure static type-safety. Now, our first goal is to provide most of the original lenses [2] with similar semantics but with static typing in Scala. However, to be applicable in MDE, the original framework of tree lenses has to be extended with lenses that handle non-containment references and account for objects' identities. Although we are confident, that some of those lenses can be defined using the original lens semantics, we plan to investigate if our approach can also be applied to delta-based lenses and how this affects integration with existing MDE technologies.

Beyond that, an issue with our current implementation is the missing support for *generic traversal*. Therefore, we explore how generic programming techniques from Haskell can improve our approach. Oliveira and Gibbons showed how some of these techniques can be implemented in Scala [10].

Acknowledgements We like to thank Terje Gjøsæter, Siamak Haschemi, Frank Kühnlenz, Martin Schmidt, Guido Wachsmuth, and the anonymous reviewers for comments on a preliminary version of this paper. This work is supported by the Deutsche Forschungsgemeinschaft, Graduiertenkolleg METRIK (GRK 1324).

References

1. Stevens, P.: Bidirectional Model Transformations in qvt: Semantic Issues and Open Questions. Software and Systems Modeling 9(1), 7–20 (2010)

2. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for Bi-Directional Tree Transformations: A Linguistic Approach to the View Update Problem. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, pp. 233–246. ACM (2005)
3. van den Brand, M., de Jong, H.A., Klint, P., Olivier, P.A.: Efficient Annotated Terms. Software – Practice and Experience 30(3), 259–291 (2000)
4. Kiselyov, O., Lämmel, R., Schupke, K.: Strongly Typed Heterogeneous Collections. In: Proceedings of the ACM SIGPLAN Workshop on Haskell, pp. 96–107. ACM (2004)
5. Nordenberg, J.: Type Lists and Heterogeneously Typed Arrays (2009), `http://jnordenberg.blogspot.com/2009/09/type-lists-and-heterogeneously-typed.html`
6. Sloane, A.M.: Experiences with Domain-Specific Language Embedding in Scala. In: Proceedings of 2nd Int'l Workshop on Domain-Specific Program Development (2008)
7. Xiong, Y., Song, H., Hu, Z., Takeichi, M.: Supporting Parallel Updates with Bidirectional Model Transformations. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 213–228. Springer, Heidelberg (2009)
8. Hidaka, S., Hu, Z., Kato, H., Nakano, K.: A Compositional Approach to Bidirectional Model Transformation. In: ICSE Companion, pp. 235–238 (2009)
9. Diskin, Z., Xiong, Y., Czarnecki, K.: From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case. Journal of Object Technology 10, 6:1–6:25 (2011)
10. Oliveira, B.C.D.S., Gibbons, J.: Scala for Generic Programmers: Comparing Haskell and Scala Support for Generic Programming. J. Funct. Program. 20, 303–352

Typed First-Class Communication Channels and Mobility for Concurrent Scripting Languages

Paweł T. Wojciechowski

Poznań University of Technology, Poland
Pawel.T.Wojciechowski@cs.put.poznan.pl

Abstract. In the 1990s, there was considerable interest in mobile computation: systems in which running computations (or mobile agents) could be moved from one machine to another. Much of this work was in terms of high-level programming languages and mobile process calculi. An example is Nomadic Pict—a prototype high-level programming language in which to express and verify overlay networks, for reliable communication between mobile agents. One can ask whether the language abstractions could be useful for scripting programming in modern distributed deployment platforms, such as many-core processors, grids, web servers and datacentres. In this paper, we demonstrate selected features of Nomadic Pict, and show the use of typed channels and agent mobility for programming in the grid. We demonstrate example design patterns that can be used for implementing safe message passing, test & send, system bootstrapping, and relocatable computation.

1 Introduction

In the 1990s, there was considerable interest in *mobile computation*: systems in which running computations could be moved from one machine to another. Much of this work was in terms of high-level programming languages and mobile process calculi, such as the π-calculus [11] and Mobile Ambients [5] (see e.g. [14,10,8,12,19,17] among others). Process calculi (also known as process algebras) were originally conceived for the formal study of concurrent and mobile communication systems. They provide a rigorous framework where complex systems can be accurately analyzed, including reasoning techniques to verify their essential properties. In parallel, various high-level programming languages have been designed based on the process calculi. Unfortunately a lot of this work still remains theoretical, with only a few language implementations available. Now, relocatable computation is a pervasive reality, though at the level of virtual machines rather than high-level languages. One can ask whether the semantic theory and language abstractions developed in these frameworks could be applied (or adapted) to *scripting languages* designed for distributed deployment platforms such as many-core processors, grids, web servers and datacentres?

One of the goals of scripting languages for distributed deployment platforms is to provide a lightweight but expressive set of programming constructs for connecting distributed chunks of computations (or whole applications, services, etc.)

and defining control flow. In such languages, some programming errors can be detected via *type-checking*, either statically or, more often, dynamically. Important modern scripting languages include Perl, Python, PHP, JavaScript, Ruby or extensions of Lisp. Many of these languages were originally developed for specialized domains, e.g. web services, but are increasingly being used more broadly. A shortcoming of most scripting languages is the lack of first-class support for concurrency. Concurrency is nowadays ubiquitous and no longer bound to a narrow high-performance computing domain. It is required for scalability and interacting with remote services. The newest proposals of scripting languages overcome these shortcomings. For example, Thorn [4] has support for concurrency based on message passing between lightweight, isolated processes. Clojure [6], an extension of Lisp, takes a different approach to concurrency and supports sharing changing state between threads in a synchronous and coordinated manner using Software Transactional Memory (STM). Typed message-passing is the sole means of communication between processes in the Singularity OS [7]. However, the implementation does not support cross-machine channels.

On the other hand, relocatable computation is not yet a frequently supported feature in scripting languages. In the virtualized environments, relocatable computation (or virtual machines) can make it easier to deploy applications and reduce the impact of partial system failures by moving applications from a misbehaving network node to a non-faulty node. Thus, the design of novel scripting languages that support mobile computations could improve system robustness. Are tasks typical of scripting programming that would best be expressed at the level of mobile process calculi? In this paper, we describe a series of programming examples (or patterns) that answer positively. We target the grid platform but the patterns presented in the paper are general, and so can be applied to other deployment platforms as well. In [2], the authors describe type-safe programming mechanisms for combining and managing enterprise services, in the setting of farms of virtual machines. It would be interesting future work to extend our language to control VMs, using the service combinators described in [2].

In the late 1990s, we developed *Nomadic Pict* [18,19,17,13][1]—a mobile agent distributed programming language. The low-level language extends the compiler and run-time system of Pict [14], a concurrent language based on the π-calculus, to support our primitives for agent creation, migration, and location-dependent communication. High-level languages, with particular infrastructures for location-independent communication, can then be obtained by applying user supplied translations into the low-level language. An experimental implementation of Nomadic Pict and further details are available from [13]. The goal of this paper is to show how Nomadic Pict's abstractions, such as typed first-class communication channels and agent mobility could be used to safely express typical tasks of a scripting language for distributed deployment platforms. We use concrete examples of executable programs in Nomadic Pict to express: safe message-passing communication, test & send synchronization, system bootstrapping, and relocatable computation in a networked system. The examples are toy

[1] Nomadic Pict and its theory is joint work with Peter Sewell and Asis Unyapoth.

applications that serve only to illustrate the concepts, but we hope that abstractions such as typed first-class channels and relocatable computation will pave the way into future industrial strength scripting languages.

Some of the Nomadic Pict abstractions have been encoded in libraries of general-purpose functional programming languages. For example, an experimental language Acute [15] has a distributed message-passing library that is an implementation of the Nomadic Pict constructs for migration of mobile computations and communication between them. Acute extends an ML-like core language to support distributed development, deployment, and execution, allowing type-safe interaction between separately built programs. Some of these ideas were further developed and put into practice in HashCaml language [3], an extension of the OCaml bytecode compiler with support for type-safe marshalling and related naming features.

This paper is *not* a research paper on Nomadic Pict but a paper to accompany a language demonstration. The readers interested in the design and implementation of our language, the Nomadic π-calculi, formal reasoning and proofs are referred to [18].

2 Language Demonstration

2.1 Typed Channels for Safe Communication

Consider a program in which several parallel processes communicate by means of messages. One of the frequent programming errors in message-passing programs is that the type of values marshaled for communication does not match the type of values expected on the receiver's side. Below is a small program in Nomadic Pict to illustrate this case.

```
new x : ^Int        {- Communication channel creation -}

run (
  x! 10
| x? msg= printi! msg       {- Message-passing communication -}
)
```

The above program creates a communication channel using a keyword `new`; the channel is named `x`, and has a type `^Int` of channels carrying values of type `Int`. Then, the program executes (using `run`) two parallel processes. The first process outputs a message (a value `10`) on channel `x`, and terminates. In parallel, denoted with `|` (bar), the second process waits for an input on channel `x`. After the message has been received, it is substituted for the formal parameter `msg` and the process reduces to `printi!10`, which prints out `10`.

If the program would be modified, so that the first process outputs a message of a different type, e.g. a record of two integers `[10 10]`, or the second process expects a value of a different type than `Int`, then the program would not be correct, and the Nomadic Pict compiler would generate an error. Later in the paper, we show programs that use typed channels for network communication. In such programs, the same typing principle is used, allowing type-mismatch errors to be detected at compile time. This simple typing principle could be further extended, e.g. to support session types [9].

Processes communicate using message passing instead of shared variables, which removes the need for locks. Channel names are *first-class* values, i.e. they can be created at runtime and passed as arguments or results of function calls. Contrary to the message-passing languages that follow the Actor model (such as Erlang [1]), channel names can be passed along other channels in the style of the π-calculus. For instance, in the following program a channel name `x` will be communicated on another channel of type `^^Int` to some other process executed in parallel, which can use `x` for communication. First-class communication channels can be very useful in grid programming, e.g. to dynamically reconfigure the logical network topology of a grid in response to some events.

```
new x : ^Int       {- Creation of typed first-class channels -}
new y : ^^Int

run (
  x! 10 | y! x | y? p= p? msg= printi! msg    {- Communication -}
)
```

The program above creates two communication channels, named `x` and `y`, using a keyword `new`. The channels are typed. The former channel has type `^Int` of channels that can only carry values of type `Int`, while the latter channel has type `^^Int` (understood as `^(^Int)`) of channels that can carry names of channels of the former type. In the main part of the program we execute (using `run`) three parallel processes. The first and second process output their messages, respectively on channel `x` and `y` and terminate, while the third process waits for an input on channel `y`. The output and input on channel `y` can synchronize, reducing the third process to an input process `x?arg= printi!arg`, which again synchronizes with an output on `x`. Finally, the program prints out the message received on channel `x`, i.e. `10`.

The construct `<chan>?<pattern>=` is only used for one input. If we would require the input process to be ready to accept new messages, then we should use a replicated input construct, as in the program below.

```
new x : ^Int

run (
  x!1 | x!2 | x!3          {- Three parallel output processes -}
| x?* arg = printi! arg    {- A replicated input process (server) -}
)
```

In the above program, three concurrent processes output integer numbers, which are received by another process (a server) that prints them all out. The order of message delivery is unspecified, since the parallel processes in our program are not synchronized. In case of remote communication, we may choose to replace the `x?* arg = ...` construct by a timed input as below.

```
wait
  x?* arg = printi! arg
timeout
  t -> print! "Timeout!"
```

If no message is received on channel `x` after `t` seconds (roughly), then an exception is raised and handled in the `timeout` clause.

2.2 Agents and Test & Send

A *distributed computation* is one whose portions can be executed in different sites (or grid nodes, or processors) interconnected via a network. In Nomadic Pict a distributed computation consists of *agents* located on sites, where a *site* is an instance of the Nomadic Pict runtime system. Internally, agents may consist of many concurrent processes that can communicate using channels. The channels are distinct, in that outputs and inputs can only interact if they are in the same agent. This provides a limited form of *dynamic binding*, with the semantics of a channel name (i.e., the set of partners that a communication on that channel might synchronise with) dependent on the agent in which it is used.

In order to test if an agent is present on a local site, we can use a *test & send* synchronization construct `iflocal`.

```
new chan : ^String

agent a =
 iflocal <b> chan! "MESSAGE"
   then print! "b is on this site."
   else print! "b is not here."

and b =
  chan ?* msg = print! msg
```

The above program creates two agents `a` and `b`. Execution of the conditional `iflocal <b>chan!"MESSAGE"` by agent `a` checks if agent `b` is on agent `a`'s current site. If so, then it delivers a message `"MESSAGE"` to channel `chan` inside agent `b` as part of the same atomic action, and continues with the 'then' clause. Otherwise, it continues with the 'else' clause. The `iflocal` construct may simplify programming of failure detectors in the grid.

2.3 Distributed Bootstrapping

Grid computations are to be executed on a large number of machines. Therefore, parallel portions of a distributed computation must be spawned on machines automatically, with any communication links properly established. If the programming environment does not offer any support of this sort, the programmer has to implement bootstrapping of a grid system. Below we demonstrate how this can be done in Nomadic Pict.

Execution of `migrate to n` migrates the whole agent including any communication channels to a site `n`. After migration, the agent's execution commences from the point in which it has stopped before migration. Migration transparency greatly simplifies programming, for the cost of a more complex virtual machine.

```
val n = ''sirius.cs.put.pl'':5000
new c : ^String

agent a =
(
  migrate to n     {- Migrate agent a to sirius and continue -}
  c ?* msg = print! msg
)

and b =
  <a @ n> c! "Hello!"
```

In the above program, two agents `a` and `b` are created. After creation the former agent migrates to a site `n`, identified by a pair of an IP address and a port number and waits for a message on channel `c`. In parallel, agent `b` outputs a string message `"Hello!"` to agent `a`, and terminates. Agent `a` is expected to be on site `n`. If the agent will not be there, when the message has arrived, the message is discarded. (Alternatively, a message could be sent to a static daemon agent that uses `iflocal` to deliver messages locally.)

The Nomadic Pict language also has a construct `<a>c!m` for *location-independent (LI)* communication, which does not require the agent's site to be specified. An application-specific *overlay network* will deliver message `m` to agent `a` irrespective of its current location. It is guaranteed that the message will be delivered despite of any agent migrations. Different LI overlay networks can be chosen from the package; the choice depends on the application.

2.4 Relocatable Computation

In a grid system, it is inevitable that some machines may partially fail or slow down. Thus, the grid admins should be able to relocate processes running on these machines to non-faulty nodes. Now, relocatable computation is a pervasive reality, though at the level of virtual machines rather than high-level languages. In a recent paper, we discuss the use of the Nomadic Pict calculus for verifying overlay networks for relocatable computations [16].

Below we demonstrate the use of relocatable computation for active messages. Let us assume that on the site `''sirius.cs.put.pl'':5000`, an agent `Smith` has been created, waiting for a message:

```
new ch : ^String

agent Smith =
  ch?* msg= print! msg
```

On another site, a function `dispatch` is defined that spawns an agent `messenger` for delivering message content of type `X`, to a recipient described using a triple of agent, site and channel names passed as arguments. After `messenger` is created, the function returns a value `0`. Below the function is called, resulting in `messenger` migrating to `Smith`'s site and delivering a message locally. The recipient's agent/channel names can be obtained using a name server (see Section 2.7).

```
def dispatch (#X  a:Agent  s:Site  c:^X  msg:X) : Int =
(  agent messenger =
   (
     migrate to s
     iflocal <a> c! msg
       then print! "OK, delivered."
       else print! "No recipient."
   )
0)

val stat = (dispatch Smith ''sirius.cs.put.pl'':5000 ch "Hello!")
```

To support different types of messages the channel `c` in `dispatch` has a polymorphic type (from Pict), which is defined by a type variable `X`. The type variable can be specialized to *any* type. It our example, it is specialized to `String`, when substituted in the function call by a channel `ch` of type `^String`.

The migrating agent `messenger` can be an arbitrary program, e.g. a presenter of the e-mail content. Thus, we can dynamically add some new computation on `Smith`'s node, even if the original program on this node was not designed for this. If needed, `messenger` could also voluntarily relocate to another server using `migrate to` and continue computation there.

2.5 Types for Input/Output Modalities

In some programs, we may intend a communication channel to be used only for inputs or only for outputs. Otherwise, a program may be incorrect. Below we illustrate the use of the Pict type system for safe programming of input/output modalities. This type system has been extended in Nomadic Pict for distributed programming. Below is an example program implementing a server function, which creates on demand (in response to the function call) a fresh channel that can be used for communication with the server, as explained below.

```
def server () : !Int =
 (new c : ^Int
  run c? msg = printi! msg
  c)

val x = (server)
run x! 10
```

Execution of the above function `server` creates a fresh communication channel `c` carrying integers, and waits for a message on it. The returned channel is assigned to a variable `x` that is later used for an output (of a value `10`). The type of channels returned by the function is `!Int` instead of `^Int`, where `!` (exclamation mark) means that the channel name has only an *output capability*, i.e. it cannot be used for an input. Thus, we can guarantee that only the server process can read on this channel. This mechanism supports *confidentiality* since no other process can read from this channel.

2.6 Types for Variant Messages

It would be inconvenient to create and use a different channel for every new type of a message. How to communicate messages of different types in the same channel, and still be able to statically check if the types of marshaled/unmarshaled values are correct ? Below is an example program that uses a suitable mechanism, adopted in Nomadic Pict.

```
new c : ^[num>Int text>String]

run c ?* msg =
  switch msg of
  (
    num> v : Int -> printi! (+ v 1)
    text> s : String -> print! (+$ "message: " s)
  )

run  c! [num> 2]
run  c! [text> "foo"]
```

The above program creates a channel `c` that has a *variant type* of channels carrying either messages of type `num>Int` or `text>String`, where `num` and `text` are labels that differentiate between the types. A message received on this channel must be first resolved using a construct `switch ... of`. The construct allows messages to be matched against patterns, followed by corresponding actions.

In the above program, two types of messages are sent on channel `c`: an integer `2` and a string `"foo"`. The integer message is incremented and printed out, while the string message is first concatenated (using a function `+$`) with another string, and then printed out. The program does not compile if we would try to send on the `c` channel a value of a different type.

2.7 Types for Dynamic Messages

When programs are compiled separately and should connect each other, the usual approach is to publish names of channels/agents/sites at some *name server*. The address of this server is known to all processes in the grid, so that any new agent joining the system can get the public names and use them for communication. For example, in the program in Section 2.4, a process calling function `dispatch` could obtain the channel and agent names of the message recipient from a name server, using library functions `publish` and `subscribe`.

In untyped languages, the publish/subscribe code is prone to errors that can be difficult to find. In typed languages, type-checking of dynamic values is usually done entirely at runtime. However, the risk of producing erroneous code exists if the language does not force the programmer to implement exceptions.

Below is a code fragment of a new joining process. The program uses *dynamic types*, i.e. types that are not erasured by the compiler, but which accompany values at runtime. Dynamic types are erasured in Nomadic Pict explicitly, using a construct `typecase`, which requires exception code to be specified.

```
new c : ^Dyn

run c?* v =
  typecase v of
    [ a:Agent s:Site d: ^String ] -> <a@s> d! "Hello world!"
    stat : Int -> printi! stat
  else
    print! "Type not recognized!"

run c!(dynamic 3)
```

The above program creates a name server channel `c` that can be used for carrying messages of *any* type at the same time. Then, we implement a client process that expects only two types of messages to be received from channel `c`: either a triple of agent, site and channel names to be used for communication with the agent, or some integer value. If a message received does not match these types, exception code is executed (here, an error message is printed).

Contrary to statically checked variant types, described in Section 2.6, type-checking is done dynamically, when the values are resolved by `typecase`. A dynamic value can be created using construct `dynamic`, which marshals a value with a runtime representation of its type.

3 Conclusions

In the paper, we gave some taste of distributed programming in Nomadic Pict–a prototype, strongly-typed language based on the π-calculus. As other languages based on process calculi, it offers abstractions that are small and easy to learn. In the paper, we demonstrated the use of statically and dynamically typed first-class channels for safe message-passing communication. Notably, Nomadic Pict also supports relocatable computation—a rare feature that greatly simplifies system bootstrapping and enables active messages. We think that this sort of programming abstractions are a tool that would be useful for future concurrent scripting languages in modern deployment platforms, such as many-core processors, grids, web servers and datacentres. In the paper, we demonstrated the main features of Nomadic Pict, focusing on grid programming. This prototype serves as a proof-of-concept and lacks many features that are necessary for practical applications, such as integration with other languages and environments. It would be interesting future work to develop a concurrent scripting language for managing relocatable virtual machines, using the abstractions of Nomadic Pict.

References

1. Armstrong, J.L., Virding, R.: Erlang – an experimental telephony switching language. In: Proc. XIII International Switching Symposium (May-June 1991)
2. Bhargavan, K., Gordon, A.D., Narasamdya, I.: Service Combinators for Farming Virtual Machines. In: Wang, A.H., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 33–49. Springer, Heidelberg (2008)
3. Billings, J., Sewell, P., Shinwell, M., Strniša, R.: Type-safe distributed programming for OCaml. In: Proc. 2006 ACM SIGPLAN Workshop on ML (2006)
4. Bloom, B., Nystrom, N., Östlund, J., Richards, G., Strniša, R., Vitek, J., Wrigstad, T.: Thorn–robust, concurrent, extensible scripting on the JVM. In: Proc. OOPSLA 2009 (October 2009)
5. Cardelli, L., Gordon, A.D.: Mobile Ambients. Theoretical Computer Science (TCS) 240(1), 177–213 (2000)
6. Clojure. Distribution files and documentation, http://clojure.org/
7. Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G., Larus, J.R., Levi, S.: Language support for fast and reliable message-based communication in Singularity OS. In: Proc. EuroSys 2006 (April 2006)
8. Fournet, C., Gonthier, G., Lévy, J.-J., Maranget, L., Rémy, D.: A Calculus of Mobile Agents. In: Sassone, V., Montanari, U. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 406–421. Springer, Heidelberg (1996)
9. Honda, K., Vasconcelos, V.T., Kubo, M.: Language Primitives and Type Discipline for Structured Communication-Based Programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)
10. Lopes, L., Figueira, Á., Silva, F., Vasconcelos, V.T.: A concurrent programming environment with support for distributed computations and code mobility. In: Proc. CLUSTER 2000 (November 2000)
11. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, Parts I and II. Information and Computation 100(1), 1–77 (1992)
12. De Nicola, R., Ferrari, G.L., Pugliese, R.: Klaim: A kernel language for agents interaction and mobility. IEEE TSE 24(5), 315–330 (1998)
13. Nomadic Pict Language, http://www.cs.put.poznan.pl/pawelw/npict
14. Pierce, B.C., Turner, D.N.: Pict: A programming language based on the pi-calculus. In: Proof, Language and Interaction: Essays in Honour of Robin Milner. MIT Press (2000)
15. Sewell, P., Leifer, J.J., Wansbrough, K., Nardelli, F.Z., Allen-Williams, M., Habouzit, P., Vafeiadis, V.: Acute: High-level programming language design for distributed computation. In: Proc. ICFP 2005 (September 2005)
16. Sewell, P., Wojciechowski, P.T.: Verifying overlay networks for relocatable computations (or: Nomadic Pict, relocated). In: Proc. Workshop on the Rise and Rise of the Declarative Datacentre, Microsoft MSR-TR-2008-61 (May 2008)
17. Sewell, P., Wojciechowski, P.T., Pierce, B.C.: Location-Independent Communication for Mobile Agents: A Two-Level Architecture. In: Bal, H.E., Cardelli, L., Belkhouche, B. (eds.) ICCL-WS 1998. LNCS, vol. 1686, pp. 1–31. Springer, Heidelberg (1999)
18. Sewell, P., Wojciechowski, P.T., Unyapoth, A.: Nomadic Pict: Programming languages, communication infrastructure overlays, and semantics for mobile computation. ACM TOPLAS 32(4), 1–63 (2010)
19. Wojciechowski, P.T., Sewell, P.: Nomadic Pict: Language and infrastructure design for mobile agents. IEEE Concurrency 8(2), 42–52 (2000)

Author Index