

Tutorial

LNCS 5235

Ralf Lämmel  
Joost Visser  
João Saraiva (Eds.)

# Generative and Transformational Techniques in Software Engineering II

International Summer School, GTTSE 2007  
Braga, Portugal, July 2007  
Revised Papers



*Commenced Publication in 1973*

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

*Lancaster University, UK*

Takeo Kanade

*Carnegie Mellon University, Pittsburgh, PA, USA*

Josef Kittler

*University of Surrey, Guildford, UK*

Jon M. Kleinberg

*Cornell University, Ithaca, NY, USA*

Alfred Kobsa

*University of California, Irvine, CA, USA*

Friedemann Mattern

*ETH Zurich, Switzerland*

John C. Mitchell

*Stanford University, CA, USA*

Moni Naor

*Weizmann Institute of Science, Rehovot, Israel*

Oscar Nierstrasz

*University of Bern, Switzerland*

C. Pandu Rangan

*Indian Institute of Technology, Madras, India*

Bernhard Steffen

*University of Dortmund, Germany*

Madhu Sudan

*Massachusetts Institute of Technology, MA, USA*

Demetri Terzopoulos

*University of California, Los Angeles, CA, USA*

Doug Tygar

*University of California, Berkeley, CA, USA*

Gerhard Weikum

*Max-Planck Institute of Computer Science, Saarbruecken, Germany*

Ralf Lämmel  
Joost Visser  
João Saraiva (Eds.)

# Generative and Transformational Techniques in Software Engineering II

International Summer School, GTTSE 2007  
Braga, Portugal, July 2-7, 2007  
Revised Papers

Volume Editors

Ralf Lämmel  
Universität Koblenz-Landau, Fachbereich 4  
Institut für Informatik, B127  
Universitätsstraße 1, 56070 Koblenz, Germany  
E-mail: rlaemmel@acm.org

Joost Visser  
Software Improvement Group  
A.J. Ernststraat 595-H, 1082 LD Amsterdam, The Netherlands  
E-mail: j.visser@sig.nl

João Saraiva  
Universidade do Minho, Departamento de Informática  
Campus de Gualtar, 4710-057 Braga, Portugal  
E-mail: jas@di.uminho.pt

Library of Congress Control Number: Applied for

CR Subject Classification (1998): B.2, C.1, C.2, C.5, D.2, D.3

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743  
ISBN-10 3-540-88642-7 Springer Berlin Heidelberg New York  
ISBN-13 978-3-540-88642-6 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media  
springer.com

© Springer-Verlag Berlin Heidelberg 2008  
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India  
Printed on acid-free paper SPIN: 12527151 06/3180 5 4 3 2 1 0



# Preface

The second instance of the international summer school on Generative and Transformational Techniques in Software Engineering (GTTSE 2007) was held in Braga, Portugal, during July 2–7, 2007. This volume contains an augmented selection of the material presented at the school, including full tutorials, short tutorials, and contributions to the participants workshop.

The GTTSE summer school series brings together PhD students, lecturers, technology presenters, as well as other researchers and practitioners who are interested in the generation and the transformation of programs, data, models, metamodels, documentation, and entire software systems. This concerns many areas of software engineering: software reverse and re-engineering, model-driven engineering, automated software engineering, generic language technology, to name a few. These areas differ with regard to the specific sorts of metamodels (or grammars, schemas, formats etc.) that underlie the involved artifacts, and with regard to the specific techniques that are employed for the generation and the transformation of the artifacts. The first instance of the school was held in 2005 and its proceedings appeared as volume 4143 in the LNCS series.

The 2007 instance of GTTSE offered eight tutorials, given by renowned representatives of complementary approaches and problem domains. Each tutorial combines foundations, methods, examples, and tool support. The program of the summer school also featured eight invited technology presentations, which presented concrete support for generative and transformational techniques. These presentations complemented each other in terms of the chosen application domains, case studies, and the underlying concepts. Furthermore, the program of the school included a participants workshop to which all students of the summer school were asked to submit an extended abstract beforehand. The Organizing Committee reviewed these extended abstracts and invited 12 students to present their work at the workshop.

This volume contains extended and revised versions of the material presented at the summer school. Each of the seven full tutorials included here was reviewed by two members of the Scientific Committee of GTTSE 2007. The five included short tutorials were reviewed by three members each. The four included participant contributions were selected on the basis of three reviews for each such submission. All submissions were carefully revised based on the reviews.

We are grateful to all lecturers and participants of the school for their enthusiasm and hard work in preparing excellent material for the school itself and for these proceedings. Due to their efforts the event was a great success, which we trust the reader finds reflected in this volume.

April 2008

Ralf Lämmel  
Joost Visser  
João Saraiva

# Organization

GTTSE 2007 was hosted by the Departamento de Informática, Universidade do Minho, Braga, Portugal.

## Executive Committee

Program Co-chair	Ralf Lämmel (Microsoft, Redmond, USA)
Program Co-chair	Joost Visser (Software Improvement Group, Amsterdam, The Netherlands)
Organizing Chair	João Saraiva (Universidade do Minho, Braga, Portugal)

## Scientific Committee

Uwe Aßmann	TU Dresden, Germany
Paulo Borba	Universidade Federal de Pernambuco, Brazil
Mark van den Brand	Technical University Eindhoven, The Netherlands
Charles Consel	LaBRI/INRIA, France
Jim Cordy	Queen's University, Canada
Alcino Cunha	Universidade do Minho, Portugal
Jean-Luc Dekeyser	Université des Sciences et Technologies de Lille, France
Andrea DeLucia	Università di Salerno, Italy
Stephen Freund	Williams College, UK
Jeff Gray	University of Alabama at Birmingham, USA
Reiko Heckel	University of Leicester, UK
Görel Hedin	Lund Institute of Technology, Sweden
Dirk Heuzeroth	IBM Deutschland Entwicklung GmbH, Germany
Zhenjiang Hu	The University of Tokyo, Japan
Ralf Lämmel	Microsoft Corporation, USA
Julia Lawall	University of Copenhagen, Denmark
Cristina Lopes	University of California at Irvine, USA
Tom Mens	University of Mons-Hainaut, Belgium
Marjan Mernik	University of Maribor, Slovenia
Klaus Ostermann	Technical University Darmstadt, Germany
Jens Palsberg	UCLA, USA
Benjamin C. Pierce	University of Pennsylvania, USA
João Saraiva	Universidade do Minho, Portugal
Andy Schürr	Technical University Darmstadt, Germany

## VIII Organization

Anthony Sloane	Macquarie University, Australia
Perdita Stevens	University of Edinburgh, UK
Peter Thiemann	Universität Freiburg, Germany
Simon Thompson	University of Kent, UK
Joost Visser	Universidade do Minho, Portugal
Victor Winter	University of Nebraska at Omaha, USA
Eric Van Wyk	University of Minnesota, USA
Albert Zündorf	University of Kassel, Germany

## Organizing Committee

Alcino Cunha	Universidade do Minho, Braga, Portugal
João Saraiva	Universidade do Minho, Braga, Portugal
Ricardo Vilaça	Universidade do Minho, Braga, Portugal
Joost Visser	Software Improvement Group, Amsterdam, The Netherlands

## Sponsoring Institutions

Centro de Ciências e Tecnologias de Computação  
Luso-American Foundation  
Software Improvement Group



**LUSO-AMERICAN**  
FOUNDATION



Software Improvement Group

# Table of Contents

---

## I Full Tutorials

---

Design Space of Heterogeneous Synchronization . . . . .	3
<i>Michał Antkiewicz and Krzysztof Czarnecki</i>	
Software Reuse beyond Components with XVCL (Tutorial) . . . . .	47
<i>Stan Jarzabek</i>	
.QL: Object-Oriented Queries Made Easy . . . . .	78
<i>Oege de Moor, Damien Sereni, Mathieu Verbaere, Elnar Hajiyeu, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, and Julian Tibble</i>	
Transforming Data by Calculation . . . . .	134
<i>José N. Oliveira</i>	
How to Write Fast Numerical Code: A Small Introduction . . . . .	196
<i>Srinivas Chellappa, Franz Franchetti, and Markus Püschel</i>	
A Gentle Introduction to Multi-stage Programming, Part II . . . . .	260
<i>Walid Taha</i>	
WebDSL: A Case Study in Domain-Specific Language Engineering . . . . .	291
<i>Elco Visser</i>	

---

## II Short Tutorials

---

Model-Driven Engineering of Rules for Web Services . . . . .	377
<i>Marko Ribarić, Dragan Gašević, Milan Milanović, Adrian Giurca, Sergey Lukichev, and Gerd Wagner</i>	
An Introduction to Context-Oriented Programming with ContextS . . . . .	396
<i>Robert Hirschfeld, Pascal Costanza, and Michael Haupt</i>	
A Landscape of Bidirectional Model Transformations . . . . .	408
<i>Perdita Stevens</i>	
Evolving a DSL Implementation . . . . .	425
<i>Laurence Tratt</i>	
Adding Dimension Analysis to Java as a Composable Language Extension (Extended Abstract) . . . . .	442
<i>Eric Van Wyk and Yogesh Mali</i>	

---

### III Participants Contributions

---

Model Transformations for the Compilation of Multi-processor Systems-on-Chip . . . . .	459
<i>Éric Piel, Philippe Marquet, and Jean-Luc Dekeyser</i>	
Implementation of a Finite State Machine with Active Libraries in C++ . . . . .	474
<i>Zoltán Juhász, Ádám Sipos, and Zoltán Porkoláb</i>	
Automated Merging of Feature Models Using Graph Transformations . . .	489
<i>Sergio Segura, David Benavides, Antonio Ruiz-Cortés, and Pablo Trinidad</i>	
Modelling the Operational Semantics of Domain-Specific Modelling Languages . . . . .	506
<i>Guido Wachsmuth</i>	
<b>Author Index</b> . . . . .	521

**Part I**  
**Full Tutorials**

# Design Space of Heterogeneous Synchronization

Michał Antkiewicz and Krzysztof Czarnecki

University of Waterloo  
Generative Software Development Lab  
{mantkiew,k2czarne}@uwaterloo.ca  
<http://gsd.uwaterloo.ca>

**Abstract.** This tutorial explores the design space of *heterogeneous synchronization*, which is concerned with establishing consistency among artifacts that conform to different schemas or are expressed in different languages. Our main application scenario is synchronization of software artifacts, such as code, models, and configuration files. We classify heterogeneous synchronizers according to the cardinality of the relation that they enforce between artifacts, their directionality, their incrementality, and whether they support reconciliation of concurrent updates. We then provide a framework of artifact operators that describes different ways of building heterogeneous synchronizers, such as synchronizers based on artifact or update translation. The design decisions within the framework are described using feature models. We present 16 concrete instances of the framework, discuss tradeoffs among them, and identify sample implementations for some of them. We also explore additional design decisions such as representation of updates, establishing correspondence among model elements, and strategies for selecting a single synchronization result from a set of alternatives. Finally, we discuss related fields including data synchronization, inconsistency management in software engineering, model management, and model transformation.

## 1 Introduction

The sheer complexity of today’s software-intensive systems can only be conquered by incremental and evolutionary development. As Brooks points out [1], “teams can grow much more complex entities in four months than they can *build*,” where “build” refers to the traditional engineering approach of specifying structures accurately and completely before they are constructed. However, despite important advances in software methods and technology, such as agile development and object orientation, evolving software to conform to a changed set of requirements is notoriously hard. Evolution is hard because it requires keeping multiple software artifacts such as specifications, code, configuration files, and tests, consistent. A simple change in one artifact may require multiple changes in many artifacts and current development tools offer little help in identifying the artifacts and their parts that need to be changed and performing the changes.

*Synchronization* is the process of enforcing consistency among a set of artifacts and *synchronizers* are procedures that automate—fully or in part—the synchronization process. *Heterogeneous synchronizers* synchronize artifacts that conform to different schemas or are expressed in different languages. Many processes in software engineering can be viewed as heterogeneous synchronization. Examples include reverse engineering models from code using code queries, compiling programs to object code, generating program code from models, round-trip engineering between models and code, and maintaining consistency among models expressed in different modeling languages.

While many approaches to synchronization of heterogeneous software artifacts exist, it is not clear how they differ and how to choose among them. The purpose of this tutorial is to address this problem. We explore the design space of heterogeneous synchronizers. We cover both the simpler synchronization scenarios where some artifacts are never edited directly but are re-generated from other artifacts and the more complex scenarios where several artifacts that can be modified directly need to be synchronized. Both kinds of scenarios occur in software development. Example of the simpler scenario is generation of object code from source code. The need for synchronizing multiple heterogeneous artifacts that are edited directly arises in *multi-view development* [2,3], where each stakeholder can understand and change the system through an appropriate view. The motivation for providing different views is that certain changes may be most conveniently expressed in a particular view, e.g., because of conciseness of expression or the familiarity of the a stakeholder with a particular view.

The tutorial is organized as follows. In Section 2, we present kinds of relations among software artifacts and concrete examples of such relations. In Section 3, we introduce kinds of synchronizers that can be used for reestablishing the consistency among artifacts. We classify heterogeneous synchronizers according to the cardinality of the relation that they enforce between artifacts, their directionality, their incrementality, and whether they support reconciliation of concurrent updates in Sections 4-6. The need for reconciliation arises in the context of concurrent development, where developers need to concurrently modify multiple related artifacts. We provide a framework of artifact operators that describes different ways of building heterogeneous synchronizers, such as synchronizers based on artifact or update translation. The operator-based approach is inspired by the manifesto for model merging by Brunet et al. [4]. The design decisions within the framework are described using feature models. We present 16 concrete instances of the framework, discuss their properties, and identify sample implementations for some of them. We summarize the synchronizers and discuss the tradeoffs among the synchronizers in Section 7. In Section 8, we explore additional design decisions such as representation of updates, establishing correspondence among model elements, and strategies for selecting a single synchronization result from a set of alternatives. Finally, we discuss related fields including data synchronization, inconsistency management in software engineering, model management, and model transformation in Section 9. We conclude in Section 10.



**Purpose and Approach.** The purpose of the tutorial is to present a wide family of scenarios that require heterogeneous synchronization and the different solutions that can be applied in each scenario. The solutions are characterized by the scenarios they support, such as unidirectional or bi-directional synchronization, and the different design choices that can be made when constructing a synchronizer. The discussion of the scenarios and design choices is made more precise by considering the properties of the relations that are to be maintained among sets of artifacts and formulating the synchronizers using a set of artifact operators. The formalization does not consider the structure of the artifacts or their semantics. Whereas such a treatment would allow more precision in the analysis of choices, it would introduce a considerable amount of additional complexity and detail. We leave this endeavor for future work.

The intended audience is primarily those interested in building heterogeneous synchronizers. This audience can learn about the different design choices, the tradeoffs among the choices, and examples of systems implementing particular kinds of synchronizers. Furthermore, the operator-based formalization of the different kinds of synchronizers may also be of interest to researchers studying the semantics of model transformations.

## 2 Relations among Software Artifacts

Modern software development involves a multitude of artifacts of different types, such as requirements and design models, program code, tests, XML configuration files, and documentation. Since the artifacts describe the same software system, they are related to each other in various ways. For example, a design model and its implementation code should be related by *refinement*. Furthermore, both the code and its XML configuration files have to use *consistent* names and identifiers. Also, the design model should *conform* to the *metamodel* defining the abstract syntax of the language in which the model is expressed.

In this tutorial, we usually consider software artifacts simply as typed values. An *artifact type* is a set of artifacts and it may be viewed as an extensional definition of a language. For example, assuming that  $\mathcal{J}$  denotes the Java language, we write  $P \in \mathcal{J}$  in order to denote that the artifact  $P$  is a Java program. Alternatively, we may also indicate the type of an artifact using a subscript, e.g.,  $P_{\mathcal{J}}$ . On few occasions, we also consider the internal structure of an artifact, in which case we view an artifact as a collection of elements with attributes and links among the elements.

When an artifact is modified, related artifacts need to be updated in order to reestablish the relations. For example, when the design model is changed, the implementation code may need to be updated, and vice versa. The general problem of identifying relations among artifacts, detecting inconsistencies, handling of inconsistencies, and establishing relations among artifacts is referred to as *consistency management*. Furthermore, the update of related artifacts in order to re-establish consistency after changes to some of these artifacts is known as *synchronization*,

*change propagation*, or *co-evolution*. We refer to synchronization as *heterogeneous* if the artifacts being synchronized are of different types.

**Definition 1.** CONSISTENT ARTIFACTS. *We say that two artifacts  $S_S$  and  $T_T$  are consistent or synchronized with respect to the relation  $R \subseteq \mathcal{S} \times \mathcal{T}$  iff  $(S_S, T_T) \in R$ .*

In general, two or more artifacts need not be consistent at all times [2,5]. For example, the implementation code may be out of sync with its design model while several changes are being applied to the model. In this case, the inconsistency between the code and the design is desirable and should be tolerated. Only after the changes are completed, the code is updated and the consistency re-established. Consequently, some authors use the term *inconsistency management* [6,7,8].

The relations among software artifacts may have different properties. For a binary relation  $R \subseteq \mathcal{S} \times \mathcal{T}$ , we distinguish among the following three interesting cases:

1.  $R$  is a *bijection*. This is the *one-to-one* case where each artifact in  $\mathcal{S}$  corresponds to exactly one artifact in  $\mathcal{T}$  and vice versa.
2.  $R$  is a *total and surjective function*. This is the *many-to-one* case where each artifact in  $\mathcal{S}$  corresponds to exactly one artifact in  $\mathcal{T}$  and each artifact in  $\mathcal{T}$  corresponds to at least one artifact in  $\mathcal{S}$ .
3.  $R$  is a *total relation*. This is the *many-to-many* case where each artifact in  $\mathcal{S}$  corresponds to at least one artifact in  $\mathcal{T}$  and each artifact in  $\mathcal{T}$  corresponds to at least one artifact in  $\mathcal{S}$ .

Note that all of the above cases assume total binary relations. In practice, cases where  $R$  covers  $\mathcal{S}$  or  $\mathcal{T}$  only partially can be handled, e.g., by making these sets smaller using additional well-formedness constraints or by introducing a special value representing an error. For example, a source artifact that has no proper translation into the target type would be mapped to such an error element in the target type. Furthermore, the above cases are distinguished only in regard to the correspondence between whole artifacts. In practice, the artifact relations also need to establish correspondence between the structures within the artifacts, i.e., the correspondence between the elements and links in one artifact and the elements and links in another artifact. We will explicitly refer to this structural correspondence whenever necessary. Finally, the artifact relations need not be binary, but could be relating three or more sets of artifacts.

**Examples.** Let us look at some examples of relations among different kinds of artifacts.

*Example 1.* Simple class diagrams and KM3.

KM3 [9] is a textual notation that can be used for the specification of simple class diagrams. The relation between graphical class diagrams and their textual specifications is a *bijection*. In this example, assuming that the layout of diagrams

and text is irrelevant, artifacts expressed in one language can be translated into the other language without any loss of information.

*Example 2.* Java and type hierarchy.

A type hierarchy of a Java program is a graph in which *classes* and *interfaces* are nodes and *extends* and *implements* relations are edges. Such a type hierarchy is an abstraction of a Java program because it contains a subset of the information contained in the program and it does not contain any additional information that does not exist in the program. Furthermore, many different Java programs may have the same type hierarchy. Therefore, the relation between a Java program and its type hierarchy is a *function*.

*Example 3.* Java and XML and Struts Framework-Specific Modeling Language (FSML).

Struts FSML [10] is a modeling language that can be used for describing how Struts' concepts *actions*, *forms*, and *forwards* are implemented in an application consisting of Java code and XML configuration files. A model expressed in the Struts FSML is an abstraction of the code and it can be fully recreated from the code. Actions, forms, and forwards can be implemented in the code in various ways, some of which are equivalent with respect to the model. For example, a Java class is represented in the model as an action if it is a direct or indirect subclass of the Struts' `Action` class. The relation between the code and the model expressed in Struts FSML is a *function*: parts of the code do not have any representation in the model and equivalent ways of implementing actions, forms, and forwards are represented the same way in the model.

*Example 4.* UML class diagrams and RDBMS.

This example considers UML class diagrams and relational database schemas. The relation between the two languages is a *general relation* because inheritance and associations in class diagrams can be represented in many different ways in database schemas and every database schema can be represented using different class diagrams with or without inheritance [11]. For example, each single class can be mapped to a separate table or an entire class hierarchy can be mapped to a single table. Furthermore, different class hierarchies may still be translated into the same table structure.

*Example 5.* Statecharts and sequence diagrams.

The relation between statecharts and sequence diagrams is a *general relation* because a statechart can be synthesized from multiple sequence diagrams and a given sequence diagram can be produced by different statecharts.

*Example 6.* Metamodels and models.

In model-driven software development [12], the syntax of a modeling language is often specified as a class model, which is referred to as a *metamodel*. A metamodel defines all syntactically correct models and a model is syntactically correct if it conforms to its metamodel. As any other software artifacts, metamodels evolve over time. Some changes to the metamodels may break the conformance of existing models, in which case the models need to be updated [13]. The relation

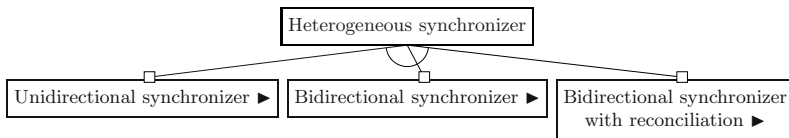
between a metamodel and a model is a *general relation* because many models can conform to a single metamodel and a single model can conform to many metamodels. As an example of the latter situation, consider two metamodels representing the same set of models, but one using abstract and concrete classes and the other using concrete classes only.

### 3 Mappings, Transforms, Transformations, Synchronizers, and Synchronizations

We refer to the specifications of relations among artifacts as *mappings*. Furthermore, we refer to programs that implement mappings as *transforms* and executions of those programs as *transformations*. In this tutorial, we focus on *synchronizers*, which are transforms used for (re-)establishing consistency among related artifacts. Consequently, we refer to the execution of a synchronizer as *synchronization*. Note that not every transform is a synchronizer. For example, *refactorings*, which change the structure of an artifact while preserving the artifact's semantics are transforms, but they are not synchronizers.






Transforms are executable programs, which may be interactive. For example, they may seek additional inputs from the user to decide among possible alternative results. In this tutorial, we model transforms as *computable functions*, where any additional inputs are given to the functions up-front as arguments. In particular, we represent interactive choices as *decision functions* that are passed as parameters to the transforms.

In the following sections we present various kinds of heterogeneous synchronizers that can be used to synchronize two artifacts, which we refer to as *source* and *target*. At the highest level, a synchronizer falls into one of the three distinct categories: *unidirectional*, *bidirectional*, and *bidirectional with reconciliation* (cf. Figure [1](#)). The three alternatives are represented as a *feature model* [\[14, 15\]](#). A feature model is a hierarchy of common and variable features characterizing the set of instances of a concept that is represented by the root of the hierarchy. In this tutorial, the features provide a terminology and a representation of the design choices for heterogeneous synchronizers. The subset of the feature model notation used in this tutorial is explained in Table [1](#). The three categories of synchronizers are modeled in Figure [1](#) as a group of three alternative features. Each of these alternative features is actually a reference to a more refined feature model that is presented later.



**Fig. 1.** Artifact synchronization synchronizers

**Table 1.** Feature modeling notation used in this tutorial

Symbol	Explanation
	Solitary feature with cardinality [1..1], i.e., <i>mandatory</i> feature
	Solitary feature with cardinality [0..1], i.e., <i>optional</i> feature
	Reference to feature $F$
	XOR feature group (groups alternative features)
	Grouped feature (a feature under a feature group)

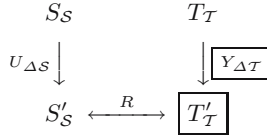
*Unidirectional synchronizers* synchronize the target artifact with the source artifact. *Bidirectional synchronizers* (without reconciliation) can be used to synchronize the target artifact with the source artifact and vice versa. They synchronize in one direction at a time, meaning that they are most useful if only one of the artifacts was changed since the last synchronization. Bidirectional synchronizers can also be used when both artifacts have changed since the last synchronization; however, they cannot be used to resolve conflicting changes to both artifacts, as one artifact acts as a slave and its changes may get overridden. Finally, *bidirectional synchronizers with reconciliation* can be used to synchronize both artifacts at the same time. Thus, these synchronizers are also applicable in situations where both artifacts were changed since the last synchronization and they can be used for conflict resolution in both directions.

## 4 Unidirectional Synchronizers

Unidirectional synchronization from  $\mathcal{S}$  to  $\mathcal{T}$  enforcing the relation  $R \subseteq \mathcal{S} \times \mathcal{T}$  involves up to four artifacts (cf. Figure 2):

1.  $S_S$  is the *original source artifact*, i.e., the version of the source artifact before it was modified by the developer;
2.  $T_T$  is the *original target artifact*, i.e., the version of the target artifact that co-existed with the original source artifact;
3.  $S'_S$  is the *new source artifact*, i.e., the version of the source artifact after it was modified by the developer; and
4.  $T'_T$  is the *new target artifact*, i.e., the version of the target artifact after synchronization with the new source artifact.

The first three of these artifacts are the ones that typically exist before the synchronization occurs. However, the first two are optional since the new source could have been created from scratch and the original target might have not been yet created. Note that we use the convention of marking new versions of artifacts by a prime.

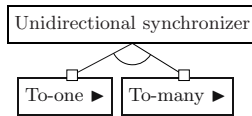


**Fig. 2.** Artifacts involved in unidirectional synchronization

The fourth artifact,  $T'_T$ , is the new target that needs to be computed during the synchronization. The enclosing boxes in Figure 2 indicate elements that are computed during the synchronization. The arrows pointing downwards in the figure denote updates:  $U_{\Delta S}$  is the update applied to the original source and  $Y_{\Delta T}$  is the target update resulting from the synchronization. The double-headed arrow between the new artifacts indicates that they are consistent, i.e.,  $(S'_S, T'_T) \in R$ . Note that, in general, the original artifacts  $S_S$  and  $T_T$  do not have to be consistent; however, some synchronizers might impose such a requirement.

A *unidirectional synchronizer* from  $\mathcal{S}$  to  $\mathcal{T}$  implementing the relation  $R \subseteq \mathcal{S} \times \mathcal{T}$  computes the new target  $T'_T$  given the new source  $S'_S$ , and optionally the original source  $S_S$  and the original target  $T_T$ , as inputs, such that the new source and the new target are consistent, i.e.,  $(S'_S, T'_T) \in R$ . Note that the new source can also be passed as input to the synchronizer indirectly by passing both the original source and the update of the source as inputs. Furthermore, some synchronizer variants require the original source and the original target to be consistent.

Unidirectional synchronizers can be implemented using different operators and the choices depend first and foremost on the cardinality of the end of the relation in the direction of which the synchronizers are executed. In particular, a synchronizer can be executed towards the cardinality of one, which we refer to as *to-one* case, and towards the cardinality of many, which we refer to as *to-many* case (cf. Figure 3).



**Fig. 3.** Unidirectional synchronizers

The to-one case corresponds to the situation where the mapping between source and target is a function from source to target, which also covers the case of a bijection. The mapping clearly specifies a single target artifact  $T'_T$  that a synchronizer has to return for a given source artifact  $S'_S$ . The to-many case corresponds to the situation where the mapping between source and target is not a function in the source-to-target direction. In other words, the relation is either a function in the target-to-source direction or a general relation. Consequently, the mapping may specify several alternative target artifacts that a synchronizer

could return for a given source artifact. Since all synchronizers are functions returning only a single synchronization result, too-many synchronizers will require a mechanism for selecting one target artifact from the set of possible alternatives.

#### 4.1 Unidirectional Synchronizers in To-One Direction

The unidirectional to-one case could be described as computing a “disposable view”, where the target  $T'_T$  is fully determined by the source  $S'_S$ . In other words, the source-to-target mapping is a function and the target can be automatically re-computed whenever needed based on the source  $S'_S$  only.

In general, a disposable view can be computed in an *incremental* or a *non-incremental* fashion. The non-incremental approach implies that the view is completely re-computed whenever the source is modified, whereas the incremental approach involves computing only the necessary updates to the existing view and applying these updates. As a result, all incremental synchronizers take the original target as a parameter.

All to-one synchronizers are *original-target-independent*, meaning that the computed new target does not depend on the original target in a mathematical sense. Although the incremental to-one synchronizers take the original target as a parameter, the new target depends only on the new source because the relationship between the new source and the new target is a function. The original target is used by the synchronizer implementation purely to improve performance, which is achieved by reusing structures from the original target and avoiding recomputing these structures. We present examples of *original-target-dependent* synchronizers in Section 4.2.

Depending on the operator that is used to translate between source and target artifact types, we distinguish among three fundamental ways of realizing unidirectional to-one synchronizers (cf. Figure 4). The first synchronizer variant is non-incremental and it uses *artifact translation*, an operator that translates an entire source artifact into a consistent target artifact. The other two variants are incremental. The second variant uses *heterogeneous artifact comparison*, an operator that directly compares two artifacts of *different* types and produces an update that can be applied to the second artifact in order to make it consistent with the first artifact. The third variant uses *update translation*, an operator that

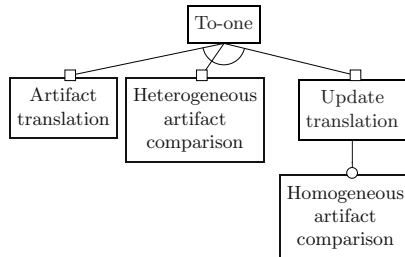
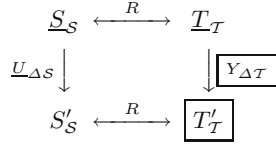


Fig. 4. Operators used in to-one unidirectional synchronizers



**Fig. 5.** Artifacts involved in unidirectional synchronization using update translation

translates an update to the source artifact into a consistent update of the target artifact. In addition, update translation expects the original source and the original target to be consistent. The artifacts involved in the synchronization using update translation are shown in Figure 5. The input artifacts are underlined. As an option, the transform may use *homogeneous artifact comparison* to compute the source update as a difference between the original and the new source.

**Artifact translation.** The non-incremental variant of the to-one synchronizer uses an operator that translates a source artifact into a consistent target artifact.

**Operator 1.** *Artifact translation:*  $AT_{S,T} : S \rightarrow T$ . For an artifact  $S_S$ , the operator  $AT_{S,T}(S_S)$  computes  $S_T$  such that  $(S_S, S_T) \in R$ .

In this tutorial, operators are defined generically over artifact types and the type parameters are specified as subscripts. For example, the operator  $AT_{S,T}$  has the artifact type parameters  $S$  and  $T$  and these parameters are used in the operator's signature.

We are now ready to state the non-incremental to-one synchronizer. We present all synchronizers using the form *input+ precondition\**  $\implies$  *computation*  $\implies$  *output+*, which makes the input artifact(s), the precondition(s) (if any), the computation steps, and the output artifact(s) explicit. This form may seem too verbose for the following simple synchronizer, but its advantages become apparent for more complex synchronizers.

**Synchronizer 1.** *Unidirectional, non-incremental, and to-one synchronizer using artifact translation:*

$\boxed{S}T_{S,T} : S \rightarrow T$

$$S'_S \implies T'_T = AT_{S,T}(S'_S) \implies T'_T$$

In this non-incremental variant, the new source artifact is translated into the new target artifact, which then replaces the original target artifact.

### Examples for Synchronizer 1

*Example 7.* Type hierarchy.

Examples of Synchronizer 1 are type hierarchy extractors for object-oriented programs (cf. Example 2). Such extractors are offered by many integrated development environments (IDEs).



*Example 8.* Reverse engineering in FSMs.

Another example of Synchronizer [1](#) is reverse engineering of framework-based Java code in FSMs (cf. Example [3](#)). The result of reverse engineering is a framework-specific model that describes how framework abstractions are implemented in the application code [\[16,10\]](#). For any application code, a unique model is retrieved using code queries.

*Example 9.* Lenses in Harmony.

Synchronizer [1](#) corresponds to the *get* function in Lenses [\[17\]](#). In Lenses, the source-to-target relationship is many-to-one and the target is also referred to as *view*. A *get* function takes the new source and creates the corresponding new view for it. A full lens, as shown later, is a bidirectional synchronizer and consists of two functions: *get* and *putback*.

**Updates.** Incremental synchronization can be achieved either by coercing the original target artifact into conformance with the new source artifact or by translating updates of the source artifact into the updates of the target artifact. Both variants require the notion of an *update*.

**Definition 2.** UPDATE. *An update  $U : \mathcal{S} \rightarrow \mathcal{S}$  for artifact(s) of type  $\mathcal{S}$  is a partial function that is defined for at least one artifact  $S_S$ . Artifacts on which an update is defined are referred to as reference artifacts of that update.*

The intuition behind an update is that it connects an original version of an artifact with its new version, e.g.,  $S'_S = U_{\Delta\mathcal{S}}(S_S)$ . Note that we abbreviate the space of all partial functions  $\mathcal{S} \rightarrow \mathcal{S}$  as  $\Delta\mathcal{S}$  and we use this abbreviation to specify the type of an update.

The size of the set of reference artifacts of an update can vary. An extreme case is when an update is applicable to only a single artifact. A more practical solution is to implement updates so that they can be applied to a number of artifacts that share certain characteristics. For example, an update could be defined so that it applies to all artifacts that contain a certain structure that the update modifies.

In practice, we can think of an update as a program that takes the original version of an artifact and returns its new version. The update instructions, such as inserting or removing elements, could be recorded while the user edits the original artifact. The recorded sequence can then be applied to a reference artifact, e.g., the original artifact.

Alternatively, an update can be computed using a *homogeneous artifact comparison* operator, which takes an original version of an artifact and its new version and returns an update connecting the two. We refer to this comparison operator as *homogeneous* since it takes two artifacts of the same type.

**Operator 2.** *Homogeneous artifact comparison:  $AC_S : \mathcal{S} \times \mathcal{S} \rightarrow \Delta\mathcal{S}$ . For artifacts  $S_S$  and  $S'_S$ , the operator  $AC_S(S_S, S'_S)$  computes  $U_{\Delta\mathcal{S}}$  such that  $S'_S = U_{\Delta\mathcal{S}}(S_S)$ .*

We further discuss the design choices for creating and representing updates in Section [8.1](#).

**Heterogeneous artifact comparison.** The first incremental synchronizer uses *heterogeneous artifact comparison*, an operator that directly compares two artifacts of *different* types and produces an update that can be applied to the second artifact in order to make it consistent with the first artifact.

**Operator 3.** *Heterogeneous artifact comparison:*  $AC_{S,T} : \mathcal{S} \times \mathcal{T} \rightarrow \Delta\mathcal{T}$ . For artifacts  $S'_S$  and  $T_T$ , the operator  $AC_{S,T}(S'_S, T_T)$  computes an update  $U_{\Delta\mathcal{T}}$  such that  $(S'_S, U_{\Delta\mathcal{T}}(T_T)) \in R$ .

The incremental synchronizer using *heterogeneous artifact comparison* takes the original target in addition to the new source as an input and produces the new target.

**Synchronizer 2.** *Unidirectional, incremental, original-target-independent, and to-one synchronizer using heterogeneous artifact comparison:*

$$\mathbb{R}_{S,T} : \mathcal{S} \times \mathcal{T} \rightarrow \mathcal{T}$$

$$S'_S, T_T \implies \begin{array}{l} U_{\Delta\mathcal{T}} = AC_{S,T}(S'_S, T_T) \\ T'_T = U_{\Delta\mathcal{T}}(T_T) \end{array} \implies T'_T$$

In general, the synchronizer needs to analyze the original target with respect to the new source, compute the updates, and apply the updates to the original target. Although the above formulation separates the update computation and application, all these actions could be performed in one pass over the existing target by synchronizing the target in place.

Note that the above operator and synchronizer assume the situation shown in Figure 2, where  $S_S$  and  $T_T$  do not have to be consistent. However, in cases where  $S_S$  and  $T_T$  are consistent and a small source update  $U_{\Delta\mathcal{S}}$  corresponds to a small target update  $Y_{\Delta\mathcal{T}}$ , the performance savings from reusing  $T_T$  in the computation of  $T'_T$  are expected to be high.

**Update translation.** The second incremental synchronizer assumes that the original source  $S_S$  and the original target  $T_T$  are consistent (cf. Figure 5). The key idea behind this synchronizer is to translate the update of the source into a consistent update of the target.

**Definition 3.** CONSISTENT UPDATES. *Two updates  $U_{\Delta\mathcal{S}}$  and  $Y_{\Delta\mathcal{T}}$  of two consistent reference artifacts  $S_S$  and  $T_T$ , respectively, are consistent iff application of both updates results in consistent artifacts, i.e.,  $(U_{\Delta\mathcal{S}}(S_S), Y_{\Delta\mathcal{T}}(T_T)) \in R$ .*

We can now define the *update translation* operator. The operator takes not only the update of the source artifact but also the original source and target artifacts as parameters. The reason is that consistent updates are defined with respect to these artifacts.

**Operator 4.** *Update translation:*  $UT_{S,T} : \Delta\mathcal{S} \times \mathcal{S} \times \mathcal{T} \rightarrow \Delta\mathcal{T}$ . For consistent artifacts  $S_S$  and  $T_T$ , i.e.,  $(S_S, T_T) \in R$ , and an update  $U_{\Delta\mathcal{S}}$  of the source artifact  $S_S$ , the operator  $UT_{S,T}(U_{\Delta\mathcal{S}}, S_S, T_T)$  computes an update  $U_{\Delta\mathcal{T}}$  of the target artifact  $T_T$  such that  $U_{\Delta\mathcal{S}}$  and  $U_{\Delta\mathcal{T}}$  are consistent for  $S_S$  and  $T_T$ , i.e.,  $(U_{\Delta\mathcal{S}}(S_S), U_{\Delta\mathcal{T}}(T_T)) \in R$ .

Using the update translation operator we can define the second incremental synchronizer as follows.

**Synchronizer 3.** *Unidirectional, incremental, original-target-independent, and to-one synchronizer using update translation:*

$$\mathbb{S}_{S,T} : \mathcal{S} \times \Delta\mathcal{S} \times \mathcal{T} \rightarrow \mathcal{T}$$

$$\begin{array}{l} S_S, U_{\Delta S}, T_T \\ (S_S, T_T) \in R \end{array} \implies \begin{array}{l} U_{\Delta T} = UT_{S,T}(U_{\Delta S}, S_S, T_T) \\ T'_T = U_{\Delta T}(T_T) \end{array} \implies T'_T$$

The synchronizer requires the original source and the original target, which have to be consistent, and an update to the original source.

Note that the update of the source artifact  $U_{\Delta S}$  can also be computed by comparing the new source against the original source using the homogeneous artifact comparison. This possibility allows us to rewrite Synchronizer 3 as follows.

**Synchronizer 4.** *Unidirectional, incremental, original-target-independent, and to-one synchronizer using homogeneous artifact comparison and update translation:*

$$\mathbb{A}_{S,T} : \mathcal{S} \times \mathcal{S} \times \mathcal{T} \rightarrow \mathcal{T}$$

$$\begin{array}{l} S_S, S'_S, T_T \\ (S_S, T_T) \in R \end{array} \implies \begin{array}{l} U_{\Delta S} = AC_S(S_S, S'_S) \\ U_{\Delta T} = UT_{S,T}(U_{\Delta S}, S_S, T_T) \\ T'_T = U_{\Delta T}(T_T) \end{array} \implies T'_T$$

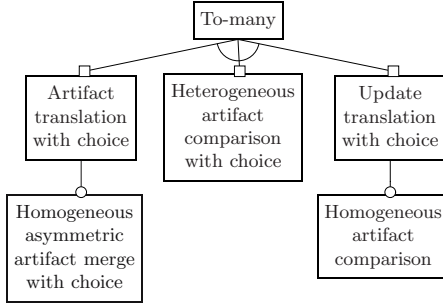
### An example for Synchronizer 3

*Example 10.* Live Update.

An example implementation of Synchronizer 3 is *live update* [18]. In live update, a target artifact is first obtained by executing a transformation on the source artifact. The transformation execution context is preserved and later used for incremental update of the target artifact in response to an update of the source artifact. The update translation operator works by locating the points in the transformation execution context that are affected by the source update. Update application works by resuming the transformation from the identified points with the new values from the source.

## 4.2 Unidirectional Synchronizers in To-Many Direction

The operators used in unidirectional to-many synchronizers are summarized in Figure 6. The feature diagram is similar to the diagram for the to-one case in Figure 4 except that each operator appears as a “with choice” variant. Furthermore, an additional variant using a special merge operator was added (on the bottom left in the diagram). The to-many case implies that a given source artifact may correspond to multiple target artifacts. Thus, each translating operator



**Fig. 6.** Operators used in to-many unidirectional synchronizers

in its “with choice” variant produces a set of possible targets rather than a single target. Consequently, all to-many synchronizers need a decision function as an additional input that they use to select only one result from the set of possible targets.

Like their to-one counterparts, the to-many synchronizers can be non-incremental or incremental. However, whereas all to-one synchronizers are original-target-independent, the to-many synchronizers have only one original-target-independent variant. The remaining ones are *original-target-dependent*, which means that values and structures from the original target are used in the computation of the new target and the resulting new target depends both on the new source and the original target.

The dependency on the original target is desirable for to-many synchronizers if the target can be edited by developers. The original-target-dependent synchronizers can preserve parts of the original target that have no representation in the source artifact type when the target is updated. These parts could be added to the target and edited by developers. Such edits should be preserved during the synchronization of the target in order to preserve developers’ work.

The first two unidirectional to-many synchronizers are non-incremental and correspond to the left branch of the feature diagram in Figure 6. The first one is original-target-independent. It uses *artifact translation with choice* to translate the new source into a set of possible new targets and selects one target using a decision function. The second synchronizer is original-target-dependent. It also uses *artifact translation with choice* to translate the new source into a set of possible new targets, but then it merges the selected new target with the original target. For this purpose, it uses *homogeneous asymmetric artifact merge with choice*, an operation which merges a slave artifact with a master artifact while preserving a certain property of the master artifact. As a result, some structures from the original target can be preserved.

The remaining synchronizers are incremental and operate similarly to their to-one counterparts. However, unlike the latter, they are original-target-dependent. The first incremental variant uses *heterogeneous artifact comparison with choice*. The other one uses *update translation with choice*. The source update

may optionally be computed using *homogeneous artifact comparison* between the original source and the new source.

**Artifact translation with choice.** Let us first consider the first non-incremental variant. This variant requires an artifact translation operator that returns a set of possible results. Note that  $\mathcal{P}^+(\mathcal{T})$  denotes the power set of the set  $\mathcal{T}$  without the empty set. We mark all “with choice” variants of operators with  $*$ .

**Operator 5.** *Artifact translation with choice:*  $AT_{\mathcal{S},\mathcal{T}}^* : \mathcal{S} \rightarrow \mathcal{P}^+(\mathcal{T})$ . For an artifact  $S'_\mathcal{S}$ , the operator  $AT_{\mathcal{S},\mathcal{T}}^*(S'_\mathcal{S})$  computes  $\{S'_\mathcal{T} : (S'_\mathcal{S}, S'_\mathcal{T}) \in R\}$ .

A single resulting artifact can be chosen using a *decision* function.

**Definition 4.** *DECISION.* A decision for an artifact type  $\mathcal{T}$  is a function  $D_{\mathcal{D}_\mathcal{T}} : \mathcal{P}^+(\mathcal{T}) \rightarrow \mathcal{T}$  such that  $\forall X \in \mathcal{P}^+(\mathcal{T}) : D_{\mathcal{D}_\mathcal{T}}(X) \in X$ . We denote a set of all decision functions for an artifact type  $\mathcal{T}$  as  $\mathcal{D}_\mathcal{T}$ .

Intuitively, a decision function chooses one artifact out of a set of artifacts of a given type. It models both the situation where the user makes a choice interactively or the situation where a choice is made based on some predefined criteria or default settings. We discuss some design choices for implementing decision functions in Section 8.5.

**Synchronizer 5.** *Unidirectional, non-incremental, original-target-independent, and to-many synchronizer using artifact translation with choice:*

$$\mathfrak{S}_{\mathcal{S},\mathcal{T}} : \mathcal{S} \times \mathcal{D}_\mathcal{T} \rightarrow \mathcal{T}$$

$$S'_\mathcal{S}, D_{\mathcal{D}_\mathcal{T}} \implies T'_\mathcal{T} = D_{\mathcal{D}_\mathcal{T}}(AT_{\mathcal{S},\mathcal{T}}^*(S'_\mathcal{S})) \implies T'_\mathcal{T}$$

Synchronizer 5 is only of interest for scenarios where the target artifact is not supposed to be manually edited, e.g., code generation in model compilation.

### Examples for Synchronizer 5

*Example 11.* Code and model compilation.

In compilation, the resulting artifacts, regardless if they are machine code, byte code, or code in a high-level programming language, depend on many settings of the compiler such as optimizations or coding style. Although the relation between the source and target artifacts is many-to-many, the selection of options allows the synchronizer (the compiler) to produce a single result.

*Example 12.* Pretty printing.

Similarly to the previous example, many code style options influence the result of pretty printing an abstract syntax tree representing a program.

**Homogeneous asymmetric artifact merge.** Unlike the first variant, which completely replaces the original target with the new one, the second non-incremental variant uses a merge operator to preserve some structures from the original target.

The merge operator is homogeneous as it merges two artifacts of the same type. It is also asymmetric as one of the artifacts is a *master artifact* and the other one is a *slave artifact*, that is, the operator merges the master and slave artifacts in such a way that the result of the merge satisfies the same property as the master artifact does. The merge can be implemented in two ways: by copying structures from the master artifact to the slave artifact or vice versa.

In our context, the slave artifact will be the original target and the master artifact will be the target obtained by translating the new source into the target artifact type. The property of the master artifact to be preserved will be its consistency with the new source artifact.

We model artifact properties as binary functions.

**Definition 5.** ARTIFACT PROPERTY. *A property function  $\phi$  for artifacts of type  $\mathcal{T}$  is a function with the following signature  $\phi : \mathcal{T} \rightarrow \{0, 1\}$ . We say that the property  $\phi$  holds for an artifact  $T_{\mathcal{T}}$  iff  $\phi(T_{\mathcal{T}}) = 1$ . We denote the set of all properties for an artifact type  $\mathcal{T}$  as  $\Phi_{\mathcal{T}}$ .*

**Operator 6.** *Homogeneous asymmetric artifact merge with choice:  $M_{\mathcal{T}}^* : \mathcal{T} \times \mathcal{T} \times \Phi_{\mathcal{T}} \rightarrow \mathcal{P}^+(\mathcal{T})$ . For a slave artifact  $T_{\mathcal{T}}$ , a property  $\phi$ , and a master artifact  $S_{\mathcal{T}}$  such that  $\phi(S_{\mathcal{T}}) = 1$ , the operator  $M_{\mathcal{T}}^*(T_{\mathcal{T}}, S_{\mathcal{T}}, \phi)$  computes a non-empty subset of  $\{T'_{\mathcal{T}} : \phi(T'_{\mathcal{T}}) = 1\}$ . The elements of the subset preserve structures from both master and slave artifacts according to some criteria.*

The key intention behind this operator, which is only partially captured by the formal part, is that the resulting set contains artifacts obtained by combining structures from both input artifacts such that each of the artifacts in the resulting set satisfies the input property. The merge returns a subset of all the artifacts satisfying the property, meaning that some artifacts satisfying the property are rejected if they do not preserve structures from both artifacts well enough according to some criteria. The operator returns a set of artifacts rather than a single artifact since, in general, there may be more than one satisfactory way to merge the input artifacts.

**Synchronizer 6.** *Unidirectional, non-incremental, original-target-dependent, and to-many synchronizer using artifact translation with choice and homogeneous asymmetric artifact merge with choice:*

**S6** $_{S, \mathcal{T}} : \mathcal{S} \times \mathcal{T} \times \mathcal{D}_{\mathcal{T}} \times \mathcal{D}_{\mathcal{T}} \rightarrow \mathcal{T}$

$$S'_S, T_{\mathcal{T}}, D_{\mathcal{D}_{\mathcal{T}}}, E_{\mathcal{D}_{\mathcal{T}}} \implies \begin{aligned} S'_{\mathcal{T}} &= D_{\mathcal{D}_{\mathcal{T}}}(AT_{S, \mathcal{T}}^*(S'_S)) \\ T'_{\mathcal{T}} &= E_{\mathcal{D}_{\mathcal{T}}}(M_{\mathcal{T}}^*(T_{\mathcal{T}}, S'_{\mathcal{T}}, \phi_{\Phi_{\mathcal{T}}})) \implies T'_{\mathcal{T}} \end{aligned}$$

$$\text{where } \phi_{\Phi_{\mathcal{T}}}(T) = \begin{cases} 1 & \text{if } (S'_S, T) \in R \\ 0 & \text{otherwise} \end{cases}$$

The synchronizer takes two decision functions. The first function selects a translation of the new source artifact into the target artifact type from the alternatives returned by the artifact translation with choice. The selected translation  $S'_{\mathcal{T}}$  is

then merged with the original target artifact, where the property passed to the merge is consistency with the new source artifact  $S'_S$ . The second decision function is used to select one target artifact from the alternatives returned by the merge.

In practice, the decision functions are likely to be realized as default settings allowing the entire synchronizer to be executed automatically. Furthermore, practical implementations, while focusing on preserving manual edits from the original target, often do not restore the full consistency during the merge. In such cases, the developers are expected to complete the merge by manual editing.

## An example for Synchronizer [6](#)

*Example 13.* JET and JMerge.

An example implementation of artifact merge with choice is *JMerge*, which is a part of Java Emitter Templates (JET) [\[19\]](#). JET is a template-based code generation framework in Eclipse.

JMerge can be used to merge an old version of Java code (slave artifact) with a new version (master artifact), such that developers can control which parts of the old versions get overridden by the corresponding parts from the new version. JMerge replaces Java classes, methods, and fields of the slave artifact that are annotated with `@generated` with their corresponding new versions from the master artifact. Developers can remove the `@generated` annotation from the elements they modify in order to preserve their modifications during subsequent merges. The behavior of JMerge is parameterized with a set of rules, which is an implementation of the decision function  $E_{\mathcal{D}_T}$ . JMerge is not concerned with preserving the consistency of the master artifact with the new source, meaning that the merged result might require manual edits in order to make it consistent. However, JMerge guarantees that all program elements in the slave that are not annotated with the `@generated` annotation remain unchanged in the merged result.

The code generator of Eclipse Modeling Framework (EMF) [\[20\]](#) implements Synchronizer [6](#) and uses JMerge as an implementation of the merge operator. The code generator is based on JET and takes a new EMF model as an input, which is the new source artifact  $S'_S$ . Code generation is controlled by a separate *generator model*, which specifies both global generation options and options that are specific to some source model elements. The latter can be thought of as decorations or mark-up of the source elements, but ones that are stored in a separate artifact. Effectively, the generator model corresponds to the decision function  $D_{\mathcal{D}_T}$ . The code generator emits the Java code implementing the model, i.e.,  $S'_T$ . JMerge is then used to merge the freshly-generated code  $S'_T$  (master artifact) with the original Java code  $T_T$  (slave artifact) that may contain developer's customizations. The resulting new Java code  $T'_T$  is now synchronized with the new model in the sense that all code elements annotated with the `@generated` annotation were replaced with the code elements generated from the new model.

The JMerge approach is an example of the concept of *protected blocks*. Protected blocks are specially marked code sections that are preserved during code re-generation. In JMerge, protected blocks are marked by virtue of not being annotated with `@generated`.

**Heterogeneous artifact comparison.** Analogously to the incremental synchronizers from the previous section, incremental to-many synchronizers can be realized using either heterogeneous comparison or update translation. However, both operators need to be modified to produce sets of results.

**Operator 7.** *Heterogeneous artifact comparison with choice:*  $AC_{\mathcal{S},\mathcal{T}}^* : \mathcal{S} \times \mathcal{T} \rightarrow \mathcal{P}^+(\Delta\mathcal{T})$ . For artifacts  $S'_S$  and  $T_T$ , the operator  $AC_{\mathcal{S},\mathcal{T}}^*(S'_S, T_T)$  computes a non-empty subset of  $\{U_{\Delta\mathcal{T}} : (S'_S, U_{\Delta\mathcal{T}}(T_T)) \in R\}$ . The elements of the subset preserve structures from  $T_T$  according to some criteria.

We can now state the first incremental synchronizer as follows.

**Synchronizer 7.** *Unidirectional, incremental, original-target-dependent, and to-many synchronizer using heterogeneous artifact comparison with choice:*

$$\mathbb{S}_{\mathcal{S},\mathcal{T}} : \mathcal{S} \times \mathcal{T} \times \mathcal{D}_{\Delta\mathcal{T}} \rightarrow \mathcal{T}$$

$$S'_S, T_T, D_{\mathcal{D}_{\Delta\mathcal{T}}} \implies \begin{array}{l} U_{\Delta\mathcal{T}} = D_{\mathcal{D}_{\Delta\mathcal{T}}}(AC_{\mathcal{S},\mathcal{T}}^*(S'_S, T_T)) \\ T'_T = U_{\Delta\mathcal{T}}(T_T) \end{array} \implies T'_T$$

### An example for Synchronizer 7

*Example 14.* Lenses in Harmony.

Synchronizer 7 corresponds to the *putback* function in Lenses [17]. In Lenses, the source-to-target relationship is many-to-one and *putback* is used in the target-to-source direction. In other words, *putback* is a unidirectional *to-many* synchronizer. The function takes the new view and the original source and returns the new source. A full lens combines *putback* with *get* (cf. Example 9) to form a bidirectional synchronizer (cf. Example 18).

**Update translation with choice.** The second incremental variant uses update translation with choice.

**Operator 8.** *Update translation with choice:*  $UT_{\mathcal{S},\mathcal{T}}^* : \Delta\mathcal{S} \times \mathcal{S} \times \mathcal{T} \rightarrow \mathcal{P}^+(\Delta\mathcal{T})$ . For two consistent artifacts  $S_S$  and  $T_T$  and an update  $U_{\Delta\mathcal{S}}$  of  $S_S$ , the operator  $UT_{\mathcal{S},\mathcal{T}}^*(U_{\Delta\mathcal{S}}, S_S, T_T)$  computes a non-empty subset of  $\{U_{\Delta\mathcal{T}} : (U_{\Delta\mathcal{S}}(S_S), U_{\Delta\mathcal{T}}(T_T)) \in R\}$ . The elements of the subset preserve structures from  $T_T$  according to some criteria.

**Synchronizer 8.** *Unidirectional, incremental, original-target-dependent, and to-many synchronizer using update translation with choice:*

$$\mathbb{S}_{\mathcal{S},\mathcal{T}} : \mathcal{S} \times \Delta\mathcal{S} \times \mathcal{T} \times \mathcal{D}_{\Delta\mathcal{T}} \rightarrow \mathcal{T}$$

$$(S_S, U_{\Delta\mathcal{S}}, T_T, D_{\mathcal{D}_{\Delta\mathcal{T}}}) \in R \implies \begin{array}{l} U_{\Delta\mathcal{T}} = D_{\mathcal{D}_{\Delta\mathcal{T}}}(UT_{\mathcal{S},\mathcal{T}}^*(U_{\Delta\mathcal{S}}, S_S, T_T)) \\ T'_T = U_{\Delta\mathcal{T}}(T_T) \end{array} \implies T'_T$$



## Examples for Synchronizer [8](#)

*Example 15.* Incremental code update in FSMs.

An example of Synchronizer [8](#) is incremental code update in FSMs [10](#). During forward propagation of model updates to code, code update transformations are executed for every added, modified, or removed model element. This translation of element updates into corresponding code updates is an example of an update translation function. Different code updates can be applied for a given model update depending on the desired implementation variant. An example of an implementation variant is the creation of an assignment to a field either as a separate statement or as an expression of the field’s initializer. The variants can be selected based on source model annotations that are provided by default and can also be modified by the developer. This annotation mechanism represents an implementation of the decision function  $D_{\mathcal{D}\Delta\mathcal{T}}$ .

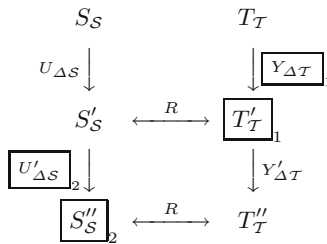
*Example 16.* Co-evolution of models with metamodels.

Wachsmuth [13](#) describes an approach to the synchronization of models in response to certain well-defined kinds of updates in their metamodels. The updates are classified into *refactoring*, *construction*, and *destruction*. These metamodel updates are then translated into the corresponding updates of the models. The model updates are an example of updates whose sets of reference artifacts contain more than one artifact (cf. Definition [2](#)).

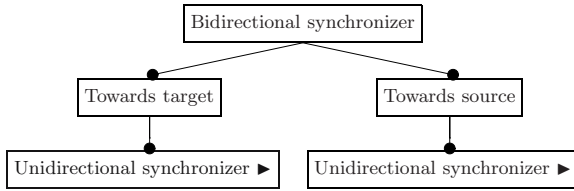
## 5 Bidirectional Synchronizers

Propagating change only in one direction is often not practical as certain changes may only be possible in certain artifacts. *Bidirectional synchronization* involves propagating changes in both directions using bidirectional synchronizers. Bidirectional synchronization is also referred to as *round-trip engineering* [21,22,23](#).

In this section, we focus on synchronization where changes to one artifact are propagated to the other artifact only in one direction at a time, whereas in the next section we focus on synchronization in which changes to both artifacts can be reconciled and propagated in both directions at once.



**Fig. 7.** Bidirectional synchronization scenario with a source-to-target synchronization followed by a target-to-source synchronization



**Fig. 8.** Bidirectional synchronizer

A sample bidirectional synchronization scenario with a source-to-target synchronization followed by a target-to-source synchronization is shown in Figure 7. The results of the first synchronization are placed in boxes with subscript one and the results of the second synchronization are placed in boxes with subscript two. The first synchronization is executed in response to update  $U_{\Delta S}$ , and the second synchronization is executed in response to update  $Y'_{\Delta T}$ .

A bidirectional synchronizer can be thought of as a pair of unidirectional synchronizers, one synchronizer for one direction, as shown in Figure 8. The feature *towards target* represents the unidirectional synchronizer from source to target, and the feature *towards source* represents the unidirectional synchronizer from target to source. Both synchronizers could be constructed separately using a unidirectional language, or they could be derived from a single description in a bidirectional language. We discuss these possibilities in Section 8.6.

**Properties.** According to Stevens [24], the key property of a pair of unidirectional synchronizers implementing bidirectional synchronization for a given relation is that they are *correct* with respect to the relation. Correctness means that each synchronizer enforces the relation between the source and target artifacts. Clearly, any pair  $(S_{i_{S,T}}, S_{j_{T,S}})$  of the unidirectional synchronizers defined in the previous sections (where  $i$  and  $j$  may be equal) is correct with respect to  $R$  by the definition of the synchronizers.

Another desired property of a synchronization synchronizer is *hippocraticness* [24], meaning that the synchronizer should not modify any of the artifacts if they already are in the relation. The *hippocraticness* property is also referred to as *check-then-enforce*, which suggests that the synchronizer should only enforce the relation if the artifacts are not in the relation.

Note that, in practice, a synchronization step may be partial in the sense that it does not establish full consistency. Artifact developers may choose to synchronize only certain changes at a time and ignore parts of the artifacts that are not yet ready to be synchronized. Therefore, the correctness property only applies to complete synchronization.

## Examples of bidirectional synchronizers

*Example 17.* Triple Graph Grammars in FUJABA.

Giese and Wagner describe an approach to bidirectional synchronization using Triple Graph Grammars (TGG) [25]. Their approach is implemented in the

Fujaba tool suite [26]. TGG rules are expressed using a bi-directional, graphical language. For two models, the user can choose the direction of synchronization. Both models are then matched by TGG rules, which can be viewed as an implementation of the heterogeneous artifact comparison. The updates determined by each rule are applied to the target in a given direction, which amounts to incremental synchronization. The authors assume that the relationship between source and target is a bijection [25, p. 550]. Thus, the approach can be described as  $(\mathcal{S}_{S,T}, \mathcal{S}_{T,S})$ .

*Example 18.* Lenses in Harmony.

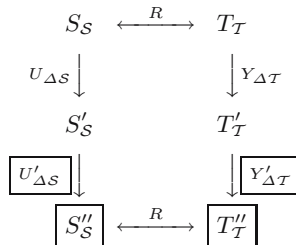
A *lens* [17] is a bidirectional synchronizer for the many-to-one case. It consists of two unidirectional synchronizers: *get* (cf. Example 9) and *putback* (cf. Example 14). In other words, a lens can be described as  $(\mathcal{S}_{S,T}, \mathcal{T}_{T,S})$ . Note that the second synchronizer executes in the target-to-source direction, i.e., the direction towards the end with the cardinality of many, and the artifact at that end can be edited. Consequently, the synchronizer should be one of the unidirectional, to-many, and original-target-dependent synchronizers, which is satisfied by  $\mathcal{T}_{T,S}$ .

## 6 Bidirectional Synchronizers with Reconciliation

In this section we focus on synchronization where both artifacts can be changed simultaneously in-between two consecutive synchronizations and the changes can be reconciled and propagated in both directions during a single synchronization.

Bidirectional synchronization with reconciliation involves up to six artifacts (cf. Figure 9). Four of them are the same as in the case of unidirectional synchronization (cf. Figure 2), except that the original source  $S_S$  and the original target  $T_T$  are now assumed to be consistent. Furthermore, the new target  $T'_T$  is given as a result of a user update  $Y_{\Delta T}$  just as the new source  $S'_S$  is given as a result of another user update  $U_{\Delta S}$ . The purpose of a bidirectional synchronizer with reconciliation is to compute a *reconciled source artifact*  $S''_S$  and a *reconciled target artifact*  $T''_T$ , such that the two are consistent. In essence, such a synchronizer can also be viewed as a *heterogeneous symmetric merge operation*.

As in the unidirectional case, some of the four input artifacts may be missing. The two extreme cases are when only the new source or only the new target



**Fig. 9.** Artifacts involved in bidirectional synchronization with reconciliation

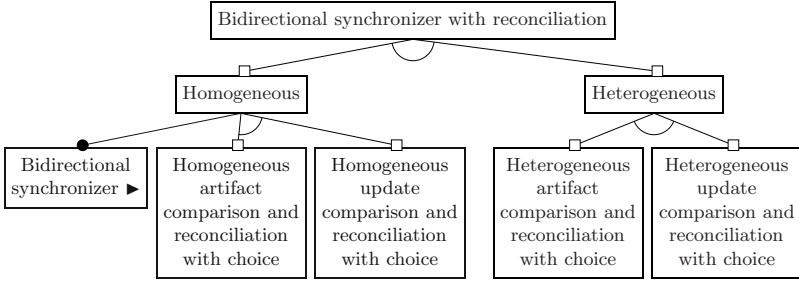


Fig. 10. Bidirectional synchronizer with reconciliation

exist. The synchronization in these cases corresponds to the initial generation of the target artifact or the source artifact, respectively. The case where both original artifacts are missing corresponds to the situation where two artifacts are synchronized for the first time. Note that a “missing” artifact corresponds to a special value that represents a *minimal* artifact, that is, an artifact that contains the minimum structure required by its artifact type. We assume that minimal artifacts of all types are always consistent.

In general, bidirectional synchronization with reconciliation involves

- Translation of updates, artifacts, or both;
- Identification of conflicting updates;
- Creation of updates that resolve conflicts and reconcile the artifacts; and
- Application of the updates.

The identification of conflicting updates and their resolution can be performed in *homogeneous* or *heterogeneous* fashion as indicated in Figure 10.

Homogeneous reconciliation means that updates to both source and target artifacts are compared and then reviewed by the user in terms of one artifact type, which is either the source or the target type. In other words, if the comparison and review (and resolution of potential conflicts) is done on the target side, the new source artifact or the update of the source artifact need to be first translated into the target type. Depending whether the entire artifact or just the update is translated, the comparison and reconciliation is done either by *homogeneous artifact comparison and reconciliation with choice* or its *update* counterpart (cf. Figure 10). Assuming reconciliation on the target side, both operators return an update of the new source artifact (but expressed in the target artifact type) and an update to the new target artifact, such that the two updates reconcile both artifacts. Finally, the first update has to be translated back into the source artifact type and applied to the new source artifact, and the second update is applied to the new target artifact. Note that the translation of artifacts or updates in one direction and the translation of updates in the other direction essentially requires a bidirectional synchronizer, as indicated in Figure 10 by a reference to the feature *bidirectional synchronizer*.

Heterogeneous reconciliation implies a heterogeneous comparison between the artifacts or the updates. A bidirectional synchronizer with heterogeneous

reconciliation can be implemented using the operator *heterogeneous artifact comparison and reconciliation with choice* or its *update* counterpart (cf. Figure 10). The operators are similar to their homogeneous counterparts with the difference that they directly compare artifacts of different types and thus do not require a pair of unidirectional synchronizers for both directions.

### 6.1 Comparison and Reconciliation Procedures

In general, comparison and reconciliation operators work at the level of individual structural updates that occurred within the overall update of the source artifact  $U_{\Delta S}$  and the overall update of the target artifact  $Y_{\Delta T}$ . The updates can be *atomic*, such as element additions, removals, and relocations and attribute value modifications. The updates can also be *composite*, i.e., consisting of other atomic and composite updates.

We categorize updates into *synchronizing*, *propagating*, *consistent*, *conflicting*, *non-reflectable*, and *inverse*. An update in one artifact is *synchronizing* if it establishes the consistency of the artifact with the related artifact. An update in one artifact is *propagating* if it forces a synchronizing update in the related artifact. Two updates, one in each artifact, are *consistent* if one is a synchronizing update of the other one. On the other hand, two updates, one in each artifact, are *conflicting* if the propagation of one update would override the other one. An update in one artifact is *non-reflectable* if it does not force any synchronizing update in the other artifact. An *inverse* update (intuitively *undo*) for a given update and a reference artifact maps the result of applying the given update to the reference artifact back to the reference artifact.

A *maximal* synchronizer [27] is one that propagates *all* propagating updates. The following strategy is used to compute  $U'_{\Delta S}$  and  $Y'_{\Delta T}$  for achieving maximum synchronization:

- Consistent and non-reflectable updates in  $U_{\Delta S}$  and  $Y_{\Delta T}$  are ignored since both artifacts are already consistent with respect to these updates;
- Out of several conflicting updates in  $U_{\Delta S}$  and  $Y_{\Delta T}$ , exactly one update can be accepted as a propagating update; and
- For each propagating update in  $U_{\Delta S}$ , a synchronizing update needs to be included in  $Y'_{\Delta T}$ ; similarly, for each propagating update in  $Y_{\Delta T}$ , a synchronizing update needs to be included in  $U'_{\Delta S}$ .

After the updates are classified into consistent, non-reflectable, conflicting, and propagating by the comparison operator, the user typically reviews the classification, resolves conflicts by rejecting some of the conflicting updates, and then the final updates  $U'_{\Delta S}$  and  $Y'_{\Delta T}$  are computed by determining and composing the necessary synchronizing updates. In practice, simple acceptance or rejection of updates might not be sufficient to resolve all conflicts, in which case the input artifacts may need to be manually edited to resolve and merge conflicting updates.

In general, conflict resolution is not the only possible conflict management strategy. Other possibilities include storing all conflicting updates in each reconciled artifact or allowing artifacts to diverge for conflicting updates [27].

## 6.2 Bidirectional Synchronizers for One-to-One Relations

Note that due to the need for reconciliation, none of the synchronizers can be fully non-incremental since at least one artifact needs to be updated by update application. Let us first consider a target-incremental synchronizer. This variant requires *homogeneous artifact comparison and reconciliation with choice* operation. The need for choice arises from the fact that conflicts may be resolved in different ways.

**Operator 9.** *Homogeneous artifact comparison and reconciliation with choice:*  $ACR_T^* : \mathcal{T} \times \mathcal{T} \times \mathcal{T} \rightarrow \mathcal{P}^+(\Delta\mathcal{T} \times \Delta\mathcal{T})$ . For two artifacts  $S'_T$  and  $T'_T$ , and the reference artifact  $T_T$ , the operator  $ACR_T^*(S'_T, T'_T, T_T)$  computes a non-empty subset of  $\{(U'_{\Delta\mathcal{T}}, Y'_{\Delta\mathcal{T}}) : U'_{\Delta\mathcal{T}}(S'_T) = Y'_{\Delta\mathcal{T}}(T'_T)\}$ . Each pair of updates  $(U'_{\Delta\mathcal{T}}, Y'_{\Delta\mathcal{T}})$  from that subset is such that the updates resolve conflicting changes and enforce all propagating changes from  $U_{\Delta\mathcal{T}}$  and  $Y_{\Delta\mathcal{T}}$ , where  $U_{\Delta\mathcal{T}} = AC_T(T_T, S'_T)$  and  $Y_{\Delta\mathcal{T}} = AC_T(T_T, T'_T)$ .

The operator  $ACR_T^*$  performs a three-way comparison of the artifacts and returns a set of pairs of updates. The reference artifact is included in the three-way comparison as it allows precisely determining the kind and location of updates. In particular, it allows determining whether certain updates occurred consistently in both artifacts, inconsistently in both artifacts, or only in one artifact. Each resulting pair of updates modifies both artifacts  $S'_T$  and  $T'_T$  such that they become identical and all conflicting updates are resolved and all propagating updates are propagated. The second condition is necessary: without it, the operator could simply return updates that could, for example, revert each artifact back to the reference artifact, or even to the minimal artifact. Each pair of resulting updates represents one possible way of reconciling conflicts. The resulting updates are constructed using the strategy given at the end of the previous section.

Now we are ready to formulate the target-incremental synchronizer. Note that all discussed synchronizers perform reconciliation on the target side.

**Synchronizer 9.** *Bidirectional, target-incremental, and one-to-one synchronizer using artifact translation and homogeneous artifact comparison and reconciliation with choice:*

$$\mathfrak{S}_{\mathcal{S}, \mathcal{T}} : \mathcal{S} \times \mathcal{T} \times \mathcal{T} \times \mathcal{D}_{\Delta\mathcal{T} \times \Delta\mathcal{T}} \rightarrow \mathcal{S} \times \mathcal{T}$$

$$\begin{aligned} S'_S, T'_T, T_T, \\ F_{\mathcal{D}_{\Delta\mathcal{T} \times \Delta\mathcal{T}}} \implies & \begin{aligned} S'_T &= AT_{\mathcal{S}, \mathcal{T}}(S'_S) \\ (-, Y'_{\Delta\mathcal{T}}) &= F_{\mathcal{D}_{\Delta\mathcal{T} \times \Delta\mathcal{T}}}(ACR_T^*(S'_T, T'_T, T_T)) \\ T''_T &= Y'_{\Delta\mathcal{T}}(T'_T) \\ S''_S &= AT_{\mathcal{T}, \mathcal{S}}(T''_T) \end{aligned} & \implies S''_S, T''_T \end{aligned}$$

In the target-incremental variant, source artifact is first translated into the target artifact type. Next, the operator  $ACR_T^*$  computes new updates for each artifact. In the target-incremental synchronizers, the update for the artifact  $S'_T$  is simply ignored. Next, the reconciled target artifact  $T''_T$  is created by applying the update

$Y'_{\Delta T}$  to  $T'_T$ . Finally, the reconciled source artifact  $S''_S$  is obtained by translating  $T''_T$  back into the artifact type  $\mathcal{S}$ .

A fully-incremental variant, in which the new source  $S'_S$  is incrementally updated, is also possible.

**Synchronizer 10.** *Bidirectional, fully-incremental, and one-to-one synchronizer using artifact translation, homogeneous artifact comparison and reconciliation with choice, and update translation:*

$$\boxed{ST}_{S,T} : \mathcal{S} \times \mathcal{T} \times \mathcal{T} \times \mathcal{D}_{\Delta T \times \Delta T} \rightarrow \mathcal{S} \times \mathcal{T}$$

$$\begin{aligned} & S'_S, T'_T, T_T, \\ & F_{\mathcal{D}_{\Delta T \times \Delta T}} \implies \begin{aligned} & S'_T = AT_{S,T}(S'_S) \\ & (U'_{\Delta T}, Y'_{\Delta T}) = F_{\mathcal{D}_{\Delta T \times \Delta T}}(ACR^*_T(S'_T, T'_T, T_T)) \\ & T''_T = Y'_{\Delta T}(T'_T) \\ & U'_{\Delta S} = UT_{T,S}(U'_{\Delta T}, S'_T, S'_S) \\ & S''_S = U'_{\Delta S}(S'_S) \end{aligned} \implies S''_S, T''_T \end{aligned}$$

A fully-incremental variant can also be realized by translating updates instead of translating the entire artifacts. The fully-incremental case requires a *homogeneous update comparison and reconciliation* operator.

**Operator 10.** *Homogeneous update comparison and reconciliation with choice:*  $UCR^*_T : \Delta T \times \Delta T \times \mathcal{T} \rightarrow \mathcal{P}^+(\Delta T \times \Delta T)$ . For two updates  $U_{\Delta T}$  and  $Y_{\Delta T}$  of a reference artifact  $T_T$ , the operator  $UCR^*_T(U_{\Delta T}, Y_{\Delta T}, T_T)$  computes a non-empty subset of  $\{(U'_{\Delta T}, Y'_{\Delta T}) : U'_{\Delta T}(U_{\Delta T}(T_T)) = Y'_{\Delta T}(Y_{\Delta T}(T_T))\}$ . Each pair of updates  $(U'_{\Delta T}, Y'_{\Delta T})$  from that subset is such that the updates resolve all conflicting changes and enforce all propagating changes from  $U_{\Delta S}$  and  $Y_{\Delta T}$ .

**Synchronizer 11.** *Bidirectional, fully-incremental, and one-to-one synchronizer using update translation and homogeneous update comparison and reconciliation with choice:*

$$\boxed{ST}_{S,T} : \mathcal{S} \times \mathcal{S} \times \Delta S \times \mathcal{T} \times \mathcal{T} \times \Delta T \times \mathcal{D}_{\Delta T \times \Delta T} \rightarrow \mathcal{S} \times \mathcal{T}$$

$$\begin{aligned} & S_S, S'_S, U_{\Delta S}, \\ & T_T, T'_T, Y_{\Delta T}, \\ & F_{\mathcal{D}_{\Delta T \times \Delta T}} \\ & U_{\Delta S}(S_S) = S'_S \\ & Y_{\Delta T}(T_T) = T'_T \\ & (S_S, T_T) \in R \implies \begin{aligned} & U_{\Delta T} = UT_{S,T}(U_{\Delta S}, S_S, T_T) \\ & (U'_{\Delta T}, Y'_{\Delta T}) = F_{\mathcal{D}_{\Delta T \times \Delta T}}(UCR^*_T(U_{\Delta T}, Y_{\Delta T}, T_T)) \\ & T''_T = Y'_{\Delta T}(T'_T) \\ & U'_{\Delta S} = UT_{T,S}(U'_{\Delta T}, T'_T, S'_S) \\ & S''_S = U'_{\Delta S}(S'_S) \end{aligned} \implies S''_S, T''_T \end{aligned}$$

Analogously to the non-incremental variant, the  $UCR^*_T$  operator performs the three-way comparison of the updates with respect to the reference artifact  $T_T$ .

Again, the result is a pair of reconciled updates. The update  $U'_{\Delta T}$  needs to be translated into the artifact type  $\mathcal{S}$ . Finally, the reconciled updates are applied.

### 6.3 Bidirectional Synchronizers for Many-to-One Relations

For many-to-one relations, we only consider homogeneous reconciliation on the target side since source artifacts or updates can be unambiguously translated in the target direction. We show two synchronizers in this category. The first synchronizer uses a non-incremental unidirectional synchronizer in the source-to-target direction, while the other uses an incremental one. For the target-to-source direction, we need to use one of the unidirectional to-many synchronizers that are original-target-dependent, where the “original target” corresponds to the new source in our context. The reason is that we want to preserve non-reflectable edits from the new source. Both synchronizers use update translation with choice in the target-to-source direction.

**Synchronizer 12.** *Bidirectional, fully-incremental, and many-to-one synchronizer using artifact translation, homogeneous artifact comparison and reconciliation with choice, and update translation with choice:*

$$\mathbf{SI2}_{S,T} : \mathcal{S} \times \mathcal{S} \times \mathcal{T} \times \mathcal{T} \times \mathcal{D}_{\Delta S} \times \mathcal{D}_{\Delta T \times \Delta T} \rightarrow \mathcal{S} \times \mathcal{T}$$

$$S_S, S'_S, T_T, T'_T,$$

$$D_{\mathcal{D}_{\Delta S}},$$

$$F_{\mathcal{D}_{\Delta T \times \Delta T}}$$

$$(S_S, T_T) \in R \implies S'_T = AT_{S,T}(S'_S)$$

$$(U'_{\Delta T}, Y'_{\Delta T}) = F_{\mathcal{D}_{\Delta T \times \Delta T}}(ACR^*_T(S'_T, T'_T, T_T))$$

$$T''_T = Y'_{\Delta T}(T'_T)$$

$$U'_{\Delta S} = D_{\mathcal{D}_{\Delta S}}(UT^*_{T,S}(U'_{\Delta T}, S'_T, S'_S))$$

$$S''_S = U'_{\Delta S}(S'_S)$$

$$\implies S''_S, T''_T$$

**Synchronizer 13.** *Bidirectional, fully-incremental, and many-to-one synchronizer using update translation, homogeneous update comparison and reconciliation with choice, and update translation with choice:*

$$\mathbf{SI3}_{S,T} : \mathcal{S} \times \mathcal{S} \times \Delta \mathcal{S} \times \mathcal{T} \times \mathcal{T} \times \Delta \mathcal{T} \times \mathcal{D}_{\Delta S} \times \mathcal{D}_{\Delta T \times \Delta T} \rightarrow \mathcal{S} \times \mathcal{T}$$

$$S_S, S'_S, U_{\Delta S},$$

$$T_T, T'_T, Y_{\Delta T},$$

$$D_{\mathcal{D}_{\Delta S}},$$

$$F_{\mathcal{D}_{\Delta T \times \Delta T}}$$

$$U_{\Delta S}(S_S) = S'_S$$

$$Y_{\Delta T}(T_T) = T'_T$$

$$(S_S, T_T) \in R \implies U_{\Delta T} = UT_{S,T}(U_{\Delta S}, S_S, T_T)$$

$$(U'_{\Delta T}, Y'_{\Delta T}) = F_{\mathcal{D}_{\Delta T \times \Delta T}}(UCR^*_T(U_{\Delta T}, Y_{\Delta T}, T_T))$$

$$T''_T = Y'_{\Delta T}(T'_T)$$

$$U'_{\Delta S} = D_{\mathcal{D}_{\Delta S}}(UT^*_{T,S}(U'_{\Delta T}, T'_T, S'_S))$$

$$S''_S = U'_{\Delta S}(S'_S)$$

$$\implies S''_S, T''_T$$



## An example for Synchronizer 12

*Example 19.* Synchronization in FSMLs.

The FSML infrastructure 10 supports synchronization according to Synchronizer 12. Source artifact is Java code, XML code, or a combination of both. Target artifact is a model in an FSML designed for a particular framework, e.g., Apache Struts (cf. Example 3). The relation between source and target is many-to-one. The infrastructure performs homogeneous artifact comparison and reconciliation on the model (target) side since every code update has a unique representation on the model side. The reverse is not true: a model update can be translated in different ways into code updates.

The first step of the synchronizer is to retrieve  $S'_T$ , i.e., the *model of the new code*, from the new code  $S'_S$  using  $AT_{S,T}$ , which is implemented by a set of code queries (cf. Example 8).

The second step is a three-way compare between the model of the new code  $S'_T$  and the new model  $T'_T$  while using the original model  $T_T$  as a reference artifact. The original model corresponds to the initial situation when the model and the code were consistent after the previous synchronization, and the new model and the new code are the results of independent updates of the respective original artifacts (cf. Figure 9).

The artifact comparison and reconciliation  $ACR^*_T$  operates on framework-specific models. A model is an object structure conforming to a class model, i.e., the metamodel. The object structure consists of objects (i.e., *model elements*), attributes with primitive values, and containment and reference links between objects. The containment links form a containment hierarchy, which is a tree. The comparison process starts with establishing the correspondence among the model elements in all three models, namely  $S'_T$ ,  $T'_T$ , and  $T_T$ . The correspondence is established using structural matching, which takes into account the location of the elements in the containment hierarchy and their identification keys that are specified in the metamodel. Approaches to establishing correspondence are further discussed in Section 8.2. The result of the matching is a set of 3-tuples, where each tuple contains the corresponding elements from the three input models. Each position in a 3-tuple is dedicated to one of the three input models and contains the corresponding element from the model or a special symbol representing the absence of the corresponding element from that model.

The comparison process continues by processing each 3-tuple to establish the updates that occurred in the new source and the new target according to Table 2. The first and the second column classifies each 3-tuple according to whether all three elements or only some were present and whether the corresponding elements were equal or not. Two elements are equal iff their corresponding attribute values are equal, their corresponding reference links point to the same element, and the corresponding contained elements are equal. The third column describes the detected updates as element additions, modifications, and removals, and the fourth column classifies the updates as propagating, consistent, or conflicting (cf. Section 6.1).

**Table 2.** Results of three-way compare of the corresponding elements  $t$ ,  $s$ , and  $r$  in the new artifacts  $T'_T$  and  $S'_T$ , and the reference artifact  $T_T$ , respectively. The absence of a corresponding element is represented by -. Table adapted from [28].

$T'_T$	$S'_T$	$T_T$	condition	detected updates to element	update classification
s	t	r	$t = s = r$	unchanged	no updates
s	t	r	$t = s \wedge t \neq r$	modified consistently in $T'_T$ & $S'_T$	consistent updates
s	t	-	$t = s$	added consistently to $T'_T$ & $S'_T$	consistent updates
s	t	r	$t \neq s \wedge t = r$	modified in $S'_T$	propagating update in $S'_T$
s	t	r	$t \neq s \wedge s = r$	modified in $T'_T$	propagating update in $T'_T$
s	t	r	$t \neq s \neq r \neq t$	modified inconsistently in $T'_T$ & $S'_T$	conflicting updates
s	t	-	$t \neq s$	added inconsistently to $T'_T$ & $S'_T$	conflicting updates
s	-	r	$t = r$	removed from $S'_T$	propagating update in $S'_T$
s	-	r	$t \neq r$	removed from $S'_T$ , modified in $T'_T$	conflicting updates
s	-	-	-	added to $T'_T$	propagating update in $T'_T$
-	t	r	$s = r$	removed from $T'_T$	propagating update in $T'_T$
-	t	r	$s \neq r$	removed from $T'_T$ , modified in $S'_T$	conflicting updates
-	t	-	-	added to $S'_T$	propagating update in $S'_T$
-	-	r	-	removed from $T'_T$ & $S'_T$	consistent updates

The classification results are then presented to the user, who can review each of the updates and decide to accept or reject it. More precisely, a propagating update can be accepted or rejected and a pair of conflicting updates can be enforced in the forward or the reverse direction or rejected all together. Note that the decisions can be taken at different levels in the containment hierarchy. In an extreme, the user might only review the updates at the level of the corresponding model elements representing the model roots. The user might also desire to drill down the hierarchy and review the updates at a finer granularity.

The conflict resolution decisions made by the user correspond to the decision function  $F_{\mathcal{D}_{\Delta T} \times \Delta T}$ . It is desirable that the decisions taken by the user should result in a well-formed model  $T''_{\mathcal{T}}$  before the code is updated. However, in practice, developers may choose to synchronize one element at a time. Also, only accepting and/or rejecting updates may not be enough to arrive at the desired model, meaning that developers might need to perform some additional edits during reconciliation.

The last stage of  $ACR^*_{\mathcal{T}}$  is to compute the resulting updates  $U''_{\Delta T}$  and  $Y''_{\Delta T}$ . The resulting update  $U''_{\Delta T}$  for  $S'_T$  is computed by collecting the synchronizing update for every accepted propagating and conflicting update to  $T'_T$  and the inverse updates to the rejected propagating updates. An inverse update reverts an element back to its state from  $T_T$ . There is no need to include an inverse update for the rejected update from a conflicting pair since the accepted update will override the corresponding element. The update  $Y''_{\Delta T}$  is computed in a similar way.

Finally, the update of the model representing the new code,  $U''_{\Delta T}$ , is translated into the update of the new code,  $U''_{\Delta S}$ . The translation is achieved using update translation  $UT^*_{\mathcal{T}, S}$  as described in Example 15. At last, both the new code and the new model are incrementally updated by applying  $U''_{\Delta S}$  and  $Y''_{\Delta T}$ , respectively, and the synchronizer returns the two reconciled artifacts  $S''_S$  and  $T''_T$ .

## 6.4 Bidirectional Synchronizers for Many-to-Many Relations

Reconciliation for many-to-many relations can be performed in the homogeneous or heterogeneous fashion. A bidirectional synchronizer with homogeneous reconciliation for a many-to-many relation needs to use unidirectional original-target-dependent to-many synchronizers in both directions.

First we show a bidirectional synchronizer with homogeneous reconciliation that uses update translation with choice in both directions.

**Synchronizer 14.** *Bidirectional, fully-incremental, and many-to-many synchronizer using update translation with choice and homogeneous update comparison and reconciliation with choice:*

$$\mathbb{S}14_{S,T} : \mathcal{S} \times \mathcal{S} \times \Delta\mathcal{S} \times \mathcal{T} \times \mathcal{T} \times \Delta\mathcal{T} \times \mathcal{D}_{\Delta\mathcal{S}} \times \mathcal{D}_{\Delta\mathcal{T}} \times \mathcal{D}_{\Delta\mathcal{T} \times \Delta\mathcal{T}} \rightarrow \mathcal{S} \times \mathcal{T}$$

$$\begin{aligned} & S_S, S'_S, U_{\Delta\mathcal{S}}, \\ & T_T, T'_T, Y_{\Delta\mathcal{T}}, \\ & D_{\mathcal{D}_{\Delta\mathcal{S}}}, E_{\mathcal{D}_{\Delta\mathcal{T}}}, \\ & F_{\mathcal{D}_{\Delta\mathcal{T} \times \Delta\mathcal{T}}} \\ & U_{\Delta\mathcal{S}}(S_S) = S'_S \\ & Y_{\Delta\mathcal{T}}(T_T) = T'_T \\ (S_S, T_T) \in R \implies & \begin{aligned} & U_{\Delta\mathcal{T}} = E_{\mathcal{D}_{\Delta\mathcal{T}}}(UT_{S,T}^*(U_{\Delta\mathcal{S}}, S_S, T_T)) \\ & (U'_{\Delta\mathcal{T}}, Y'_{\Delta\mathcal{T}}) = F_{\mathcal{D}_{\Delta\mathcal{T} \times \Delta\mathcal{T}}}(UCR_{T,T}^*(U_{\Delta\mathcal{T}}, Y_{\Delta\mathcal{T}}, T_T)) \\ & T''_T = Y'_{\Delta\mathcal{T}}(T'_T) \\ & U'_{\Delta\mathcal{S}} = D_{\mathcal{D}_{\Delta\mathcal{S}}}(UT_{T,S}^*(U'_{\Delta\mathcal{T}}, S_S, T_T)) \\ & S''_S = U'_{\Delta\mathcal{S}}(S'_S) \end{aligned} \implies S''_S, T''_T \end{aligned}$$

The heterogeneous variant of the many-to-many synchronizer requires a heterogeneous comparison and reconciliation operator.

**Operator 11.** *Heterogeneous artifact comparison and reconciliation with choice:*  $ACR_{S,T}^* : \mathcal{S} \times \mathcal{T} \times \mathcal{S} \times \mathcal{T} \rightarrow \mathcal{P}^+(\Delta\mathcal{S} \times \Delta\mathcal{T})$ . For two artifacts  $S'_S$  and  $T'_T$ , and two consistent reference artifacts  $S_S$  and  $T_T$ , the operator  $ACR_{S,T}^*(S'_S, T'_T, S_S, T_T)$  computes a non-empty subset of  $\{(U'_{\Delta\mathcal{S}}, Y'_{\Delta\mathcal{T}}) : (U'_{\Delta\mathcal{S}}(S'_S), Y'_{\Delta\mathcal{T}}(T'_T)) \in R\}$ . Each pair of updates  $(U'_{\Delta\mathcal{T}}, Y'_{\Delta\mathcal{T}})$  from that subset is such that the updates resolve conflicting updates and enforce all propagating updates from  $U_{\Delta\mathcal{S}}$  and  $Y_{\Delta\mathcal{T}}$ , where  $U_{\Delta\mathcal{S}} = AC_S(S_S, S'_S)$  and  $Y_{\Delta\mathcal{T}} = AC_T(T_T, T'_T)$ .

**Synchronizer 15.** *Bidirectional, fully-incremental, and many-to-many synchronizer using heterogeneous artifact comparison and reconciliation with choice:*

$$\mathbb{S}15_{S,T} : \mathcal{S} \times \mathcal{S} \times \mathcal{T} \times \mathcal{T} \times \mathcal{D}_{\Delta\mathcal{S} \times \Delta\mathcal{T}} \rightarrow \mathcal{S} \times \mathcal{T}$$

$$\begin{aligned} & S_S, S'_S, T_T, T'_T, \\ & F_{\mathcal{D}_{\Delta\mathcal{S} \times \Delta\mathcal{T}}} \\ (S_S, T_T) \in R \implies & \begin{aligned} & (U'_{\Delta\mathcal{S}}, Y'_{\Delta\mathcal{T}}) = F_{\mathcal{D}_{\Delta\mathcal{S} \times \Delta\mathcal{T}}}(ACR_{S,T}^*(S'_S, T'_T, S_S, T_T)) \\ & S''_S = U'_{\Delta\mathcal{S}}(S'_S) \\ & T''_T = Y'_{\Delta\mathcal{T}}(T'_T) \end{aligned} \implies S''_S, T''_T \end{aligned}$$

We introduce the last variant, Synchronizer [16](#), by first discussing its sample implementation.

### An example for Synchronizer 16

*Example 20.* ATL Virtual Machine extension for synchronization.

An example of a bidirectional many-to-many synchronizer is an extension to the Atlas Transformation Language (ATL) [29] virtual machine [30]. While the synchronizer works in the reconciliation setting as illustrated in Figure 9 and allows independent updates to the original source and the original target, it only supports partial reconciliation. More specifically, while the synchronizer propagates all propagating updates, it does not support conflict resolution. Furthermore, the synchronizer does not tolerate additions made to the original target model. In any of the above situations, the synchronizer reports an error and terminates.

The mapping between source and target is given as an artifact translator expressed in ATL, which is a unidirectional transformation language. While an ATL translator is a partial function, the extension supports many-to-many relations by merging the translation results with existing artifacts using asymmetric homogeneous merge (cf. Operator 6). In this way, the non-reflectable updates from the new source and the new target can be preserved.

The following synchronizer describes the synchronization procedure.

**Synchronizer 16.** *Bidirectional, source-incremental, and many-to-many synchronizer using artifact translation, homogeneous artifact comparison, update translation with choice, and homogeneous asymmetric artifact merge with choice:*

$$\text{S16}_{S,T} : \mathcal{S} \times \mathcal{S} \times \mathcal{T} \times \mathcal{D}_{\Delta S} \times \mathcal{D}_{\Delta S} \times \mathcal{D}_{\Delta T} \rightarrow \mathcal{S} \times \mathcal{T}$$

$$\begin{array}{l} S_S, S'_S, T'_T, \\ D_{\mathcal{D}_S}, E_{\mathcal{D}_S}, F_{\mathcal{D}_T} \end{array} \implies \begin{array}{l} T_T = AT_{S,T}(S_S) \\ Y_{\Delta T} = AC_T(T_T, T'_T) \\ Y_{\Delta S} = D_{\mathcal{D}_S}(UT_{T,S}^*(Y_{\Delta T}, T_T, S_S)) \\ S_S^Y = Y_{\Delta S}(S_S) \\ S_S'' = E_{\mathcal{D}_S}(M_S^*(S'_S, S_S^Y, \phi_{\Phi_S}^1)) \\ S_T'' = AT_{S,T}(S_S'') \\ T_T'' = F_{\mathcal{D}_T}(M_T^*(T'_T, S_T'', \phi_{\Phi_T}^2)) \end{array} \implies S_S'', T_T''$$

$$\text{where } \phi_{\Phi_S}^1(S) = \begin{cases} 1 & \text{if } (T'_T, S) \in R \\ 0 & \text{otherwise} \end{cases}$$

$$\phi_{\Phi_T}^2(T) = \begin{cases} 1 & \text{if } (S_S'', T) \in R \\ 0 & \text{otherwise} \end{cases}$$

First, the original target  $T_T$  is obtained by executing an artifact translation written in ATL. Next, the update of the original target  $Y_{\Delta T}$  is translated into the corresponding update of the original source  $S_S$  using the virtual machine extension. The information that is necessary for the update translation in the reverse direction was recorded by the ATL virtual machine extension during the execution of the artifact translation in the forward direction. Next, the new source  $S'_S$  is merged with  $S_S^Y$ , which is the updated original source incorporating the source translation of  $Y_{\Delta T}$ . The merged artifact  $S_S''$  is the reconciled source artifact. Finally,

the reconciled source  $S_S''$  is translated into the artifact  $S_T''$ , which is then merged with the new target  $T_T'$  to produce the reconciled target artifact  $T_T''$ .

## 7 Summary of Synchronizers and Tradeoffs

In this section, we summarize the presented synchronizers and discuss the tradeoffs among them. Figure [11](#) presents a composite feature model of the design space of heterogeneous synchronizers. The feature model serves two purposes: 1) it consolidates the fragments of the feature model spread over the course of the tutorial, and 2) it provides section and page numbers of the feature descriptions. The leaf features that are not references, i.e., the leaves without ►, correspond to artifact operators.

Table [3](#) shows a feature comparison of the presented synchronizers and their inputs. Synchronizers 1–8 are unidirectional, of which Synchronizers 1–4 are to-one and Synchronizers 5–8 are to-many. Synchronizers 9–16 are bidirectional. Among them, Synchronizers 9–11 are one-to-one, Synchronizers 12–13 are many-to-one, and Synchronizers 14–16 are many-to-many.

**Tradeoffs for unidirectional to-one synchronizers.** The incremental variants offer higher performance than the non-incremental one because only individual updates are considered instead of the whole artifacts. Consequently, they enable more frequent synchronization for large artifacts. However, implementing heterogeneous artifact comparison or update translation operators is usually more complex than implementing artifact translation. The reason is that additional design decisions for implementing updates (cf. Section [8.1](#)) and matching (cf. Section [8.2](#)) need to be considered.

Furthermore, while the incremental variant based on update translation is likely to be more efficient than the one based on heterogeneous artifact comparison, the additional requirement that the original versions of the artifacts need to be consistent may be too restrictive in some situations. For example, it could be sufficient for the original versions to be nearly consistent.

**Tradeoffs for unidirectional to-many synchronizers.** The first synchronizer, i.e., the one based on artifact translation with choice and without homogeneous merge, is only useful if the target is not going to be manually edited in between target regenerations, as in the case of compiling a program into object code. If the target is intended to be edited, any of the remaining to-many variants needs to be used.

The synchronizer using the merge operator is simple to implement for cases where the structures of the target artifact that are non-reflectable in the source are well separated from the structures that are reflectable in the source. Such separation simplifies the implementation of the merge. If both kinds of structures are strongly intertwined, one of the incremental to-many synchronizers may be a better choice since they take the original target into account already in the translation operator.

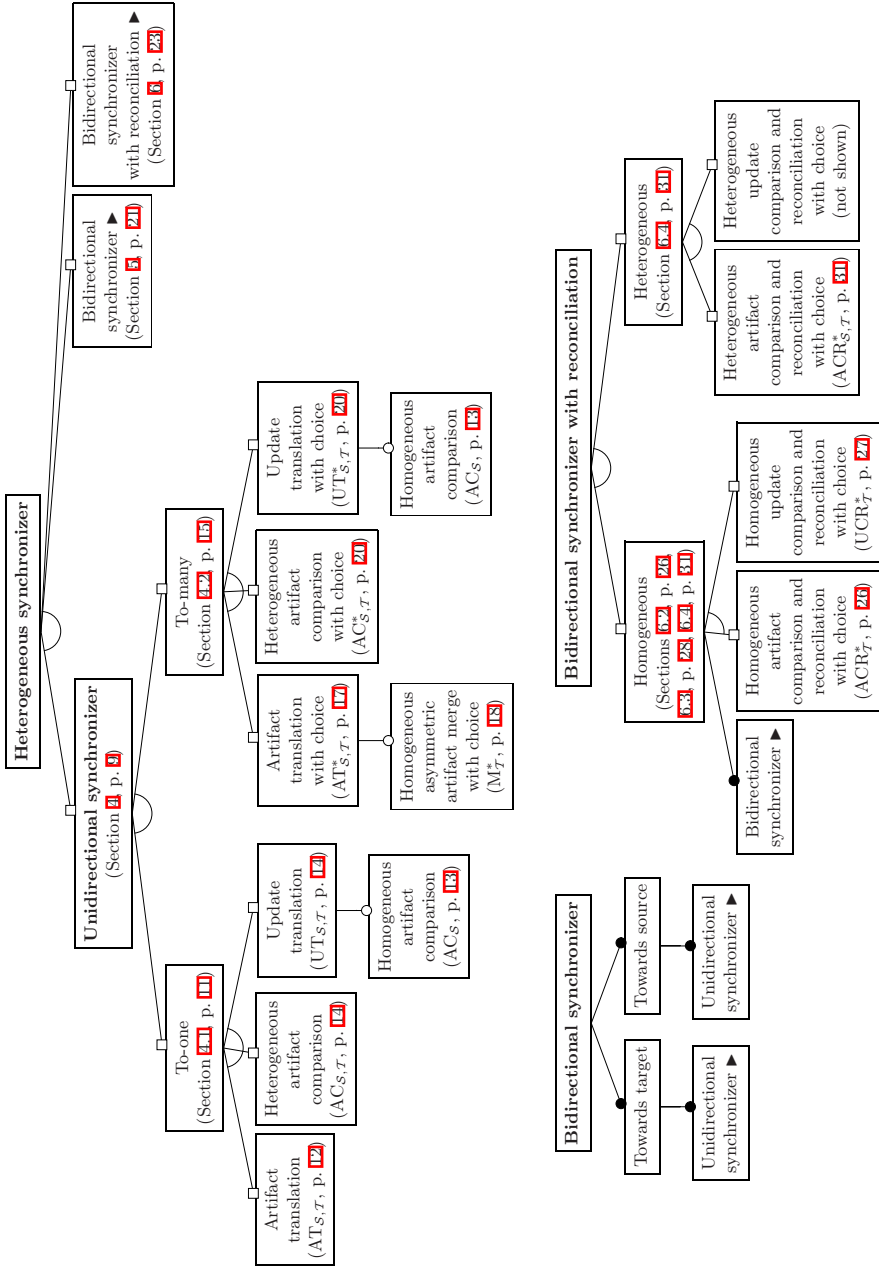


Fig. 11. Design space of heterogeneous synchronizers

**Table 3.** Summary of features and inputs of the synchronizers

		$\rightarrow 1$			$\rightarrow *$			$1 \leftrightarrow 1$			$* \leftrightarrow 1$		$* \leftrightarrow *$					
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
Synchronizer		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
increment.	non-incremental	•	.	.	.	•	•	.	.	.	.	.	.	.	.	.	.	
	(target-) incremental	.	•	•	•	.	.	•	•	•	.	.	.	.	.	.	.	
	source-incremental	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	•	
	fully-incremental	.	.	.	.	.	.	.	.	•	•	•	•	•	•	•	.	
	original-target-dependent	.	.	.	.	•	•	•	•	•	•	•	•	•	•	•	•	
inputs	$S_S$ (original source artifact)	.	.	•	•	.	.	.	•	.	.	•	•	•	•	•	•	
	$S'_S$ (new source artifact)	•	•	•	•	•	•	•	.	•	•	•	•	•	•	•	•	
	$U_{\Delta S}$ (update of the orig. source art.)	.	.	•	.	.	.	.	•	.	.	•	.	•	•	.	.	
	$D_{\mathcal{D}_S}$ (decision function on artifact)	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	•
	$D_{\mathcal{D}_{\Delta S}}$ (decision function on update)	.	.	.	.	.	.	.	.	.	.	.	•	•	•	.	.	
	$E_{\mathcal{D}_S}$ (decision function on artifact)	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	•
	$T_T$ (original target artifact)	.	•	•	•	.	•	•	•	•	•	•	•	•	•	•	•	•
	$T'_T$ (new target artifact)	.	.	.	.	.	.	.	.	•	•	•	•	•	•	•	•	•
	$Y_{\Delta T}$ (update of the orig. target art.)	.	.	.	.	.	.	.	.	.	.	•	.	•	•	.	.	
	$D_{\mathcal{D}_T}$ (decision function on artifact)	.	.	.	.	•	•	•	.	.	.	.	.	.	.	.	.	.
	$D_{\mathcal{D}_{\Delta T}}$ (decision function on update)	.	.	.	.	.	.	.	•	.	.	.	.	.	.	.	.	.
	$E_{\mathcal{D}_T}$ (decision function on artifact)	.	.	.	.	•	•	•	.	.	.	.	.	.	•	.	.	.
	$F_{\mathcal{D}_{\Delta T \times \Delta T}}$ (decision fun. on updates)	.	.	.	.	.	.	.	.	•	•	•	•	•	•	•	.	.
	$F_{\mathcal{D}_{\Delta S \times \Delta T}}$ (decision fun. on updates)	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	•
$F_{\mathcal{D}_T}$ (decision function on artifact)	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	•	
precon.	$(S_S, T_T) \in R$	.	.	•	•	.	.	.	•	.	.	•	•	•	•	•	.	
	$U_{\Delta S}(S_S) = S'_S$	.	.	.	.	.	.	.	.	.	.	.	.	•	•	.	.	
	$Y_{\Delta T}(T_T) = T'_T$	.	.	.	.	.	.	.	.	.	.	.	.	•	•	.	.	
operations	artifact translation	•	.	.	.	.	.	.	.	•	•	.	•	.	.	.	•	
	heterogeneous artifact comparison	.	•	.	.	.	.	.	.	.	.	.	.	.	.	.	.	
	update translation	.	.	•	•	.	.	.	.	.	•	•	.	•	.	.	.	
	homogeneous artifact comparison	.	.	.	•	.	.	.	.	.	.	.	.	.	.	.	.	•
	artifact translation*	.	.	.	.	•	•	.	.	.	.	.	.	.	.	.	.	
	update translation*	.	.	.	.	.	.	.	•	.	.	.	•	•	•	.	•	
	homog. asymmetric artifact merge*	.	.	.	.	.	.	•	.	.	.	.	.	.	.	.	.	•
	heterogeneous artifact comparison*	.	.	.	.	.	.	•	.	.	.	.	.	.	.	.	.	.
	homog. artifact comp. & recon.*	.	.	.	.	.	.	.	.	•	•	.	•	.	.	.	.	.
homog. update comp. & recon.*	.	.	.	.	.	.	.	.	.	.	•	.	•	•	•	.	.	
heterog. artifact comp. & recon.*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	•	.	

The tradeoffs between the two incremental synchronizers, i.e., the one using heterogeneous artifact comparison with choice and the other one using update translation with choice, are similar to the tradeoffs between their to-one counterparts.

**Tradeoffs for bidirectional synchronizers.** The choice of the unidirectional synchronizer mainly depends on the cardinalities of the relation's ends, i.e., one or many. Each of the synchronizers can be non-incremental or incremental depending on the performance requirements. If the cardinality of the end towards which the synchronizer should be executed is many, any of the unidirectional

to-many synchronizers that are original-target-dependent should be used. The synchronizer should be original-target-dependent because the target of the synchronizer can be edited.

**Tradeoffs for bidirectional synchronizers with reconciliation.** Homogeneous reconciliation is most appropriate if the relation between the source and target has at least one end with the cardinality of one because artifacts or updates can be unambiguously translated in the direction of that end. In contrast, heterogeneous reconciliation appears to be more appropriate for many-to-many relations.

## 8 Additional Design Decisions

In this Section, we present additional design decisions related to the implementation of the synchronizers: creation and representation of updates, structure identification and matching, modes of synchronization, implementation of decision functions, and construction and correctness of synchronizers.

### 8.1 Creation and Representation of Updates

An update describes the change of artifact's internal structure. The application of an update corresponds to the execution of a sequence of artifact update operations such as element additions, removals, and relocations and attribute value changes. One way of obtaining a sequence of artifact update operations is by recording editing operations performed by the artifact's developer. In this case, the artifact at the beginning of the recording is a reference artifact of the recorded update. We refer to updates obtained by recording developer's edits as *history-based updates*. The sequence of developer's edits can be transformed into a *canonical form*, which produces the same result as the original sequence, but lacks redundant edits, such as, modifying the same attribute multiple times. Another way of creating an update is by comparing two artifacts using *homogeneous artifact comparison*. We refer to updates obtained by comparing two artifacts as *state-based updates*. History-based updates contain more information than state-based updates, but may be more difficult to implement in practice. For example, implementations have to ensure that all artifact updates are performed through an appropriate change-tracking interface.

### 8.2 Structure Identification and Matching

Comparison operators such as homogeneous and heterogeneous artifact comparison require a way to establish the correspondence between the elements of the artifacts being compared. Furthermore, the implementation of an update function must also contain information about the elements it affects and a way to identify them in the reference artifact. We refer to the process of establishing the correspondence between elements as *matching*.



The two fundamentally different approaches to matching are *non-structural matching* and *structural matching*. Non-structural matching assumes that elements receive globally unique, structure-independent identifiers at the time of their creation. By “globally unique” we mean that the identifiers are unique at least in the scope of the matched artifacts. Structure-independent means that the identifiers are independent of the artifact structure, meaning that they remain constant when the structure evolves. For example, the identifier could be generated as a combination of the IP address of the machine where the identifier is generated, a timestamp, and a random number. This approach greatly simplifies matching among different versions of an artifact as the correspondence of elements can be established immediately based on the equality of the identifiers. The main drawback of this approach is that it tends to be brittle with respect to artifact evolution that involves a deletion and subsequent recreation of an element. For example, consider the removal of a method from a class and its later re-introduction. The new method would have a new identifier, which would mark it as a new element even though it is probably just a new version of the original method. Furthermore, identifiers tend to pollute and bloat the artifacts, especially if they need to be stored in a human-readable textual form.

Structural matching avoids both problems by establishing correspondence through the structural information that is already in the artifacts, e.g., element nesting, element’s position in ordered lists, and attribute values such as element’s local name. In our method evolution example, the old and the new version of the method could be matched by using the fully qualified name of the containing class and the method’s signature as an identifier. The matching can still use precomputed identifiers, but these identifiers would be structure dependent as they encode structural information. The main drawback of structural matching is that sometimes the structural information needed for recovering a particular relationship may be missing or difficult to identify. For example, while the fully qualified name and signature of a method is sufficient to unambiguously match a single call to that method within the body of another method, identifying multiple calls within a single body is challenging. Using the lexical order of the calls is a possible solution, but one that is brittle with respect to evolution when the body is restructured, for example, when additional calls are inserted in the middle of the body. A practical solution may need to use more local context information of each call in order to establish the correspondence between the two versions. The problem of recognizing element relocations in nested structures is an active research topic, e.g., [31].

In practice, both non-structural and structural matching can be used in combination. For example, the model management infrastructure of IBM Rational Software Modeler [32], which is IBM’s UML modeling tool, supports both non-structural and structural matching.

Finally, matching could be realized at a *semantic level* rather than a structural (i.e., syntactic) one. For example, Nejati et al. [33] present an approach for matching and merging Statecharts specifications that respects the behavioral semantics of the specifications.

### 8.3 Instantaneous vs. On-Demand Synchronization

Another design decision is the time of update propagation. We distinguish between *instantaneous* and *on-demand* synchronization. Instantaneous synchronization translates and applies updates to the target artifact immediately after the updates occurred in the source artifact. On-demand synchronization translates and applies updates at the time most convenient for the developer. Instantaneous synchronization is likely to require an incremental synchronizer since translating the entire source artifact after each update would be highly inefficient.

### 8.4 Disconnected vs. Live Synchronization

Update propagation can be implemented as a *disconnected* or a *live* transformation. Live transformation is a transformation that does not terminate [18,25] and whose intermediate execution data, referred to as *execution context*, is preserved. The context of a live transformation maintains the links between structures in the source artifact and the resulting structures in the target artifact. The preservation of the execution context allows for efficient propagation of updates made to the source artifact (cf. Example 10). In contrast, a disconnected transformation terminates and its execution trace is lost, in which case a structure matching mechanism is needed (cf. Section 8.2).

### 8.5 Strategies for Selecting Synchronization Result from Multiple Choices

Synchronization in the “to-many” direction requires a way to select a single target from the set of possible targets that are consistent with the source. We distinguish among the following selection strategies:

- *Pre-determined choice*: The choice is fixed by the synchronizer developer and hardcoded in the synchronizer.
- *Interactive selection*: The available choices, typically ranked according to some criteria, are presented to the user interactively. While the number of choices may be infinite, a finite number is presented at a time and the user can ask for more.
- *User-specified defaults*: The user may use global options to specify preference. Alternatively, the choices may be related to individual source elements, in which case the source elements are annotated. Examples of annotation mechanisms are Java annotations and UML profiles.
- *Adaptive defaults*: The default settings could be obtained by mining from the original target or from a corpus of existing sample targets. An example of this strategy is the automatic application of code formatting that was extracted from a corpus of sample programs using data mining techniques [34].
- *Target preservation*: The available choices may be restricted by the desire to preserve structures in the original target. We accounted for this possibility in the original-target-dependent synchronizers.

## 8.6 Construction of Bidirectional Synchronizers

Bidirectional synchronizers can be constructed using either a bidirectional or a unidirectional transformation language. Synchronizers constructed using a bidirectional language can be directly executed in both directions from a single specification.

Examples of bidirectional transformation languages include QVT Relations [35], *triple graph grammars* [25,36] (TGGs), and Lenses for trees [17]. In QVT and TGGs, synchronizers are expressed by a set of rules, which can be executed in both forward and reverse directions. Implementations of the QVT Relations language include ModelMorf by TATA Research Development and Design Centre and Medini QVT by IKV++. Tool support for creating TGG-based synchronizers exists as a plug-in for the FUJABA tool suite [26]. In the Lenses approach complex bidirectional synchronizers are implemented by composing bidirectional primitives using *combinators*. Similarly to lenses, Xiong et al. propose an approach to building bidirectional synchronizers using combinators that translate modification operations performed on one artifact to synchronizing operations on the other artifact [37]. In this approach, a synchronizer is defined by creating a *synchronizer graph*, which consists of primitive synchronizers, input artifacts, and intermediate (temporary) artifacts. The approach additionally supports different synchronization behaviors by parameterizing primitive synchronizers with mode options.

Using a unidirectional transformation language requires either writing two unidirectional synchronizers, one in each direction, or writing a unidirectional synchronizer in one direction and automatically computing its inverse. Depending on the type of relation among the artifacts, the two unidirectional transformations can be constructed in many ways. For some bijections, an inverse transformation can be automatically computed from the transformation in one direction. Pierce provides a list of examples of interesting cases of computing such inverse transformations [38]. Xiong et al. [30] developed an approach that can execute a synchronizer written in ATL, a unidirectional language, in the reverse direction (cf. Example 20). The information that is necessary for the reverse transformation is recorded by an extension to the ATL virtual machine during the execution of the synchronizer in the forward direction.

## 8.7 Correctness of Synchronizers

In practice, establishing full consistency automatically may not always be possible. First, developers may desire to synchronize partially finished artifacts, i.e., the synchronizer may need to be able to handle artifacts of which only parts are well-formed. Second, the complex semantics of some artifacts and relations can sometimes be only approximated by programs implementing translation operators. For example, a synchronizer that operates on program code may need to rely on static approximations of control and data flow.

Code queries for FSMLs exemplify both situations [16]. The precise FSML semantics relate model elements with structural and behavioral patterns in Java

code. However, the code queries implementing the reverse engineering for the behavioral patterns are incomplete and unsound approximations of the behavioral patterns. Furthermore, the code query evaluation engine relies on an incremental Java compiler, which allows for querying code that does not completely compile.

## 9 Related Work

In this section we discuss related works in three areas: data synchronization in optimistic replication, inconsistency management in software development, and model management and model transformation.

### 9.1 Data Synchronization in Optimistic Replication

The need for synchronization arises in the area of optimistic replication, which allows replica contents to diverge in the short term in order to allow concurrent work practices and to tolerate failures in low quality communication links. Optimistic replication has applications to file systems, personal digital assistants, internet services, mobile databases, and software revision control. Saito and Shapiro [39] provide an excellent survey of optimistic replication algorithms, which are essentially synchronization algorithms. They distinguish the following phases of synchronization: update submission at multiple sites, update propagation, update scheduling, conflict detection and resolution, and commitment to final reconciliation result. The scheduling of update operations is of particular interest in the context of multiple master sites with background propagation, which leads to the challenge that not all update operations are received at all sites in the same order. Furthermore, Saito and Shapiro distinguish several key characteristics of optimistic replication:

- *Single vs. multi-master synchronization*: Synchronization scenarios can involve different numbers of master sites. Master sites are those that can modify replicas. In contrast, slave sites store read-only replicas. The scope of this tutorial is limited to master-slave (i.e., unidirectional) and master-master (i.e., bidirectional) synchronization.
- *State-transfer vs. operation transfer*: We discussed this distinction in Section 8.1.
- *Conflict detections and resolution granularity*: Conflicts may be easier to resolve if smaller sub-objects are considered.
- *Syntactic vs. semantic update operations*: Replicas can be compared syntactically or semantically. This distinction is concerned with the extent to which the synchronizer system is aware of the application semantics of the replicas and the update operations. Semantic approaches avoid some conflicts that would arise in syntactic approaches, but are more challenging to implement.
- *Conflict management*: This characteristic is concerned with the way the system defines and handles conflicts. Conflict detection policies can be syntactic or semantic. Conflict resolution may involve selecting one update among a

set of conflicting ones while the others are discarded, storing all conflicting updates in each synchronized replica, or allowing replicas to diverge for conflicting updates [27].

- *Update propagation strategy*: This dimension includes the degree of synchrony, e.g., pull vs. push strategies, and the communication topology, e.g., star vs. ad-hoc propagation.
- *Consistency guarantees*: Some synchronizers may guarantee consistency of the accessed replicas while other may give weaker guarantees, such as guaranteeing that the state of replicas will eventually converge to being consistent.

An additional dimension given by Foster et al. [27] is

- *Homogeneity vs. heterogeneity*: This dimension refers to the distinction whether the data to be synchronized adheres to a single schema or to different schemas expressed in the same schema language (e.g., relational algebra). The focus of this tutorial is on heterogeneous synchronization.

Saito and Shapiro [39] and Foster et al. [27] give many example of existing synchronization systems; however, Harmony [27] seems to be the only generic synchronizer handling *heterogeneous* replicas. Harmony is concerned with the special case of mappings which are functions. The same case is also studied in databases as the *view update problem*, e.g., see Bancilhon and Spyrtos [40] and Gottlob et al. [41].

## 9.2 Data Integration and Schema Mapping

Another related area is *data integration*, which is concerned with integrating data from multiple sources, such as different databases. A particular challenge in this context is *schema integration*, i.e., the integration of the vocabularies defined by the schemas, which is addressed by *schema matching*. Bernstein and Rahm [42] provide an excellent survey of approaches to automated schema matching.

## 9.3 Inconsistency Management in Software Development

Software artifact synchronization is a topic in *inconsistency management* in software engineering [2, 5, 6, 8, 43]. Spanoudakis and Zisman [8] provide a survey of this area. They identify a broad set of activities related to inconsistency management: detection of overlaps (i.e., identification of relationships), detection of inconsistencies, diagnosis of inconsistencies, handling of inconsistencies, tracking (not all inconsistencies need to be resolved), and specification and application of an inconsistency management policy. Grundy and Hosking [7] explore architectures and user-interface techniques for inconsistency management in the context of multiple-view development environments.

## 9.4 Model Management and Model Transformation

Software artifact synchronization is also closely related to *model management* and *model transformation*. In *model-driven software development* (MDSD) [12],

models are specifications that are inputs to automated processes such as code generation, specification checking, and test generation. Furthermore, models in MDSD are typically represented as object graphs conforming to a class model usually referred to as a *metamodel*.

*Model management* is concerned with providing operators on models such as comparison, splitting, and merging. Bernstein et al. argued for the need of such generic model operators and the existence of mappings among models as first-class objects [44]. Later, Bernstein applied the model management operators to three problems: schema integration, schema evolution, and round-trip engineering [45]. Brunet et al. [4] wrote a manifesto for model management, in which they argue for an algebraic framework of model operators as a basis for comparing different approaches to model merging. Indeed, the use of operators in our design space was partly inspired by this manifesto. The *diff* operator corresponds to the homogeneous comparison operator presented in this tutorial. Furthermore, the manifesto refers to updates as *transformations* and to the application of updates as *patching*. The manifesto defines additional operators, e.g., *split* and *slice*. The operators in this tutorial treat the relation  $R$  as an implicit parameter. In contrast, the operators in the manifesto are defined explicitly over artifacts and *relations*. While the manifesto focuses on homogeneous merge, our design space is concerned with heterogeneous synchronization. In fact, bidirectional synchronizers with reconciliation can be understood as heterogeneous merge operations. One of the uses of model management is detecting and resolving inconsistencies in models, e.g., see work by Egyed [46] and Mens [47]. Sriplakich et al. [48] discuss a middleware approach to exchanging model updates among different tools. Finally, Diskin [49,50] proposes using category theory as a mathematical formalism for expressing the operators for both homogeneous and heterogeneous generic model management.

Another related area is *model transformation*, which is concerned with providing an infrastructure for the implementation and execution of operations on models. Mens et al. [51] provide a taxonomy of model transformation and apply it to model transformation approaches based on graph transformations [52]. The taxonomy discusses several tool-oriented criteria such as level of automation, preservation, dealing with incomplete and inconsistent models, and automatic suggestion of transformations based on context. Czarnecki and Helsen [53] survey 26 approaches to model transformation. The survey and the design space presented in this tutorial both use a feature-based approach and have some features in common, such as target incrementality, source incrementality, and preservation of user edits in the target. In contrast to this tutorial, the survey mainly focuses on the different paradigms of transformation specification, such as relational, operational, template-based, and structure-driven approaches, and it does not consider reconciliation. Some ideas for an algebraic semantics for model transformations are presented by Diskin and Dingel [54].

The topic of bidirectional model transformation has recently attracted increased attention in the modeling community. Stevens [24] analyzes properties of the relational part of OMG's Query View Transformation (QVT) and

argues that more basic research on bidirectional transformation is needed before practical tools will be fully realizable. Giese and Wagner [25] identify a set of concepts around bidirectional incremental transformations. In particular, they distinguish between *bijective* and *surjective* bidirectional transformations. The latter correspond to the situation where several sources correspond to a single target. Furthermore they refer to a transformation as *fully incremental* if the effort of synchronizing a source model change is proportional to the size of the source change. Finally, Ehrig et al. [36] study the conditions under which model transformations based on triple-graph grammars are reversible.

## 10 Conclusion

In this tutorial we explored the design space of heterogeneous synchronization, i.e., the synchronization of artifacts of different types. We presented a number of artifact operators that can be used in the implementation of synchronizers and presented 16 example synchronizers. The example synchronizers illustrate different approaches to synchronization and can be characterized along a number of dimensions, such as directionality, incrementality, original-target-dependency, and support for the reconciliation of concurrent updates. For some of the synchronizers, we provided examples of existing systems that implement a given approach to synchronization. Furthermore, we discussed a number of additional design decisions such as representation of updates, establishing correspondence among model elements, and strategies for selecting a single synchronization result from a set of alternatives. Finally, we discussed important works in related fields including data synchronization, inconsistency management in software engineering, model management, and model transformation.

**Acknowledgments.** The authors would like to thank Zinovy Diskin, Lech Tuzinkiewicz, and the anonymous reviewers for their valuable comments on earlier drafts of this tutorial.

## References

1. Frederick, P., Brooks, J.: No silver bullet: essence and accidents of software engineering. *Computer* 20(4), 10–19 (1987)
2. Nuseibeh, B., Kramer, J., Finkelstein, A.: Expressing the relationships between multiple views in requirements specification. In: ICSE, pp. 187–196 (1993)
3. Maier, M.W., Emery, D., Hilliard, R.: Software architecture: Introducing IEEE standard 1471. *Computer* 34(4), 107–109 (2001)
4. Brunet, G., Chechik, M., Easterbrook, S., Nejati, S., Niu, N., Sabetzadeh, M.: A manifesto for model merging. In: GaMMa, pp. 5–12 (2006)
5. Balzer, R.: Tolerating inconsistency. In: ICSE, pp. 158–165 (1991)
6. Easterbrook, S., Nuseibeh, B.: Using viewpoints for inconsistency management. *BCS/IEE Software Engineering Journal* 11(1), 31–43 (1996)
7. Grundy, J., Hosking, J., Mugridge, W.B.: Inconsistency management for multiple-view software development environments. *IEEE Trans. Softw. Eng.* 24(11), 960–981 (1998)



8. Spanoudakis, G., Zisman, A.: Inconsistency management in software engineering: Survey and open research issues. In: Handbook of Software Engineering and Knowledge Engineering, pp. 329–380. World Scientific Publishing Co, Singapore (2001)
9. Jouault, F., Bézivin, J.: KM3: a DSL for metamodel specification. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 171–185. Springer, Heidelberg (2006),  
<http://www.lina.sciences.univ-nantes.fr/Publications/2006/JB06a>
10. Antkiewicz, M.: Framework-Specific Modeling Languages. PhD thesis, University of Waterloo (2008) (submitted for review)
11. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Professional, Reading (2002)
12. Stahl, T., Völter, M.: Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons, Chichester (2006)
13. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609. Springer, Heidelberg (2007)
14. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (1990)
15. Czarnecki, K., Kim, C.H.P.: Cardinality-based feature modeling and constraints: A progress report. In: OOPSLA International Workshop on Software Factories (2005); On-line proceedings
16. Antkiewicz, M., Tonelli Bartolomei, T., Czarnecki, K.: Automatic extraction of framework-specific models from framework-based application code. In: ASE, pp. 214–223 (2007)
17. 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: POPL, pp. 233–246 (2005)
18. Hearnden, D., Lawley, M., Raymond, K.: Incremental model transformation for the evolution of model-driven systems. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 321–335. Springer, Heidelberg (2006)
19. Eclipse Foundation: Java Emitter Templates Component (2007),  
<http://www.eclipse.org/modeling/m2t/?project=jet>
20. Eclipse Foundation: Eclipse Modeling Framework Project (2007),  
<http://www.eclipse.org/modeling/emf/?project=emf>
21. Nickel, U.A., Niere, J., Wadsack, J.P., Zündorf, A.: Roundtrip engineering with FUJABA. In: WSR, Fachberichte Informatik, Universität Koblenz-Landau (2000)
22. Aßmann, U.: Automatic roundtrip engineering. Electr. Notes Theor. Comput. Sci. 82(5) (2003)
23. Sendall, S., Küster, J.M.: Taming model round-trip engineering (2004)
24. Stevens, P.: Bidirectional model transformations in QVT: Semantic issues and open questions. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 1–15. Springer, Heidelberg (2007)
25. Giese, H., Wagner, R.: Incremental Model Synchronization with Triple Graph Grammars. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 543–557. Springer, Heidelberg (2006)
26. Kindler, E., Wagner, R.: Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical Report tr-ri-07-284, Software Engineering Group, Department of Computer Science, University of Paderborn (2007)
27. Foster, J.N., Greenwald, M.B., Kirkegaard, C., Pierce, B.C., Schmitt, A.: Exploiting schemas in data synchronization. J. Comput. Syst. Sci. 73(4), 669–689 (2007)



28. Antkiewicz, M., Czarnecki, K.: Framework-specific modeling languages; examples and algorithms. Technical Report 2007-18, ECE, University of Waterloo (2007)
29. ATLAS Group: ATLAS Transformation Language (2007),  
<http://www.eclipse.org/m2m/at1/>
30. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In: ASE, pp. 164–173 (2007)
31. Abi-Antoun, M., Aldrich, J., Nahas, N., Schmerl, B., Garlan, D.: Differencing and merging of architectural views. In: ASE, pp. 47–58 (2006)
32. IBM: Rational Software Modeler (2007),  
<http://www-306.ibm.com/software/awdtools/modeler/swmodeler/>
33. Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S.M., Zave, P.: Matching and merging of statecharts specifications. In: ICSE, pp. 54–64 (2007)
34. Reiss, S.P.: Automatic code stylizing. In: ASE, pp. 74–83 (2007)
35. Object Management Group: MOF QVT Final Adopted Specification. OMG Adopted Specification ptc/05-11-01 (2005),  
<http://www.omg.org/docs/ptc/05-11-01.pdf>
36. Ehrig, H., Ehrig, K., Ermel, C., Hermann, F., Taentzer, G.: Information preserving bidirectional model transformations. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 72–86. Springer, Heidelberg (2007)
37. Xiong, Y., Hu, Z., Takeichi, M., Zhao, H., Mei, H.: On-site synchronization of software artifacts. Technical Report METR 2008-21, Department of Mathematical Informatics, University of Tokyo (2008),  
<http://www.ipl.t.u-tokyo.ac.jp/~xiong/papers/METRO8.pdf>
38. Pierce, B.C.: The weird world of bi-directional programming (2006) ETAPS invited talk, slides,  
<http://www.cis.upenn.edu/~bcpierce/papers/lenses-etapsslides.pdf>
39. Saito, Y., Shapiro, M.: Optimistic replication. *ACM Comput. Surv.* 37(1), 42–81 (2005)
40. Bancilhon, F., Spyratos, N.: Update semantics of relational views. *ACM Trans. Database Syst.* 6(4), 557–575 (1981)
41. Gottlob, G., Paolini, P., Zicari, R.: Properties and update semantics of consistent views. *ACM Trans. Database Syst.* 13(4), 486–524 (1988)
42. Rahm, E., Bernstein, P.A.: A survey of approaches to automatic schema matching. *The VLDB Journal* 10(4), 334–350 (2001)
43. Nuseibeh, B., Kramer, J., Finkelstein, A.: Viewpoints: meaningful relationships are difficult! In: ICSE, pp. 676–681 (2003)
44. Bernstein, P.A., Halevy, A.Y., Pottinger, R.A.: A vision for management of complex models. *SIGMOD Rec.* 29(4), 55–63 (2000)
45. Bernstein, P.: Applying model management to classical metadata problems. In: CIDR (2003)
46. Egyed, A.: Fixing inconsistencies in UML design models. In: ICSE, pp. 292–301 (2007)
47. Mens, T., Straeten, R.V.D., D’Hondt, M.: Detecting and resolving model inconsistencies using transformation dependency analysis. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 200–214. Springer, Heidelberg (2006)
48. Sriplakich, P., Blanc, X., Gervais, M.P.: Supporting transparent model update in distributed case tool integration. In: SAC, pp. 1759–1766 (2006)
49. Diskin, Z., Kadish, B.: Generic model management. In: Rivero, Doorn, Ferragine (eds.) *Encyclopedia of Database Technologies and Applications*, pp. 258–265. Idea Group (2005)

50. Diskin, Z.: Mathematics of generic specifications for model management. In: Rivero, Doorn, Ferraggine (eds.) *Encyclopedia of Database Technologies and Applications*, pp. 351–366. Idea Group (2005)
51. Mens, T., Van Gorp, P.: A taxonomy of model transformation. In: *Proc. Int'l Workshop on Graph and Model Transformation* (2005)
52. Mens, T., Van Gorp, P., Varro, D., Karsai, G.: Applying a model transformation taxonomy to graph transformation technology. In: *Proc. Int'l Workshop on Graph and Model Transformation* (2005)
53. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. *IBM Systems Journal* 45(3), 621–645 (2006)
54. Diskin, Z., Diengel, J.: A metamodel independent framework for model transformation: Towards generic model management patterns in reverse engineering. In: Favre, J.-M., Gasevic, D., Laemmel, R., Winter, A. (eds.) *3rd Int. Workshop on Metamodels, Schemas, Grammas and Ontologies for reverse engineering, ATEM 2006* (2006)

# Software Reuse beyond Components with XVCL (Tutorial)

Stan Jarzabek

Department of Computer Science, School of Computing  
National University of Singapore, Singapore 117543  
stan@comp.nus.edu.sg

**Abstract.** The basic idea behind software reuse is to exploit similarities within and across software systems to avoid repetitive development work. Conventional reuse is based on components and architectures. We describe how reuse of structural similarities extends the benefits of conventional component reuse, and realization of the concept with a generative technique of XVCL<sup>1</sup>. Structural similarities are repetition patterns in software of any type or granularity, from similar code fragments to recurring architecture-level component configuration patterns. We represent any significant repetition pattern in subject system(s) with a generic, adaptable, XVCL meta-structure. We develop, reuse and evolve software at the level of meta-structures, deriving specific, custom systems from their meta-level representations. Lab studies and industrial applications of XVCL show that by doing that, on average, we raise reuse rates and productivity by 60-90%, reducing cognitive program complexity and maintenance effort by similar rates. The approach scales to systems of any size. The benefits are proportional to system size, and to the extent of repetitions present in subject system(s). The main application of this reuse strategy is in supporting software Product Lines.

## 1 Introduction

Software reuse is such a tempting idea as we see so much similarity within and across software systems. Experienced developers become aware of the fact that software development involves common themes that recur in variant forms from project to project, and from one software system to another. Effective reuse strategy should help us avoid repetitive development work. With reuse, we hope to exploit productivity improvements similarities offer, rather than develop similar systems from scratch.

Software Product Line (PL) approach [12] focuses on domain-specific reuse, within a family of software systems that are known to have much in common with one another. Domain-specific reuse can be easier and more effective than reuse across arbitrary, possibly very dissimilar systems.

Consider a family of Role Playing Games (RPG) for mobile phones (Fig. 1). An RPG player takes the role of a fictional character and participates in an interactive

---

<sup>1</sup> XVCL: XML-based Variant Configuration Language, [xvcl.comp.nus.edu.sg](http://xvcl.comp.nus.edu.sg)

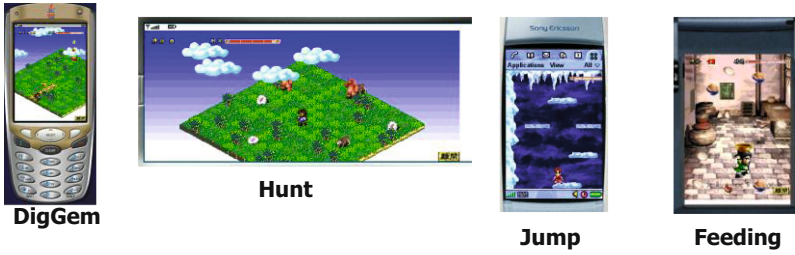


Fig. 1. A Product Line of games for mobile phones

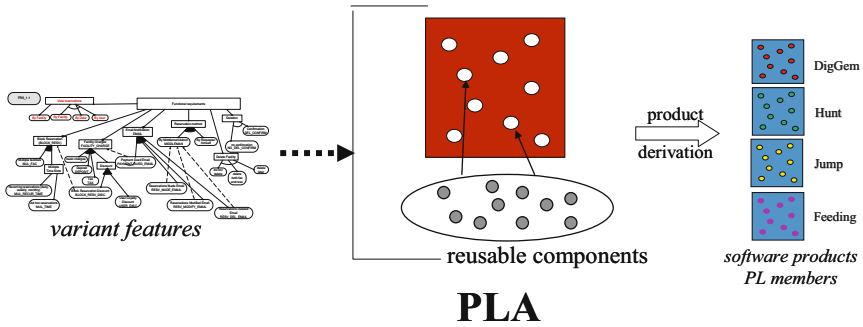


Fig. 2. Product Line Architecture (PLA) and product derivation

story. All RPGs are similar, but they also differ in some functional requirements, and in characteristics of a specific mobile device on which they run. The goal of mobile device contents providers is to support possibly large number of games, on a possibly wide range of mobile devices. Games for new mobile device models should be delivered fast. Much similarity among games makes reuse a promising approach to cut game development time and effort. Product Line (PL) approach targets this goal [48].

The core idea of a PL approach is to set up a base of reusable assets, so-called PL Architecture, PLA for short. Specific products are then built by reusing assets stored in a PLA.

Fig. 2 depicts the main concepts behind the PL approach. A range of variant features supported by a PLA are shown as a feature diagram on the left-hand-side of the figure (details are not meant to be read). Reuse-based development is called *product derivation*. Product derivation starts by selecting variant features for a product we intend to build. We try to understand the impact of variant features on PLA components, and select component configurations “best matching” variant features for a product. This is followed by component customization to accommodate the impact of variant features on components. Then, we integrate components to form a custom product, and validate the product. If component integration or validation fails, we may need to repeat the component selection, customization, integration and validation cycle until the requirements for a new PL member are properly met.

The above product derivation model generalizes experiences from a number of industrial PL projects [15], and we use it as a reference model in this paper.

Software architectures and components are the main concepts behind conventional approaches to realizing reuse in PLs. Modern component platforms (e.g., JEE or .NET), design patterns, parameterization (e.g., Java generics, C++ templates, higher-level functions or macros) and inheritance also contribute useful reuse solutions in certain situations. Platform mechanisms have many advantages, but reuse benefits are mostly limited to common services and middleware layers. Reuse potentials on a system-wide scale, especially in the application domain-specific areas of business logic and user interfaces, are more difficult to realize with component platform mechanism alone. By setting up a PL, companies can also aim at reuse in domain-specific areas.

Some companies established and benefited from PL programmes [12][27][37]. At the same time, a number of problems have also been reported. First, companies observe explosion of similar component versions in PLA. This hinders selection and customization of components for reuse when deriving new products from a PLA. Functionality already implemented may be difficult to reuse in new products [15], defeating the very purpose of establishing a PL. Second, product derivation lacks automation, and is done mostly manually, with help of complementary techniques such as wizards or configuration files. Finally, to our best knowledge, benefits of component/architecture-based reuse are mainly observed during new development, but are less evident in long-term evolution of successful products.

It is the role of variability management strategy to provide effective solution to the above problems. The main challenge of reuse is to handle variability across PL members: We wish to derive PL members from a common set of reusable software assets, a PLA. PL members share similarities, but variability in a domain causes that they also differ one from another in variant features. Effectiveness of a reuse strategy depends on how well we can exploit similarities, and deal with differences.

It follows that the ability to represent software in a generic and highly adaptable form should be a prerequisite for successful reuse, and a prominent characteristic of a PLA. Genericity is needed to express similarities among PL members. Adaptability takes care of variant features that differ from one PL member to another.

Generic, adaptable software representations are the heart and soul of reuse. However, they are difficult to build with conventional component architectures. Consequently, genericity remains underutilized in realizing today's reuse strategies. Components typically stored in a PLA are not generic enough, and their adaptation, mostly manual, is too difficult for effective reuse.

Generic design is easier to achieve at the meta-level program representation than at the level of conventional components. In this paper, we show a pragmatic way to strengthen generic design capabilities of conventional reuse techniques with a generative programming technique of XVCL [45]. The approach works as follows: We do initial design using conventional techniques, and then apply XVCL to build generic, adaptable meta-level structures. By doing that, we unify and reuse structural similarity patterns of all types and granularity (e.g., similar classes, components and patterns of collaborating components) for which conventional generic representations may not exist.

Such approach reaps reuse opportunities beyond what is possible with conventional component/architecture reuse. It works for common middleware services, as well as for application domain layers of user interfaces and business logic, which are particularly difficult to componentize for reuse. On average, we reduce cognitive program

complexity (and maintenance effort) of a program solution by 60-90%, raising the levels of reuse by similar rates.

PLA contains all types of software assets such as code, documentation, models, and test cases. Similarities and differences occur in PLA software assets and variability management should address all of them. XVCL can manage variability in any assets that can be expressed as text, written in a formal or informal language. Having said that, in this paper we focus only on code assets.

In Section 2, we discuss the software similarity phenomenon with examples. Sections 3 motivates XVCL and describes its concepts. Section 4 introduces detailed XVCL mechanisms by means of a toy example. Section 5 illustrates application of XVCL. We evaluate the XVCL approach in Section 6. Related work and conclusions end the paper.

## 2 Software Similarity Phenomenon

Similarities are inherent in software. They show within and across application domains as recurring similar software structures, so-called software clones. Clones appear in software for variety of reasons. Ad hoc copy-paste-modify practice leads to repetitions. Recurring patterns of software requirements or design also induce repetitions (e.g., analysis patterns [19] or design patterns [21]). Some clones are intentional and play a useful role in a program [31] while some clones occur because of the limitations of a programming language [26][33]. Cordy [13] describes situations where refactoring clones is not a viable option because of the risks involved in changing the software. Similarly, Rajapakse [39] describes trade-offs involved in refactoring clones in web applications developed with PHP. Clones can also be induced by a design techniques, for example, by standardized architectures and pattern-driven development (e.g., Web architectures, JEE or .NET). Uniformity of design is desirable despite inducing many repetitions. The above observations suggest that in many situations, refactoring clones from programs is neither possible nor even desirable.

In case of families of similar systems, such as Software Product Lines, repetitions are expected, evident and pervasive. With XVCL, we capture similar program structures recurring in products in a generic, non-redundant form at the meta-level, while preserving clones intact in the actual program derived from the meta-level program representation.

In this paper, we show the benefits of and trade-offs involved in changing the perspective from component reuse, to meta-level reuse of any structurally similar software representations.

### 2.1 Simple and Structural Clones

*Software clones* are any program structures of considerable size and significant similarity, irrespective of their type and granularity. The actual size and similarity (which can be measured, for example, in terms of percentage of repeated code) is subjective, varies with context, hence is left to human judgment. Similarity is a multi-faceted phenomenon that escapes precise definition.

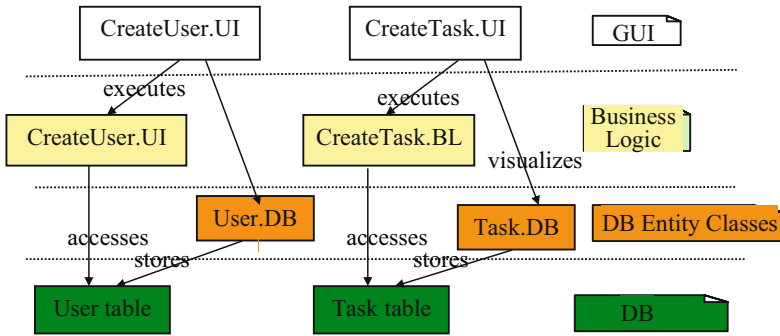


Fig. 3. Structural clones spanning multiple tiers of DEMS

Clones may or may not represent program structures that perform well-defined functions. It is structural similarity among program structures, not their function that is of our interest in this paper.

Research so far mostly focused on similar code fragments, so-called *simple clones*. Simple clones may differ in parametric or non-parametric ways, for example some clone instances may have inserted or deleted code lines as compared to others.

Software similarities are not limited to simple clones; similarities also exist at higher levels of software representation. We call large-granular, design-level similar program structures as *structural clones* [1]. Structural clones are patterns of inter-related components/classes. They are often induced by the application domain (analysis patterns) or design techniques.

Cloning situation shown in Fig. 3 has been found in a Domain Entity Management Subsystem (DEMS) of a command-and-control application developed in C# by our industry partner ST Electronics Pte Ltd (STEE). DEMS involves domain entities such as *User*, *Task* or *Resource*. For each entity, there are operations, such as *Create*, *Update*, *View*, *Delete*, *Find* or *Copy*.

The design of each operation such as *CreateUser* or *Create Task* involves a pattern of collaborating classes from GUI, service and database layers. Each box in Fig. 3 represents a number of classes: GUI classes implement various forms to display or enter data; Business Logic classes implement data validation and actions specific to various operations and/or entities; Entity classes define data access; classes at the bottom contain table definitions. Classes in corresponding boxes at each level display much similarity, but there are also differences induced by different semantics of domain entities: For example, operation *CreateTask* requires different types of data entry and data validation than *CreateUser*.

Patterns of components implementing operations such as *CreateUser* and *CreateTask* form a structural clone class.

## 2.2 Clones in the Buffer Library

A study of the Buffer library JDK 1.5 provides interesting insights into sources of software similarities. It also sheds light on the reasons why it is difficult to avoid repetitions with conventional programming techniques such as componentization,

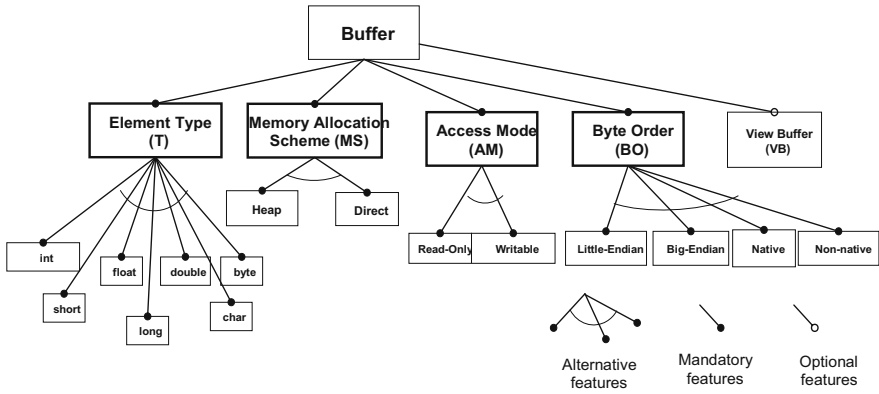


Fig. 4. Features of buffer classes

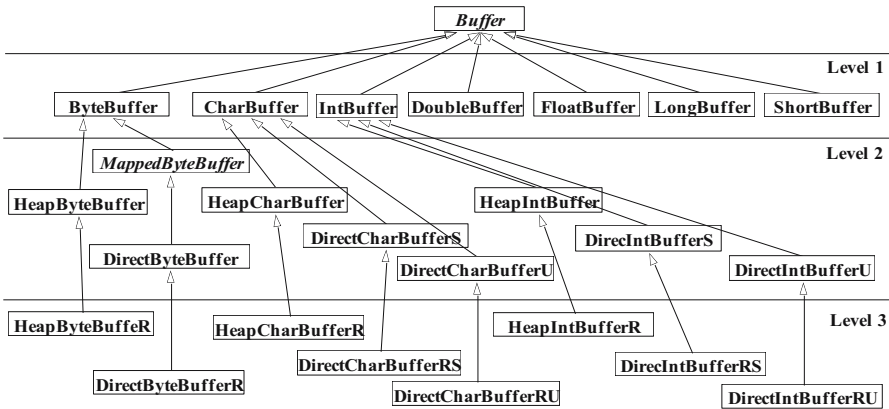


Fig. 5. A fragment of the Buffer library

type parameterization or inheritance. Here, we summarize the results from the Buffer library study, referring the reader to [26] for details. Code and step-by-step explanation of the Buffer library study can be found on XVCL Web site [45]. A study of STL [3,41] further strengthens observations we make in this section.

A buffer contains data in a linear sequence for reading and writing. Buffer classes differ in buffer element type, memory allocation scheme, byte ordering and access mode. Features of buffer classes are shown in Fig. 4, as a feature diagram [29]. We see five features groups, with specific variant features listed below a respective feature group. Each legal combination of features from various groups yields a unique buffer class. As we combine features, buffer classes grow in number, as observed in [5,9].

Some of the buffer classes are shown in Fig. 5. A class name, such as DirectInt-BufferRS, reflects combination of features implemented into a given class. Class names are derived from a template:



[MS][T]Buffer[AM][BO],

where

MS – memory scheme: Heap or Direct;

T – type: int, short, float, long double, char, or byte;

AM – access mode: W – writable (default) or R - read-only;

BO – byte ordering: S – non-native or U – native;

B – BigEndian or L – LittleEndian.

Classes whose names do not include ‘R’, by default are ‘W’ – writable.

Examination of buffer classes reveals much similarity among classes in seven groups, namely

[T]Buffer,  
 Heap[T]Buffer,  
 Heap[T]BufferR,  
 Direct[T]Buffer[S|U],  
 Direct[T]BufferR[S|U],  
 ByteBufferAs[T]Buffer[B|L], and  
 ByteBufferAs[T]BufferR[B|L].

Classes in each group differ in method signatures, data types, keywords, operators, and other editing changes. Some of the classes have extra methods and/or attributes as compared to other classes in the same group.

A non-redundant, generic representation for groups of similar classes seems a viable approach to achieving a simpler representation of buffer classes. It is interesting to see why buffer classes could not be represented in a generic form.

Any solutions to unifying similarities must be considered in the context of other design goals developers must meet. Usability, conceptual clarity and good performance are important design goals for the Buffer library. In many situations, designers could introduce a new abstract class or a suitable design pattern to avoid repetitions. However, such a solution would compromise the above design goals, and therefore, was not implemented. Many similar classes or methods were replicated because of that.

Many similarities in buffer classes sparked from the fact that buffer features (Fig. 4) could not be implemented independently of each other in separate implementation units (e.g., class methods). Feature modularization, one of the goals of Feature-Oriented Programming [7,38], did not work for the Buffer library. Code fragments related to specific features appeared with many variants in different classes, depending on the context. Whenever such code could not be parameterized to unify the variant forms, and placed in some upper-level class for reuse via inheritance, similar code structures spread through classes.

Since JDK 1.5 includes generics, one could presume that type parameterization should have a role to play in unifying parametric differences among similar classes. However, generics have not been applied to unify similar buffer classes. Groups of classes that differ only in data type are obvious candidates for generics. There are three such groups comprising 21 classes, namely [T]Buffer, Heap[T]Buffer and Heap[T]BufferR. In each of these groups, classes corresponding to Byte and Char types differ in non-type parameters and are not generics-friendly. This leaves us with 15 generics-friendly classes whose unification with three generics eliminates 27% of code. There is, however, one problem with this solution. In Java, generic types cannot

```

/*Creates a new byte buffer containing a shared
subsequence of this buffer's content. */
public ByteBuffer slice() {
    int pos = this.position();
    int lim = this.limit();
    assert (pos <= lim);
    int rem = (pos <= lim ? lim - pos : 0);
    int off = (pos << 0);
    return new DirectByteBuffer(this, -1, 0, rem, rem, off);
}

```

**Fig. 6.** Method `slice()` recurring in 13 `Direct[T]Buffer[S/U]` classes

be primitive types such as `int` or `char`. This is a serious limitation, as one has to create corresponding wrapper classes just for the purpose of parameterization. Wrapper classes introduce extra complexity and hamper performance. Application of generics to 15 buffer classes is subject to this limitation.

Buffer classes and methods differ in parameters representing constants, keywords or algorithmic elements rather than data types. This happens when the impact of various features affects the same class or method. For example, method `slice()` (Fig. 6) recurs 13 times in all the `Direct[T]Buffer[S/U]` classes with small changes highlighted in bold. Generics are not meant to unify this kind of differences in classes. We found yet other cases of similar but generics-unfriendly classes and we refer the reader to further details of the generics solution (including code) to our case studies on XVCL Web site [45].

In summary, generics have a rather limited role to play in unifying similarity patterns that we find in practical situations such as we observed in the Buffer library. It is interesting to note that repetitions often occur across classes at the same level of inheritance hierarchy, as well as in classes at different levels of inheritance hierarchy. Programming languages do not have a proper mechanism to handle such variations at an adequate (that is a sufficiently small) granularity level. Therefore, the impact of a small variation (that on a program may not be proportional to the size of the variation).

### 3 XVCL Concepts

XVCL (XML-based Variant Configuration Language) provides a systematic treatment for generic design problems that cannot be easily solved using conventional techniques. In the reuse context, XVCL adds generic design and variability management capabilities to conventional component/architecture Product Line techniques.

XVCL technology includes a *language* that helps represent programs in generic, adaptable form, *methods* guiding project application of XVCL, and *tools*. XVCL Processor is an interpreter of the XVCL notation. The Processor automates derivation of custom, executable programs from their generic meta-level XVCL representation. XVCL Workbench is an eclipse-based plug-in with additional tools such as a static/dynamic analyzer, debugger and meta-level visualizer.

XVCL [45] is not yet another programming language. Developers still use conventional design techniques, programming languages and platforms to express the core of their program solution – user interfaces, business logic or databases. XVCL is applied together and in synergy with any base programming technology, to enhance its capabilities to define generic, adaptable, changeable and extendible software representations, as needed for effective reuse and evolution. We call it *mixed-strategy*.

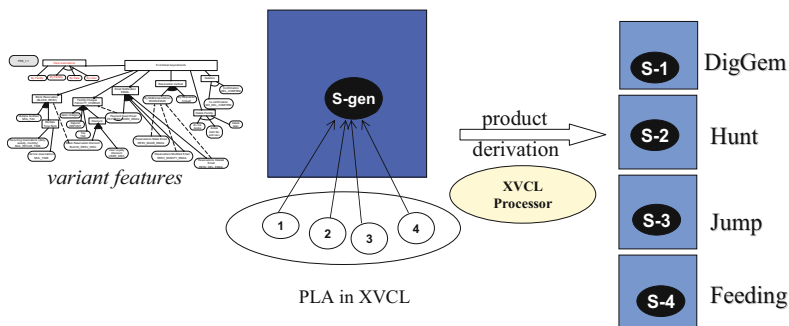
XVCL provides a mechanism for designing generic meta-level representations to unify groups of similar program structures of any kind and granularity. It also provides a change propagation mechanism to instantiate generic structures in multiple variant forms, as required in target programs (e.g., PL members). For example, any group of simple clones (e.g., **slice()** methods recurring in buffer classes) has a generic representation in XVCL; so does each of the seven groups of similar buffer classes ([T]Buffer, Heap[T]Buffer, etc.), and a group of DEMS structural clones (Fig. 3).

Product Line members typically display much similarity. In Fig. 7, S-1, S-2, S3, and S-4 is a similar program structure that recurs in four games in variant forms. A PLA, built with the help of XVCL, represents each such group of similar structures in a generic form (S-gen). At the same time, we also make a record of differences among instances of a program structure in different games (circles at the bottom numbered 1, 2, 3 and 4). This record as well as S-gen are formally expressed in XVCL. The XVCL Processor interprets specifications deriving custom instances of a programs structure required in different games.

Unification of similar program structures is done at all levels of software representations, from similar code fragments (such as class methods), to classes, components, and subsystems. At the end, we build a generic representation of PL members as a PLA from which custom products are derived.

Generic XVCL meta-components are called *x-frames*. A PLA built with XVCL is called an x-framework. Custom systems are derived from x-frames based on specifications of required customizations.

To represent in a generic form any similar program structures, we need powerful, unrestrictive parameterization, and refined mechanisms to separate commonalities from differences. We also need to build generic design solutions in a hierarchical way, with small-granularity structures (e.g., class methods) being building blocks of larger-granularity structures (e.g., classes). This will allow us to achieve reuse at as many levels as it is required.



**Fig. 7.** A PLA for RPGs and product derivation with XVCL Processor

## 4 XVCL by Example

We introduce XVCL mechanisms by means of a toy Product Line (PL). We use simplified XML-free XVCL notation and do not cover many XVCL features that are useful in practice, but not essential to understanding the essence of the approach. For full specifications of XVCL, we refer readers to XVCL Web site [45].

Consider a PL whose members are similar Java classes. Each class can print any number of messages. One such class `SavingsAccount` is shown in Fig. 8. The class name and the messages, variant features of our PL, are shown in bold in Fig. 8.

```
class SavingsAccount {
    public static void main(String[] args) {
        System.out.println("This is a bank account");
        System.out.println("Savings Account");
    }
}
```

Fig. 8. Class `SavingsAccount` printing two messages

An x-frame `Account` in Fig. 9 forms a PLA, and a SPeCification x-frame (SPC) describes how to derive class `SavingsAccount` from it .

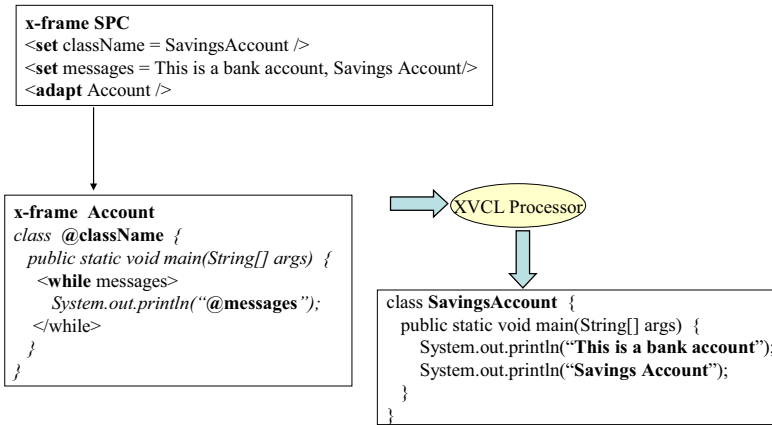


Fig. 9. Deriving class `SavingsAccount` from a generic x-frame `Account`

For readability, Java code in x-frames is shown in *italics*. The non-italics parts of the x-frame body are in XVCL. We highlight names of XVCL commands in **bold**.

XVCL variables ‘className’ and ‘messages’ are assigned values in <set> commands, in SPC. The value of variable ‘className’ is ‘SavingsAccount’. The value of variable ‘messages’ is a list of values, namely “This is a bank account” and “Savings Account”. We call ‘messages’ a multi-value variable.

```

class CurrentAccount {
    public static void main(String[] args) {
        System.out.println("This is a bank account");
        System.out.println("Current Account");
    }
}
class LoanAccount {
    public static void main(String[] args) {
        System.out.println("This is a bank account");
        System.out.println("Loan Account");
    }
}

```

**Fig. 10.** Classes CurrentAccount and LoanAccount

XVCL Processor interprets x-frames from the top to the bottom, emitting any non-XVCL text (Java code, in our case) to the output “as is”, and interpreting any XVCL commands found on the way.

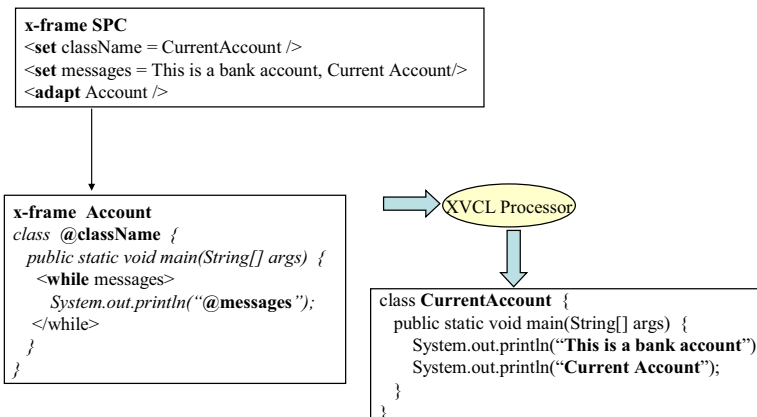
In our example, XVCL Processor starts processing with **SPC**, setting values of variables first, and then switching processing to x-frame **Account** (a class template), as instructed by **<adapt>** command (indicated by arrow between x-frames in Fig. 9).

In x-frame **Account**, ‘@className’ is a reference to variable ‘className’. XVCL Processor emits the current variable value, in this case ‘SavingsAccount’.

Loop command **<while>** is controlled by a multi-value variable ‘messages’. The *i*-th iteration of the loop uses the *i*-th value of the variable ‘messages’. In each iteration over the **<while>** body, XVCL Processor emits Java code to print a message.

We now wish to derive from the same x-frame two other classes, PL members, shown in Fig. 10.

With simple modifications of **SPC**, we derive class CurrentAccount (Fig. 11).



**Fig. 11.** Deriving class CurrentAccount a generic x-frame Account

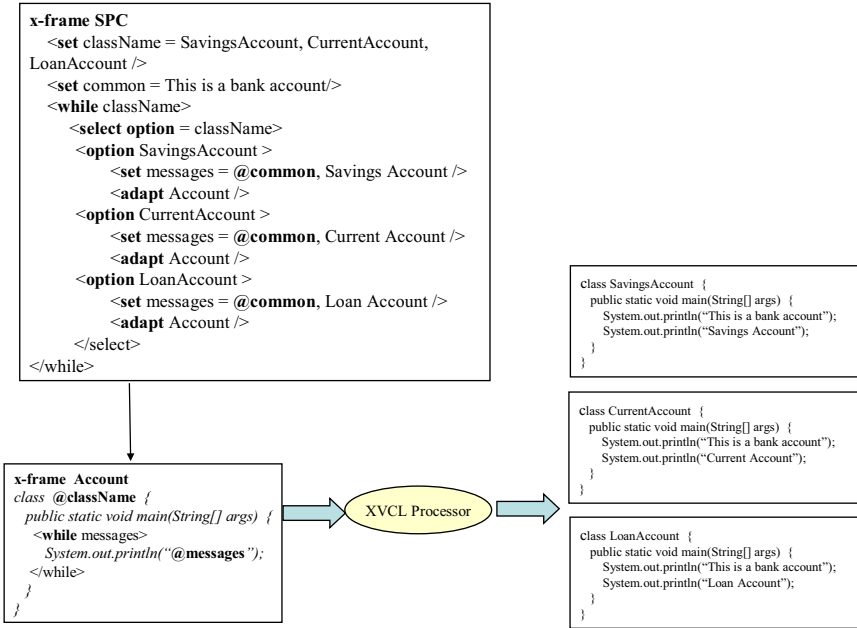


Fig. 12. Deriving three classes from x-frames

```

class FcAccount {
  public static void main(String[] args) {
    System.out.println("This is a bank account");
    System.out.println("Foreign Account");
    // extra messages for Foreign Account:
    System.out.println();
    System.out.println("Currency Swiss Francs");
  }
  // extra methods for FcAccount
  int convert () { ... }
  int interest () { ... }
}
  
```

Fig. 13. Class FcAccount printing extra messages

Derivation of all three classes is shown in Fig. 12. We define a common message in variable ‘common’ and then define messages specific to different classes in relevant <option>s of <select> command. In each <option>, multi-value variable ‘message’ is <set> to contain messages required for a given class, and x-frame Account is <adapt>ed accordingly.

In the final example, suppose that we also need a foreign currency account. A class named FcAccount requires two extra methods, convert() and interest(), in addition to

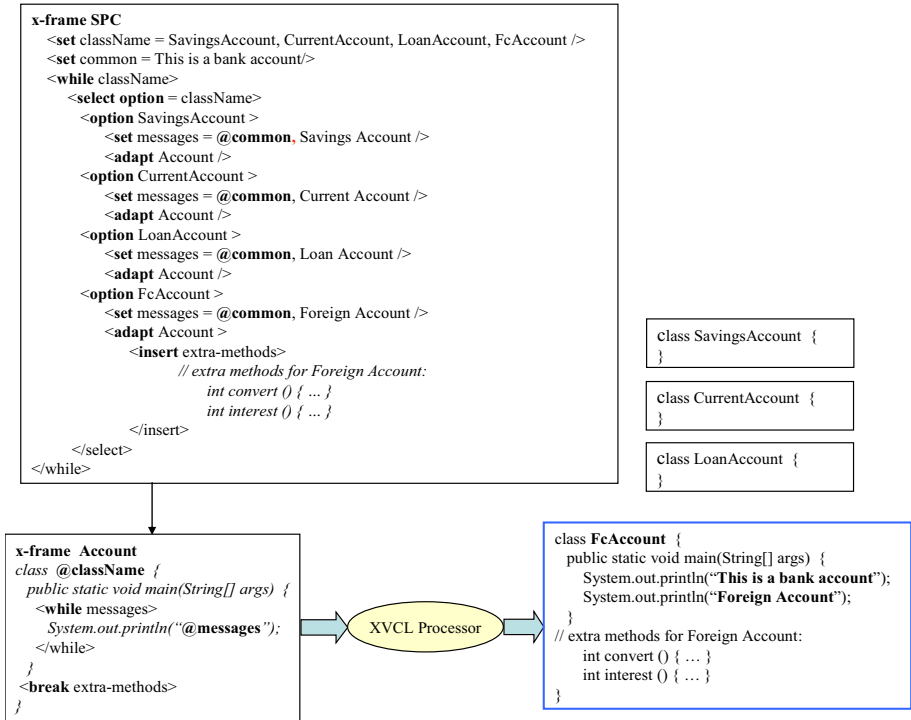


Fig. 14. Deriving four classes from x-frame Account

the methods defined in previous classes. This new requirement for class FcAccount was unexpected at the time when we designed x-frame **Account**. Such unexpected changes often happen in software, and techniques for software reuse and evolution should provide suitable mechanisms to handle it.

We use XVCL command **<insert>** into **<break>** to insert extra methods into class FcAccount (Fig. 14). XVCL **<insert>** plays a similar role to weaving aspect code in Aspect-Oriented Programming [32]. With **<insert>** command, we can modify x-frames at designated **<break>** points in arbitrary ways. Notice that **<break>** in x-frame **Account** allows us to extend any class with extra methods, if necessary. However, **<break>** does not affect classes that do not need extra methods. If not affected by **<insert>**, **<break extra-methods>** does not have any impact on classes derived from x-frame **Account**.

By now, the reader is already familiar with basic XVCL mechanisms. We summarize them below, adding some more details, not explained in the above examples.

XVCL variables and expressions provide a basic parameterization mechanism to make x-frames generic. XVCL **<set>** command assigns a value to a variable. Typically, names of program elements manipulated by XVCL, such as components, source files, classes, methods, data types, operators or algorithmic fragments, are represented by XVCL expressions, which references variable (e.g., @className) is the simplest form. Names of x-frames in **<adapt>** commands are often provided as

XVCL expressions rather than strings, allowing the actual name of an `<adapt>ed` x-frame to be determined during processing.

XVCL expressions are then instantiated by the XVCL Processor, according to the context. For example, class names and messages are represented by XVCL variables in the examples of Account classes.

XVCL variables accept a single value or a list of values. The latter are called multi-value variables.

Other than parameterization, XVCL variables control `<while>` loops and `<select>` structures, playing an important role in exercising the control over the processing of x-frames and the actual custom code that XVCL Processor emits during processing.

As variable values propagate across x-frames, variables can coordinate chains of all the customizations related to the same source of variation or change, that spans across multiple x-frames. XVCL variable scoping and propagation rules are important for achieving the overall goal of building generic, adaptable program representations. During processing of x-frames, values of variables propagate from an x-frame where the value of a variable is set, down to the lower-level x-frames. While each x-frame may set default values for its variables, values assigned to variables in higher-level x-frames take precedence over the locally assigned default values. In other words, once a value of variable is `<set>` in x-frame A, XVCL Processor ignores any subsequent `<set>` commands trying to assign value to that variable in x-frames `<adapt>ed` from A. Thanks to this overriding rule, x-frames become generic and adaptable, with potential for reuse in many contexts.

Other XVCL commands that help us design generic and adaptable x-frames include `<select>`, `<insert>` into `<break>` and `<while>`. We use `<select>` command to direct processing into one of the many pre-defined branches (called options), based on the value of a variable. With `<insert>` command, we can modify x-frames at designated `<break>` points in arbitrary ways. XVCL expressions, `<select>` `<insert>` into `<break>` are analogous to AOP's mechanism for weaving advices at specified join points. The difference is that XVCL allows us to modify x-frames in arbitrary ways, at any explicitly designated variation points.

A `<while>` command iterates over its body, with each iteration generating similar, but also different, program structures. A `<select>` command in the `<while>` loop allows us to define messages specific to Account classes in the example discussed in the last section.

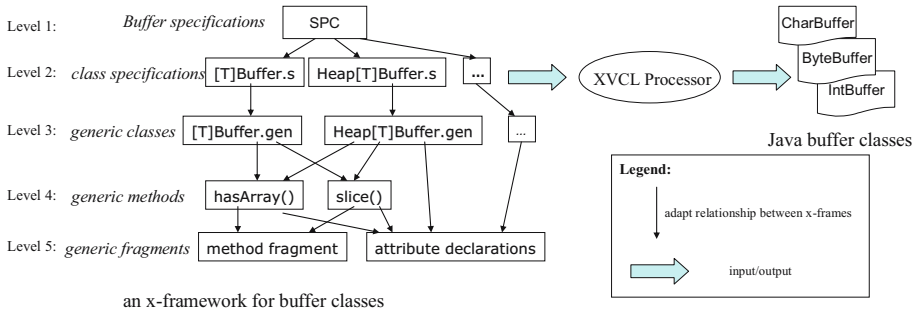
## 5 Buffer Library in Java/XVCL

Buffer library can be considered a special kind of a PL whose members are buffer classes. The overall solution to Buffer library PL in Java/XVCL is shown in Fig. 15. A PLA built with XVCL is called an x-framework.

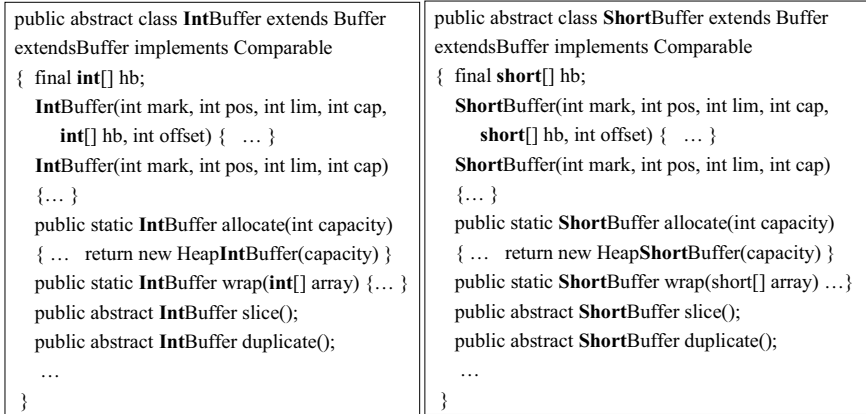
An arrow between two x-frames:  $X \rightarrow Y$  is read as "X adapts Y", meaning that X controls adaptation of Y. At Level 3, we have seven *generic class* x-frames, one for each of the seven groups of similar classes described in Section 2.2. Only two of them, namely **[T]Buffer.gen** and **Heap[T]Buffer.gen**, are shown in Fig. 15.

XVCL Processor derives all classes in group [T]Buffer from x-frame **[T]Buffer.gen**, based on specifications contained in specification x-frames **SPC** and





**Fig. 15.** A Java/XVCL x-framework for Buffer library



**Fig. 16.** Differences among IntBuffer and ShortBuffer

**[T]Buffer.s** (details to be explained). Classes in other groups are derived in a similar way from their respective generic and specification x-frames.

Each generic x-frame defines common part of classes in the respective group. Smaller granular generic building blocks for classes are defined below, at Level 4 (methods) and Level 5 (fragments of method implementation or attribute declaration sections). Therefore, lower-level generic components are composed, after possible adaptations, to construct required instances of higher-level generic components. Level 1 and 2 are specification x-frames – they tell the XVCL Processor how to generate specific components (buffer classes, in our case) from generic ones. Top-most x-frame **SPC** sets up global parameters and exercises the overall control over the generation process. Specifications of controls for each of the seven groups of similar classes are at Level 2.

The XVCL Processor interprets an x-framework starting from the **SPC**, traverses x-frames below, adapting visited x-frames and emitting buffer classes in each group one-by-one.

```

public abstract class TBuffer extends Buffer <T>
extendsBuffer implements Comparable
{ final T[] hb;
  TBuffer(int mark, int pos, int lim, int cap,
    T[] hb, int offset) { ... }
  TBuffer(int mark, int pos, int lim, int cap)
  {... }
  public static TBuffer allocate(int capacity)
  { ... return new HeapTBuffer <T> (capacity) }
  public static TBuffer wrap(T[] array) { ... }
  public abstract TBuffer slice();
  public abstract TBuffer duplicate();
  ...
}

```

Fig. 17. Generic class unifying five numeric [T]Buffer classes

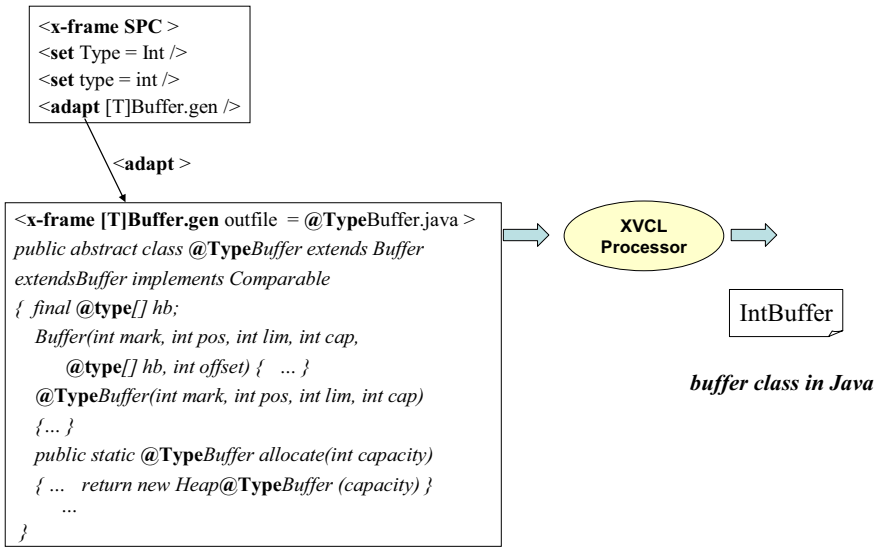
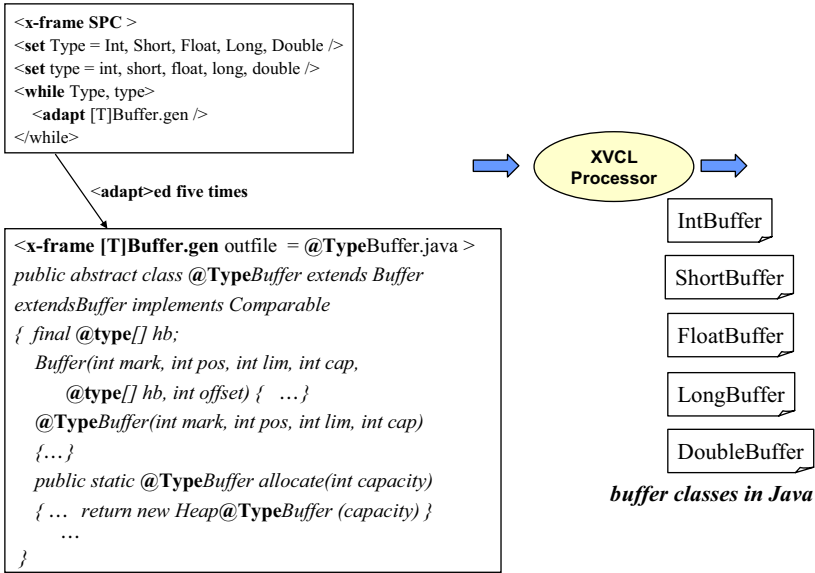


Fig. 18. Deriving class IntBuffer from x-frames

In the sections to follow, we show the steps in building a Java/XVCL representation for seven classes in the group [T]Buffer, namely IntBuffer, ShortBuffer, FloatBuffer, LongBuffer, DoubleBuffer, CharBufer and ByteBuffer.

### 5.1 Five Generics-Friendly Buffer Classes

Numeric type buffer classes differ one from another in type names only. Fig. 16 highlights in bold differences among IntBuffer and ShortBuffer.



**Fig. 19.** Deriving numeric buffer classes from x-frames

Such classes usually can be unified with type parameterization, called generics in Java or C# or templates in C++. A generic class is shown in Fig. 17.

Fig. 18 shows a generic x-frame **[T]Buffer.gen** parameterized by two XVCL variables, namely ‘Type’ and ‘type’. By setting variable values in **SPC**, we derive class **IntBuffer** from x-frame **[T]Buffer**. Attribute ‘outfile’ in x-frame **[T]Buffer.gen** defines the name of a file, **IntBuffer.java**, where we want XVCL Processor emit code for this class.

Fig. 19 shows derivation of all five numeric buffer classes from x-frame **[T]Buffer.gen**.

The reader should notice a fundamental difference between generics and XVCL: Generics are defined in a program and can be instantiated during program execution. On the other hand, in XVCL, all the classes are built in their concrete form before program runs. XVCL is used at the program construction time, not at runtime.

## 5.2 Classes CharBuffer and ByteBuffer

Fig. 20 shows some of the differences among numeric buffer classes and class **CharBuffer**. For **CharBuffer**, we must update ‘implements’ clause (the second line), re-define implementation of method **toString()**, and insert extra methods required in class **CharBuffer**, but not needed in numeric buffer classes.

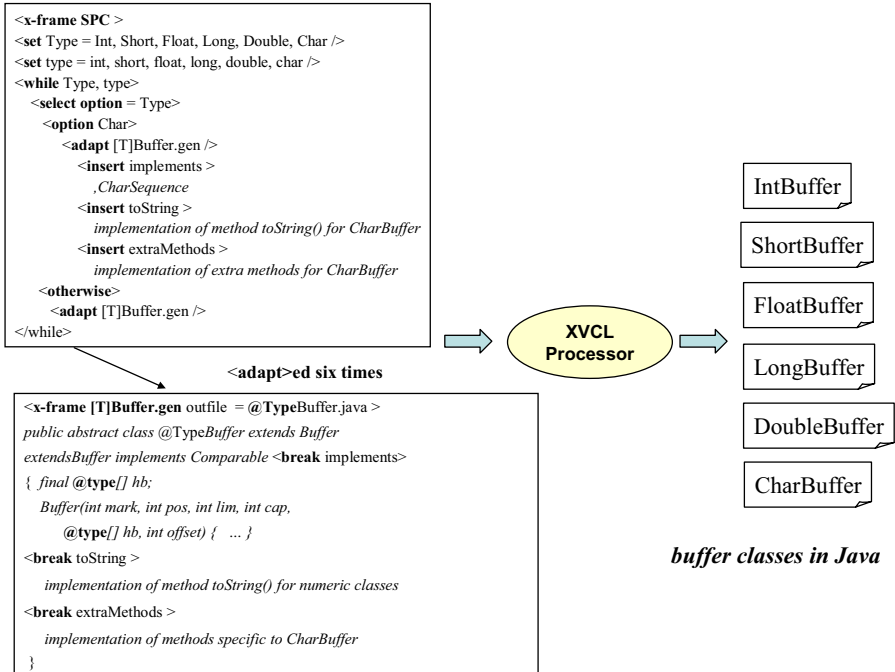
In x-frames of Fig. 21, **<option Char>** of **<select>** defines customizations required for class **CharBuffer**, but not needed in other classes. We use **<insert>** commands in the **<adapt>** body to update the ‘implements’ clause, to override the implementation of method **toString()** and to add extra methods. Notice that **<break toString>** in x-frame **[T]Buffer.gen** contains implementation of method **toString()** for all five numeric buffer classes as default. If no **<insert>** affects the **<break>**, the default

<pre>public abstract class <b>IntBuffer</b> extends Buffer extendsBuffer implements Comparable { final int[] hb; <b>IntBuffer</b>(int mark, int pos, int lim, int cap, int[] hb, int offset) { ... } <b>public String toString()</b> { ... } }</pre>	<pre>public abstract class <b>CharBuffer</b> extends Buffer extendsBuffer implements Comparable,<b>CharSequence</b> { final <b>char</b>[] hb; <b>CharBuffer</b>(int mark, int pos, int lim, int cap, <b>char</b>[] hb, int offset) { ... } public String toString() { <i>different implementation</i> } <b>many extra methods in Char Buffer:</b> public static CharBuffer wrap(CharSequence csq) { } etc. }</pre>
--	--

**Fig. 20.** Differences among classes IntBuffer and CharBuffer

contents of the **<break>** is processed as if there was no **<break>**. Any **<insert>** affecting the **<break>**, overrides the default contents of the **<break>**.

At the bottom of the **<select>** there is **<otherwise>** clause that caters for all the numeric buffer classes that are derived from x-frame **[T]Buffer.gen** as shown before, without any further customizations. **<otherwise>** is processed five times, in iterations when none of the other **<option>**s under **<select>** is processed, producing five numeric buffer classes.



**Fig. 21.** Deriving numeric buffer classes and class CharBuffer

```

<x-frame SPC >
<set Type = Int, Short, Float, Long, Double, Char, Byte />
<set type = int, short, float, long, double, char, byte />
<while Type>
  <select option = Type>
    <option Char>
      <adapt [T]Buffer.gen>
        customizations for CharBuffer
    <option Byte>
      <adapt [T]Buffer.gen>
        customizations for ByteBuffer
    <otherwise>
      <adapt [T]Buffer.gen/>
  </select>
</while>

```

Fig. 22. SPC to derive seven [T]Buffer classes

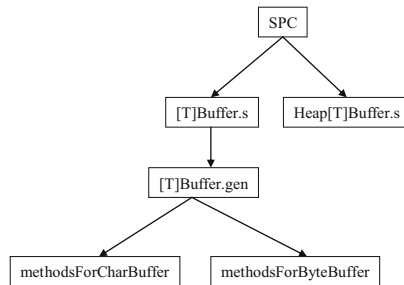


Fig. 23. An overview a Java/XVCL x-framework for Buffer library

Class ByteBuffer has yet other extra methods, not found in other [T]Buffer classes. The solution is the same as for extra methods in class CharBuffer, and the resulting SPC is shown in Fig. 22 (x-frame [T]Buffer.gen is the same as in Fig. 21).

An outline of the x-framework for the Buffer library is shown in Fig. 23, and its details in Fig. 24.

### 5.3 Evaluation of Java/XVCL Solution for the Buffer Library

The size of the Java/XVCL solution was 68% smaller than buffer classes in Java (in terms of lines of code, without blanks or comments). For the sake of fair comparison, we designed the Java/XVCL x-framework so that buffer classes generated from it were no different from the original classes. The physical size of a program is just one among many factors that collectively determine ease of understanding and changing a software system. Conceptual complexity is by far more important than the physical size. Therefore, we compared the number of conceptual elements in Java and Java/XVCL solutions. A conceptual element in a Java program is a class, method/

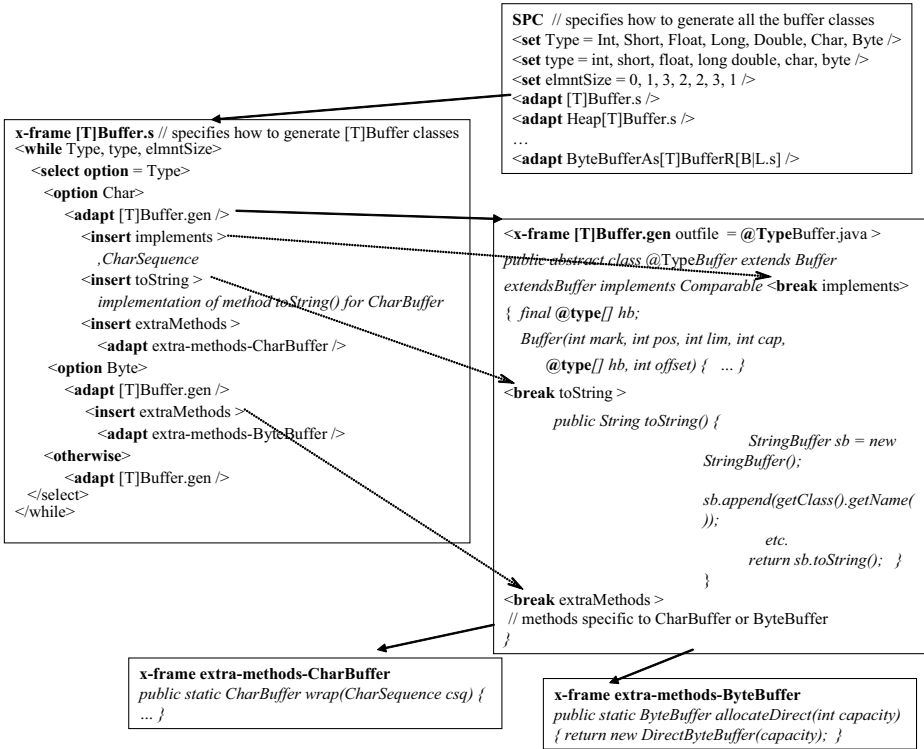


Fig. 24. A fragment of a Java/XVCL x-framework for Buffer library

constructor, declaration section or a fragment of method/constructor implementation that plays a role in the Buffer domain or in class design. Among classes at Level 1 (Fig. 5), there were 258 conceptual elements comprising 3,720 LOC (without blanks or comments) in the original Buffer classes, versus 79 conceptual elements comprising 1,400 LOC in the Java/XVCL representation. In the entire library, there were 1,385 conceptual elements comprising 6,719 LOC in the original classes, versus 324 conceptual elements comprising 2,080 LOC in the Java/XVCL representation.

This contraction of the solution space achieved by XVCL was a consequence of representing each of the important similarity patterns in a unique generic form.

Other than reducing the physical size and conceptual complexity, the XVCL solution also emphasized important relationships among program elements that matter to programmers who try to understand and modify the program. Due to genericity, instead of dealing with each class separately from others, we could understand classes in groups such as [T]Buffer or Heap[T]Buffer. We could see exact similarities and differences among specific classes in a group. This information helps in reusing existing classes when designing new buffer classes. It also reduces ripple effects and the risk of update anomalies, simplifying changes: If we want to change one class, we can check if the change also affects other similar classes. If we want to change a class method, we can analyze the impact of change on all the classes that use that method in the same or similar form.

The above relationships are implicit in the Java buffer classes (as well as in most of other conventional programs). A programmer must recover them whenever a program must be understood for change.

To further support claims of easier changeability of the XVCL solution, we extended the Buffer library with a new type of buffer element – Complex. Then, we compared the effort involved in changing each of the two solutions, Java classes and Java/XVCL representation. Many classes must be implemented to address the Complex element type, but in this experiment we concentrated only on three of them, namely ComplexBuffer, HeapComplexBuffer and HeapComplexBufferR. In Java, class ComplexBuffer could be implemented based on the class IntBuffer, with 25 modifications that could be automated by an editing tool, and 17 modifications that had to be done manually. On the other hand, in the Java/XVCL representation, all the changes had to be done manually, but only 5 modifications were required. To implement class HeapComplexBuffer, we needed 21 “automatic” and 10 manual modifications in Java, versus 3 manual modifications in the Java/XVCL. To implement class HeapComplexBufferR, we needed 16 “automatic” and 5 manual modifications in Java, versus 5 manual modifications in Java/XVCL.

## 6 Evaluation of XVCL

Applying a new technique does not come for free, it entails costs and involves trade-offs. Therefore, to be attractive, a new technique must solve some important engineering problems, providing benefits that outweigh the cost. In this section, we evaluate trade-offs involved in project application of XVCL.

We summarize experiences with XVCL first. We applied XVCL to building Product Lines in a range of application domains (business systems, Web Portals, command and control), programming languages (Java, C++, C#, ASP, PHP) and platforms (JEE, .NET, Unix, Windows) [3,24,26,37,46,48,49]. We typically found 50%-90% of code contained in similar program structures. The reasons that triggered repetitions were often similar to what we observed in the Buffer library. The logical structure of XVCL solutions was similar to the one we developed for the Buffer library, but as we were dealing with more complex program situations, we had to decompose x-frameworks into more layers than in the Buffer library.

### 6.1 Strengths

In XVCL, we represent each of the important similarity patterns in a unique generic, but adaptable form, along with the information necessary to obtain its instances – specific program structures. Such generic software representation offers some interesting engineering benefits. In particular, it (1) reduces the code size (in our studies, by 50-90%), (2) contains less number of conceptual elements than the number of conceptual elements in the concrete program, (3) bridges the gap between domain concepts and code, as similarity patterns often represent domain-specific abstractions, (4) enhances the conceptual integrity of the design, which Brooks calls “the most important consideration in system design” [10], and (5) in addition to program code, contains information that is helpful in program understanding, evolution and reuse, such as a record of

similarities/differences among program structures, and traces of how various features affect program components.

Generic structures built with XVCL can unify similarity patterns of any granularity and type – from a subsystem, to pattern of components, to component, to class and to program statement in class implementation. We can specify arbitrary differences among similar program structures. Many similarity patterns crosscut system layers and involve many components. Such similarities offer reuse opportunities that are usually missed by conventional architecture-centric and component-based approaches to reuse. XVCL exploits these extra reuse opportunities, often extending the scope and rates of reuse achievable by means of conventional techniques.

We can benefit from non-redundancy at the level of XVCL representation, and still keep clones in executable programs (as it is often desirable or unavoidable for the reasons we discussed in this paper, and as observed by others [13][33]).

A programmer can intervene in any detail of the transition from the generic structures to concrete programs. This allows XVCL to escape the problem of the techniques based on abstractions disconnected from the base code, which are found difficult to work with by maintenance programmers [13].

From the XVCL perspective, there is no distinction between maintainability (understood as the ease of changing software) and reusability. Both are achieved by means of generic design, with provisions for fine control over instantiating generic structures, matching practical needs of software reuse and evolution [24].

## 6.2 Weaknesses and How We Address Them

Despite potential benefits, applying XVCL also induces certain complexities. Designing generic, reusable and maintainable solutions is always a challenge which requires more talent, skill and time than building a concrete program. A concrete program is only a prerequisite for applying XVCL.

An XVCL solution is expressed at two inter-mixed levels, in base programming language(s) and wrapped in XVCL meta-structures. Thinking in terms of a mixed-level representation such as Java/XVCL or JEE/XVCL is different from thinking in terms of conventional program. This creates extra difficulties. However, we must keep in mind that an XVCL solution contains much useful information for evolution and reuse, in addition to information about the program(s) itself. We do not apply XVCL for quick gains during development, but for long-term gains. XVCL targets at long-lived programs that undergo extensive evolutionary changes, or need be tailored to needs of multiple customers.

As we relax the coupling between the parameterization mechanism and the rules (syntax and semantics) of the underlying programming language, the power of the parameterization mechanism increases. For example, with C++ templates we can unify a wider class of variations than with Java generics. At the end of this spectrum, there are techniques that manipulate program structures with no regard to language rules. XVCL is such a technique. By separating genericity issues from the core programming constructs, we can address genericity concerns without compromising runtime properties of programs. But as we move towards less restrictive parameterization mechanisms, we also decrease type-safety of a program representation. Therefore, there are important trade-offs to consider.



Specification, analysis and validation methods that work for concrete programs are not directly applicable to XVCL program representations. Before such methods are invented, skillful design, informal documentation and tools can mitigate problems to some extent. An x-framework can be organized based on the usual principles of the abstraction and separation of concerns. “Good design” can minimize the scope of an x-framework that has to be analyzed at any time when an x-framework is modified or reused. As lower levels x-frames become stable and reliable over time, potential errors tend to be located only in top-most, context-specific and still fragile x-frames.

The feedback from our industry partner indicates that, in practice, the benefit of enhanced reusability and maintainability may outweigh the cost of the added complexity [37]: the learning curve and development effort of an XVCL solution can be reasonable even for large programs (provided that an XVCL expert is also familiar with an application domain and program itself). At the same time, the return on investment may be quick and substantial.

Could we do better by raising the level of abstraction of XVCL? We consider the current form of XVCL an *assembly language for generic design*. XVCL contains the minimum constructs to specify any generic structures along with adaptation changes required to obtain their instances. Direct articulation is the source of XVCL’s expressive power. However, specifications can get tedious and complex. At this point, we do not know how to raise the level of abstraction without compromising the expressive

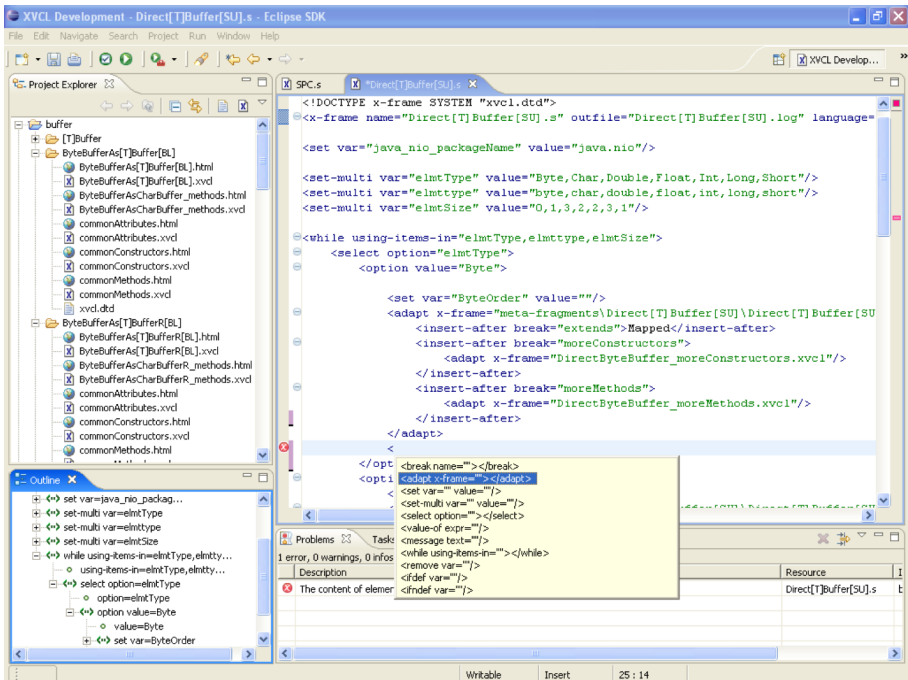


Fig. 25. A snapshot of XVCL Workbench

power of the XVCL mechanism. In the future, we hope to discover abstractions that will allow us to define higher-level forms of XVCL, equally expressive but free of current pitfalls.

Tools may considerably simplify application of XVCL. We are developing an IDE for XVCL called XVCL Workbench that helps in editing, visualizing, debugging and static/dynamic analysis of x-frameworks. The upper left-hand-side Project Explorer window (Fig. 25) shows x-frameworks. The Outline window below shows x-frame structure in XML-free format. The upper right-hand-side window shows an x-frame in raw XML format. A context-sensitive help popup menu shows XML commands valid at a given editing point. The Workbench reports errors and warnings in the lower right-hand-side window. A developer can examine static structure of an x-framework or only x-frames visited by the Processor for a given SPC. In the future, a developer will be able to ask queries about properties of x-frames, and run the Processor in a debugging mode. XVCL Workbench is implemented as a plug-in to the Eclipse platform.

XVCL affects conventional development processes in a similar way as any systematic reuse strategy does. Changing the way people think about software, changing existing processes and company structures has always been a challenge. At this point, we know how XVCL can raise productivity of small teams of highly-skilled software developers. We are yet to learn what it takes to inject XVCL methods into large-scale team-based industrial development processes.

## 7 Related Work

We contrast XVCL with other techniques that target the similar goals.

XVCL has its roots in Frame Technology™ by Netron, Inc [4]. A number of frame-based systems have been implemented in both industrial and academic institutions [18]. We believe any system based on frame principles can achieve similar engineering benefits as XVCL, independently of a specific syntactic representation that different systems may use.

Frame Technology™ has been extensively applied to maintain multi-million-line COBOL-based information systems and to build reuse frameworks in companies [4]. An independent assessment by QSM Associates, Inc. showed that frames could achieve up to 90% reuse, reduce project costs by over 84% and their time-to-market by 70%, when compared to industry norms [4]. We are in the lucky situation that the basic principles of XVCL have been already tested in practice, though in a different setting than ours. Our contribution is that we refined frame concepts into a general-purpose technique of XVCL. We also demonstrated that XVCL can enhance modern programming paradigms in areas of maintainability and reusability.

Macro-processors work on the principle of code expansion, and so does XVCL. However, from the point of view of detailed mechanisms and engineering goals, there are more differences among macro-processors and XVCL than similarities. Macros work in local scope, only at the implementation level, which causes well-known problems when trying to tackle more complex change situations with macros [30]. XVCL is full-fledged technique for taking advantage of software similarities and for controlling changes, from software architecture down to every detail of code. We believe it is

difficult to solve the problems we discussed in the paper with macros and other low level program manipulation techniques such as scripting languages, providing engineering qualities comparable to those we demonstrated with XVCL.

Software Configuration Management (SCM) systems [44] have been applied to handle variant features in software. Rather than unifying similarity patterns induced by features, for each legal combination of features, an SCM system maintains a separate component version. Thousands of component versions arise in industrial applications of product lines, creating problems for effective reuse [15]. It is difficult to synthesize a comprehensible view of domain similarities and differences from multiple component versions. In XVCL, we avoid this problem by designing generics components, and maintaining a record (both human-readable and executable by the XVCL Processor) of how to generate concrete components in required variant forms.

Powerful domain-specific solutions can be built by formalizing the domain knowledge, and using generation techniques [42] to produce custom programs in a domain. Advancements in modeling and generation techniques led to recent interest in Model-Driven Engineering (MDE) [40], where multiple, inter-related models are used to express domain-specific abstractions. Models are used for analysis, validation (via model checking), and code generation. Platforms such as Microsoft Visual Studio™ and Eclipse™ support generation of source code using domain-specific diagrammatic notations.

This is in contrast with XVCL which is an application domain- and programming language-independent technique. There is no concept of DSL in XVCL. XVCL targets the similarity patterns in any application domain. Such similarity patterns often represent important domain concepts – this observation is one of important contributions of the research described in this paper. Therefore, many XVCL structures (x-frames) map into domain concepts. However, in XVCL approach, the very philosophy of how to arrive at these structures and how to represent them is fundamentally different from domain-specific generators. Rather than extending the language towards domain-specific abstractions, in XVCL we focus on identifying similarity patterns, in both top-down and bottom-up ways, and unifying differences among instances of such patterns with generic representations.

XVCL provides simple yet powerful means to achieve that. While we do not come up with all possible domain concepts, we usually address domain concepts that are of practical engineering importance. XVCL generic meta-level structures show realization of such abstractions in the design/implementation solution space. Bridging the gap between domain concepts and their implementation is a by-product of the process of similarity analysis and unification. From this perspective, XVCL can be viewed as a domain-independent technique for capturing some of the domain-specific abstractions. Therefore, even though there is no direct competition between XVCL and domain-specific generation approaches, and the principle of the approaches and technical means are different, there is a certain overlap in goals achieved by domain-specific generators and domain-independent XVCL.

In contrast to generation approaches, XVCL offers programming language-neutral mechanisms specifically dedicated to unifying arbitrary differences among similar program structures whose unification is deemed to be useful. As such, XVCL's principle of operation does not rely on the underlying language syntax or semantics, or even knowledge of what they are. XVCL does code expansion at arbitrary program points, according to pre-defined “composition with adaptation” rules. The expansion points,

meta-level structures (x-frames) that are subject of “composition with adaptation”, forms of their parameterization, as well as concrete program structures that result from expansion are not constrained by the rules of the underlying programming language.

We believe the strength of generators lies in their ability to hide a part of program complexity from a programmer by encoding application domain knowledge, rather than in providing general means for unifying similarity patterns, which is one of the prime goals of XVCL. We are not aware of any study that would demonstrate the feasibility of solving problems as discussed in this paper by means of systems based on domain-specific generators or language-specific transformations.

Dijkstra introduced a principle of separation of concerns to the software domain in early 1980's [16]. Recently, there've been a number of attempts to bring separation of concerns from the concept down to the design and implementation levels. Aspect-Oriented Programming (AOP) [32], Multi-Dimensional Separation of Concerns (MDSOC) from IBM [43], and Feature-Oriented Programming (FOP) [7,38] are among most widely published such techniques. Separation of concerns helps in maintainability, long-term evolution, and is also supportive to building more generic, reusable software. Though we did not come across applications of techniques based on separation of concern to unify software similarities such as we discussed in this paper, the very principle and techniques that help in its realization are most relevant to the theme of this paper.

FOP [38] applies separation of concerns principle in an attempt to modularize features, and then provides a mechanism for composing features into a base program. Mixin is the most common technique for feature composition. AHEAD [7] is a well-known and the most advanced realization of FOP concepts. The premise of AHEAD is that features can be modeled separately one from another, and programs can be constructed, evolved and reused by feature refinements defined as mathematical functions. Refinements can add, override or extend data declarations and methods of classes. A combination of features a given program implements is elegantly described by hierarchical algebraic equations in a GenVoca grammar [6]. While the concept is very appealing, its realization and scalability is a challenge. Program features tend to have delocalized, diverse and highly irregular impact on program structures. Such features may not fit into the above model.

In AOP, various computational aspects are programmed separately and weaved into the base of conventional program modules of primary decomposition (e.g., classes). Aspect code is weaved into program modules at join points that are specified in a descriptive way. AOP can simply and elegantly separate a range of programming aspects such as synchronization, persistence, security transaction management, or authentication/authorization. Due to such separation, aspects can be easily modified and also added or deleted to/from program modules, which automatically become more generic and reusable in different contexts.

The trust of the MDSOC approach [43] is separation of concerns to overcome a “tyranny of a dominant decomposition” of programs into functional modules. Hyperslices are meta-level abstractions that encapsulate specific concerns and can be composed in various configurations to form custom programs. Hyperslices are written in the underlying programming language and can be composed by merging or overriding program units by name and in many other ways. Compositions yield programs with

modified or extended behavior. Unlike in AOP, it is typical for hyperslices to represent functional units.

XVCL's mechanisms cater for both generic design and separation of concerns [47]. Like AOP, MDSOC or FOP, XVCL offers a mechanism to define alternative program decompositions at the meta-level. While groups of inter-related x-frames often correspond to concerns, analysis of similarity patterns and design of generic XVCL representations unifying similarity patterns plays a driving role in the process of developing an XVCL solution. XVCL construct `<while>` facilitates generation of multiple custom program structures from their generic representation. `<while>` does not have a counterpart in generative techniques based on the separation of concerns principle only.

At the level of actual mechanisms, unlike in other approaches, XVCL's compositions (a counterpart of weaving aspect code in AOP) are defined in operational way and take place at designated program points marked with `<adapt>`, `<break>` and other XVCL commands. Concerns encapsulated in x-frames, in areas where separation of concerns with XVCL is feasible, are unconstrained in the sense that they may overlap one with another or form concern hierarchies, as one concern may contain other concerns. XVCL's concerns can be parameterized with XVCL commands, which further enhances programmer's ability to define variations in code at any level of granularity that is required, from a subsystem or component, to a single program statement.

We believe each of the discussed techniques has its unique strengths and weaknesses: For different types of software domains and engineering goals, either AOP, MDSOC, FOP or XVCL may yield the simplest, most elegant and useful solution.

## 8 Conclusion

Conventional reuse is based on component reuse. We described a technique called XVCL (XML-based Variant Configuration Language) that is based on reuse of any structural similarities. Similar programs structures are captured in generic form at the meta-level. XVCL Processor derives custom instances of program structures from their generic representation. We develop, reuse and evolve software at the level of XVCL meta-structures, deriving specific, executable programs from it. Lab studies and industrial applications of XVCL show that reuse of structural similarities extends the benefits of conventional component reuse. On average, we raise reuse rates and productivity by 60-90%, reducing cognitive program complexity and maintenance effort by similar rates. The approach scales to systems of any size. The benefits are proportional to system size and to the extent of repetitions present in a system. The main application of XVCL is in building Product Line Architectures for reuse.

Adopting a new technique always brings overheads and XVCL is no different in this respect. We evaluated trade-offs involved in applying XVCL. The ultimate test for new techniques is industrial practice. The initial feedback from our industry partner STEE who applied XVCL in two projects indicates that the benefits of enhanced reusability and maintainability outweigh the cost of the added complexity.

Methodological guidelines and tool support for applying XVCL, as well as scaling XVCL from small teams of experts to larger team-based projects is the main challenge and the subject of our on-going work. We continue studies of a software similarity phenomenon, addressing issues such as repetitions induced by the underlying programming language and design technique. We plan to study formal properties of

XVCL program representation to come up with suitable specification and verification methods for XVCL solutions.

In the context of large programs, purely manual analysis to find similarity patterns is too laborious to be practical. We implemented a Clone Minder [1], a tool that automates the search for clones as candidates for generic XVCL representations. Clone Miner extends capabilities of clone detection tools such as Duploc [17] or CCFinder [28] from simple clones (code fragments) to design-level similarity patterns.

Large software systems today comprise tens of millions of LOC, with thousands of inter-related components (MS Windows approaches 100 million LOC). Ultra-Large-Scale systems will comprise of billions lines of code [35]. Even with much more successful forms of componentization that we have today, at the level of concrete programs we are bound to be exposed to the complexity of validating and maintaining software proportional to a system size. Exploiting potentials hidden in similarity patterns opens a pragmatic way to reduce this complexity by the rates proportional to the rates of similarities a system exhibits. XVCL approach described in this paper is an attempt to do that.

XVCL addresses design issues that are poorly supported by today's programming paradigms. We believe the full potentials of this simple yet powerful approach have yet to be discovered.

## Acknowledgements

Thanks are due to numerous students at National University of Singapore who participated in various projects. Their names appear as co-authors of publications cited in this paper. Collaborations with Paul Bassett and Ulf Pettersson contributed a lot to the results and interpretations described in this paper. This research was supported by National University of Singapore Research Grant R-252-000-239-112.

## References

1. Basit, A.H., Jarzabek, S.: Detecting Higher-level Similarity Patterns in Programs. In: ESEC-FSE 2005, European Soft. Eng. Conf. and ACM Symp. on the Foundations of Soft. Eng., Lisbon, pp. 156–165 (September 2005)
2. Basit, H.A., Rajapakse, D.C., Jarzabek, S.: Beyond Generics: Meta-Level Parameterization For Effective Generic Programming. In: Proc. 17th Int. Conf. on Software Engineering and Knowledge Engineering, SEKE 2005, Taipei (July 2005)
3. Basit, H.A., Rajapakse, D.C., Jarzabek, S.: Beyond Templates: a Study of Clones in the STL and Some General Implications. In: Proc. Int. Conf. Software Engineering, ICSE 2005, St. Louis, May 2005, pp. 451–459 (2005)
4. Bassett, P.: Framing software reuse - lessons from real world. Yourdon Press, Prentice Hall, Englewood Cliffs (1997)
5. Batory, D., Singhai, V., Sirkin, M., Thomas, J.: Scalable software libraries. In: ACM SIGSOFT 1993: Symp. on the Foundations of Software Engineering, Los Angeles, California, pp. 191–199 (December 1993)
6. Batory, D., O'Malley, S.: The Design and Implementation of Hierarchical Software Systems with Reusable Components. ACM Trans. on Software Engineering and Methodology 1(4), 355–398 (1992)

7. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. In: Proc. Int. Conf. on Software Engineering, ICSE 2003, Portland, Oregon, pp. 187–197 (May 2003)
8. Baxter, I., Yahin, A., Moura, L., Sant’Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: Proc. Int. Conf. on Software Maintenance, pp. 368–377 (1998)
9. Biggerstaff, T.: The library scaling problem and the limits of concrete component reuse. In: 3rd Int. Conf. on Software Reuse, ICSR 1994, pp. 102–109 (1994)
10. Brooks, P.B.: *The Mythical Man-Month*. Addison-Wesley, Reading (1995)
11. Brooks, F.P.: No Silver Bullet, *Computer Magazine* (April 1986)
12. Clements, P., Northrop, L.: *Software Product Lines: Practices and Patterns*. Addison-Wesley, Reading (2002)
13. Cordy, J.R.: Comprehending Reality: Practical Challenges to Software Maintenance Automation. In: Proc. 11th IEEE Intl. Workshop on Program Comprehension (IWPC 2003), pp. 196–206 (2003)
14. Czarnecki, K., Eisenecker, U.: *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading (2000)
15. Deelstra, S., Sinnema, M., Bosch, J.: Experiences in Software Product Families: Problems and Issues during Product Derivation. In: Nord, R.L. (ed.) SPLC 2004. LNCS, vol. 3154, pp. 165–182. Springer, Heidelberg (2004)
16. Dijkstra, E.W.: On the role of scientific thought, *Selected Writings on Computing: A Personal Perspective*, pp. 60–66. Springer, New York (1982)
17. Ducasse, S., Rieger, M., Demeyer, S.: A language independent approach for detecting duplicated code. In: Int. Conference on Software Maintenance, ICSM 1999, Oxford, UK, pp. 109–118 (September 1999)
18. Emrich, M.: *Generative Programming Using Frame Technology*, Diploma Thesis, University of Applied Sciences Kaiserslautern, Department of Computer Science, and Micro-System Engineering, 29 (July 2003)
19. Fowler, M.: *Analysis Patterns: Reusable Object Models*. Addison-Wesley, Reading (1997)
20. Fowler, M.: *Refactoring - improving the design of existing code*. Addison-Wesley, Reading (1999)
21. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading (1995)
22. Garcia, R., et al.: A Comparative Study of Language Support for Generic Programming. In: Proc. 18th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications, pp. 115–134 (2003)
23. Goguen, J.A.: Parameterized Programming. *IEEE Trans. on Software Engineering* SE-10(5), 528–543 (1984)
24. Jarzabek, S.: *Effective Software Maintenance and Evolution: Reused-based Approach*. CRC Press, Taylor and Francis (2007)
25. Jarzabek, S.: Genericity - a Missing in Action Key to Software Simplification and Reuse. In: 13th Asia-Pacific Soft. Eng. Conference, APSEC 2006, Bangalore, India, December 6–8, pp. 293–300 (2006)
26. Jarzabek, S., Li, S.: Eliminating Redundancies with a Composition with Adaptation Meta-programming Technique. In: Proc. ESEC-FSE 2003, European Soft. Eng. Conf. and ACM Symp. on the Foundations of Soft. Eng., Helsinki, September 2005, pp. 237–246 (2005); extended version: Jarzabek, S., Li, S.: Unifying clones with a generative programming technique: a case study. *Journal of Software Maintenance and Evolution: Research and Practice* 18(4), 267–292 (2006)

27. Jensen, P.: Experiences with Product Line Development of Multi-Discipline Analysis Software at Overwatch Textron Systems. In: 11th Int. Software Product Line Conference, SPLC 2007, pp. 35–43 (September 2007)
28. Kamiya, T., Kusumoto, S., Inoue, K.: CCFinder: A multi-linguistic token based code clone detection system for large scale source code. *IEEE Trans. Software Engineering* 28(7), 654–670 (2002)
29. Kang, K., et al.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, CMU, Pittsburgh (November 1990)
30. Karhinen, A., Ran, A., Tallgren, T.: Configuring designs for reuse, International Conference on Software Engineering. In: ICSE 1997, Boston, MA, pp. 701–710 (1997)
31. Kapsner, C., Godfrey, M.W.: Cloning Considered Harmful Considered Harmful. In: Proc. 13th Working Conference on Reverse Engineering, pp. 19–28 (2006)
32. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
33. Kim, M., Sazawai, V., Notkin, D., Murphy, G.: An Ethnographic Study of Code Clone Genealogies. In: ESEC-FSE 2005, European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, Lisbon, pp. 187–196. ACM Press, New York (2005)
34. Mayrand, J., Leblanc, C., Merlo, E.: Experiment on the automatic detection of function clones in a software system using metrics. In: In Proc. Intl. Conference on Software Maintenance (ICSM 1996), pp. 244–254 (1996)
35. Northrop, L.: Ultra-Large Scale Systems: The Software Challenge of the Future, Software Engineering Institute (June 2006) ISBN 0-978656-0-7
36. Parnas, D.: On the Criteria To Be Used in Decomposing Software into Modules. *Communications of the ACM* 15(12), 1053–1058 (1972)
37. Pettersson, U., Jarzabek, S.: An Industrial Application of a Reuse Technique to a Web Portal Product Line. In: ESEC-FSE 2005, European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, Lisbon, [34], pp. 326–335. ACM Press, New York (2005)
38. Proofer, C.: Feature-Oriented Programming: A Fresh Look at Objects. In: Proc. Europe. Conf. Object-Oriented Programming (1997)
39. Rajapakse, D.C., Jarzabek, S.: Using Server Pages to Unify Clones in Web Applications: A Trade-off Analysis. In: Int. Conf. Software Engineering, ICSE 2007, Minneapolis, USA (May 2007)
40. Schmidt, D.: Model-Driven Engineering. *IEEE Computer*, 25–31 (February 2006)
41. SGI STL, <http://www.sgi.com/tech/stl/>
42. Smaragdakis, Y., Batory, D.: Application generators. In: Webster, J. (ed.) *Software Engineering volume of the Encyclopedia of Electrical and Electronics Engineering*. John Wiley and Sons, Chichester (2000)
43. Tarr, P., Ossher, H., Harrison, W., Sutton, S.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: Proc. International Conference on Software Engineering, ICSE 1999, Los Angeles, pp. 107–119 (1999)
44. Tichy, W.: Tools for Software Configuration Management. In: Proc. Int. Workshop on Software Configuration Management, pp. 1–20. Teubner, Grassau (1988)
45. XVCL (XML-based Variant Configuration Language) method and tool for managing software changes during evolution and reuse, <http://xvcl.comp.nus.edu.sg>



46. Zhang, H., Jarzabek, S.: A Mechanism for Handling Variants in Software Product Lines. special issue on Software Variability Management, *Science of Computer Programming* 53(3), 255–436 (2004)
47. Zhang, H.Y., Jarzabek, S., Soe, M.S.: XVCL Approach to Separating Concerns in Product Family Assets. In: Bosch, J. (ed.) *GCSE 2001*. LNCS, vol. 2186, pp. 36–47. Springer, Heidelberg (2001)
48. Zhang, W., Jarzabek, S.: Reuse without Compromising Performance: Experience from RPG Software Product Line for Mobile Devices. In: Obbink, H., Pohl, K. (eds.) *SPLC 2005*. LNCS, vol. 3714, pp. 57–69. Springer, Heidelberg (2005)
49. Yang, J., Jarzabek, S.: Applying a Generative Technique for Enhanced Reuse on J2EE Platform. In: Glück, R., Lowry, M. (eds.) *GPCE 2005*. LNCS, vol. 3676, pp. 237–255. Springer, Heidelberg (2005)

# .QL: Object-Oriented Queries Made Easy

Oege de Moor, Damien Sereni, Mathieu Verbaere, Elnar Hajiyev,  
Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, and Julian Tibble

Semmlé Limited

**Abstract.** These notes are an introduction to .QL, an object-oriented query language for any type of structured data. We illustrate the use of .QL in assessing software quality, namely to find bugs, to compute metrics and to enforce coding conventions. The class mechanism of .QL is discussed in depth, and we demonstrate how it can be used to build libraries of reusable queries.

## 1 Introduction

Software quality can be assessed and improved by computing metrics, finding common bugs, checking style rules and enforcing coding conventions that are specific to an API. Many tools for these tasks are however awkward to apply in practice: they often detract from the main task in hand. Above all, it is tough to customise metrics and rules to one’s own codebase, and yet that is where the greatest benefit lies.

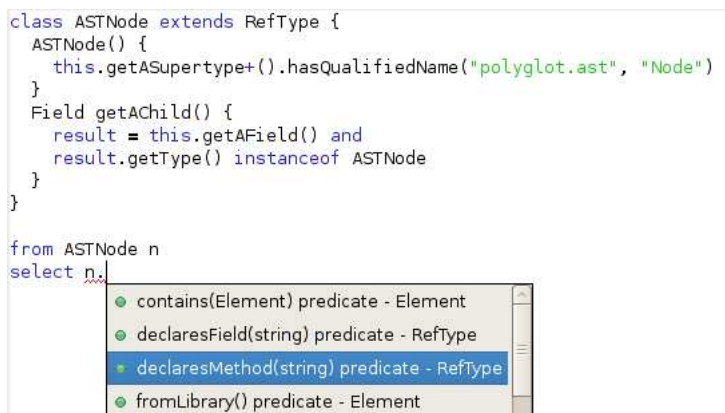
These lectures present a new approach, where all these tasks related to software quality are phrased as *queries* over a relational representation of the code base. Furthermore, the language for expressing these queries is object-oriented, encouraging re-use of queries, and making it easy to tailor them to a specific framework or project. While code queries have been considered before (both in industry and academia), the object-oriented query language (named .QL) is unique, and the key to creating an agile tool for assessing software quality.

As an advance preview of .QL, let us briefly consider a rule that is specific to the *Polyglot* compiler framework [46]. Every AST node class that has children must implement a method named “visitChildren”. In .QL, that requirement is checked by the query:

```
class ASTNode extends RefType {
  ASTNode() { this.getASupertype+().
    hasQualifiedName("polyglot.ast", "Node") }
  Field getChild() {
    result = this.getAField() and
    result.getType() instanceof ASTNode
  }
}
from ASTNode n
where not(n.declaresMethod("visitChildren"))
select n, n.getChild()
```

Of course this may appear rather complex to the reader for now, but the example still serves to illustrate a couple of important points. First, this is a very useful query: in our own research compiler for *AspectJ* called *abc* [4], we found no less than 18 violations of the rule. Second, the query is concise and in a syntax resembling mainstream languages like SQL and Java. Third, as we shall see later, the class definition for *ASTNode* is reusable in other queries.

.QL has been implemented as part of an Eclipse plugin named *SemmlCode*. *SemmlCode* can be used to query any Java project. It provides an industrial-strength editor for .QL, with syntax highlighting, autocompletion and so on (as shown in Figure 1). Furthermore, the .QL implementation itself is quite efficient. Java projects are stored in relational form in a standard database system. That database system can be a cheap-and-cheerful pure Java implementation such as H2 (which is distributed with *SemmlCode*), or a dedicated system such as PostgreSQL or SQL Server. With a proper database system in place, *SemmlCode* can easily query projects that consist of millions of lines of code. That scalability is another unique characteristic that sets *SemmlCode* apart from other code querying systems.



```

class ASTNode extends RefType {
    ASTNode() {
        this.getASupertype().hasQualifiedName("polyglot.ast", "Node")
    }
    Field getAChild() {
        result = this.getAField() and
        result.getType() instanceof ASTNode
    }
}

from ASTNode n
select n.

```

- contains(Element) predicate - Element
- declaresField(string) predicate - RefType
- declaresMethod(string) predicate - RefType
- fromLibrary() predicate - Element

Fig. 1. The SemmlCode editor

.QL is in fact a general query language, and could be thought of as a replacement for SQL — the application to software quality in these notes is just an example of its power. Like SQL, it has a simple and intuitive syntax that is easy to learn for novices. In SQL, however, that simple syntax does not carry over to complex constructs like aggregates, while in .QL it does. Furthermore, recursion is natural in .QL, and efficiently implemented. Compared to the direct use of recursion in SQL Server and DB2, it can be orders of magnitude faster. Finally, its object-oriented features offer unrivalled flexibility for the creation of libraries of reusable queries.

The structure of these lectures is as follows:

- First we shall consider simple queries, using the existing library of classes that is distributed with SemmleCode. An important concept here is the notion of *non-deterministic methods*, which account for much of the conciseness of .QL queries. We shall also examine features such as casts and instance tests, which are also indispensable for writing effective queries in .QL.
- In the second part of these lectures, we take a close look at the object-oriented features of .QL. First we illustrate the ideas with a number of motivating examples, and then zoom in on a number of subtle issues in the design of .QL's class mechanism. As indicated above, our notion of classes somehow must be tied to a traditional database, and we outline how that is done by appropriate annotation of a database schema.

## 1.1 Exercises

Exercises for the reader have been sprinkled throughout these notes. Most of the exercises involve writing a new query in .QL, and it is strongly recommended that readers follow along with SemmleCode running on a computer. For full instructions on how to install SemmleCode, visit the Semmle website [53].

The Java project used in the exercises is JFreeChart 1.0.6 [33]. We have chosen JFreeChart because it is a well-written piece of Java code, and its developers already make extensive use of *checkstyle* [12], the most popular Eclipse plugin for checking coding rules. Nevertheless, as we shall see, there are still several problems and possible improvements that are easily unearthed with SemmleCode.

There is a special web page accompanying these notes that takes you through the steps required to load JFreeChart in Eclipse, and populate the database with facts about the project [54].

Each exercise has an indicator of its difficulty at the end: one heart is easy (less than five minutes), two hearts is medium (requiring at most ten minutes), and three hearts is a tough exercise (requiring up to fifteen minutes). Full answers can be found in an appendix to these notes.

## 2 Program Queries

### 2.1 A Simple Query

Program queries allow programmers to navigate their source code to identify program fragments with arbitrary semantic properties. As a simple example of a program query in .QL, let us attempt to find classes which violate Java's `compareTo` / `equals` contract. The Java documentation for the `compareTo` method states:

*The natural ordering for a class C is said to be consistent with equals if and only if `(e1.compareTo((Object)e2) == 0)` has the same boolean value as `e1.equals((Object)e2)` for every `e1` and `e2` of class C ... It is strongly recommended (though not required) that natural orderings be consistent with equals.*

The following .QL query identifies those classes that only implement the `compareTo` method without implementing the `equals` method. This is likely to indicate a bug, though it is not necessarily erroneous:

```

from Class c
where c.declaresMethod("compareTo")
      and
      not(c.declaresMethod("equals"))
select c.getPackage(), c

```

This query consists of three parts. First, the **from** statement declares the variables of interest (in this case just the class `c` that we are looking for) together with their types. The second part of the query is the **where** clause imposing some conditions on the results. In this query, the condition is that the class `c` should declare a method called `compareTo` but not a method called `equals`. The final part of the query is the **select** statement, to choose the data to return for each search result, namely the package in which the offending class `c` occurs, together with `c` itself. The order of the **select** items is chosen so that results are presented grouped by the package in which they occur in the source.

The type `Class` is an example of a .QL class. This type defines those programs elements which are Java classes, and defines operations on them. For instance, `declaresMethod` is a test on elements of type `Class`, to select only those Java classes declaring a particular method. We will be describing .QL types and classes in more detail in Section 3, but examples will appear throughout.

*Exercise 1.* Run the above query on JFreeChart. You can do that in a number of ways, but here the nicest way to look at the results is as a table, so use the run button marked with a table (shown below) at the top right-hand side of the Quick Query window. You will get two results, and you can navigate to the relevant locations in the source by double-clicking. Are both of them real bugs? ♡



## 2.2 Methods

Predicates such as `declaresMethod` are useful, but can only filter results. Another common task is to compute some properties of an element. This is achieved by more general .QL methods, which may return results. Let us illustrate this with an example query. Unlike the previous query, which attempted to detect violations of Java's style rules, and therefore could easily be hard-coded into a development environment, the next query is domain-specific.

Suppose that we are working on a compiler, and would like to identify the children of nodes in the AST, for instance to ensure that appropriate methods for visiting children are implemented. To code this as a query, we declare three

variables: *child* for the field, *childType* for type of that field, and *parentType* for the parent class:

```
from Field child, ASTNode childType, ASTNode parentType
where child.getType() = childType
and
    child.getDeclaringType() = parentType
select child
```

The *ASTNode* class is an example of a user-defined class, picking out those types that are AST nodes, and described further in Section 3. The methods *getType* and *getDeclaringType* are defined in the class *Field*, and are used to find the declared type of a field and the type in which the field declaration appears, respectively. The *ASTNode* types appearing in the **from** clause serve to restrict the range of values for the variables they qualify, so that values of the wrong type are simply ignored.

This query is concise, but not terribly satisfactory. In the **from** clause, we define variables *childType* and *parentType* to denote the types of the field and its containing class respectively. However we are not really interested in these types, and indeed they do not appear in the **select** clause. To avoid polluting queries with such irrelevant types, local declarations can be introduced through the **exists** statement:

```
from Field child
where exists(ASTNode childType | child.getType() = childType)
and
    exists(ASTNode parentType | child.getDeclaringType() = parentType)
select child
```

An advantage of the resulting query is that the scopes of the variables representing the types of the field and the container are made explicit. There is a further improvement to be made, however. These fields are only used to restrict the types we are looking for, as we are only interested in AST nodes. We do not need to know the exact types of the child and parent, and so it would be better not to introduce variables to hold these types. .QL offers an **instanceof** construct to achieve this, and we can finally rewrite the query as:

```
from Field child
where child.getType() instanceof ASTNode
and
    child.getDeclaringType() instanceof ASTNode
select child
```

*Exercise 2.* Write a query to find all methods named **main** in packages whose names end with the string **demo**. You may find it handy to use the predicate *string.matches("%demo")* (as is common in query languages, % is a wildcard matching any string). ♥

## 2.3 Sets of Results

Methods in .QL are a convenient way of finding properties of elements, as well as a powerful abstraction mechanism in conjunction with classes. The methods we have seen so far define attributes of elements, such as the declaring type of a field. This is only represent a special case, however, since the data model behind .QL is relational and thus allows methods to define arbitrary *relationships* between elements.

As an example, we will consider a query to find calls to the `System.exit` method. This method terminates the Java virtual machine, without offering the opportunity to clean up any state. This should therefore usually be avoided, and identifying calls to this method allows potentially fragile code to be found. The query is:

```

from Method m, Method sysexit, Class system
where system.hasQualifiedName("java.lang", "System")
    and sysexit.getDeclaringType() = system
    and sysexit.hasName("exit")
    and m.getACall() = sysexit
select m

```

The first line of the **where** clause identifies the `java.lang.System` class, while the second and third lines find the `exit` method in this class. The last line is of more interest. The expression `m.getACall()` finds *all* methods that are directly called by `m`. This method returns a result for each such call, and any logical test on the result is performed for each possible result. In this case, each method called by `m` is compared to the `exit` method. If one of the calls matches (*i.e.*, `m` calls `exit`), then the equality succeeds and `m` is returned. Otherwise, this value of `m` is not returned. The query thus singles out just those methods that (directly) call `exit`.

Methods returning several results can be chained arbitrarily. In the following example, we search for calls between packages, that is all the calls from any method in one package to any method in another package. This may be used to construct a call graph representing dependencies between packages, and identify potential problems such as cycles of dependencies between packages.

```

from Package caller, Package callee
where caller.getARefType().getACallable().calls(
    callee.getARefType().getACallable())
    and caller.fromSource()
    and callee.fromSource()
    and caller != callee
select caller , callee

```

The expression `caller.getARefType()` finds any type within the package `caller`, so that `caller.getARefType().getACallable()` finds any method or constructor (referred to as a *callable*) within some type in the `caller` package. The use of methods returning several values greatly simplifies this expression, and avoids the

need to name unimportant elements such as the type or callable, focusing only on the pairs of packages that we are searching for. As this expression (and its analogue for *callee*) return all callables in the package, the query succeeds exactly for those pairs of packages in which any callable of *caller* calls some callable in *callee*. The predicate *fromSource()*, which holds of program elements defined in a source file (as opposed to a Java class file), serves to exclude results from library code. Finally, the last line of the **where** statement removes trivial dependencies of packages on themselves.

The use of sets of results is sometimes called *nondeterminism*, and a method that possibly has multiple results (like *getARefType* above) is said to be *nondeterministic*. Nondeterminism can sometimes be a bit subtle when used inside a negation. For instance, consider the .QL method *getACallable* that returns any callable (constructor or method) of a class. We could find classes that define a method named “equals” with the query

```
from Class c
where c.getACallable().hasName("equals")
select c
```

In words, for each class *c*, we try each callable, and test whether it is named “equals”; if one of these tests succeeds, *c* is returned as a result. Now consider the dual query, where we wish to identify classes that do *not* have a method named “equals”. We can do that just by negating the above condition, as in

```
from Class c
where not (c.getACallable().hasName("equals"))
select c
```

The negated condition succeeds only when *none* of the tests on the callables of *c* succeeds.

*Exercise 3.* The above queries show how to find types that define a method named “equals”, and how to find types that do not have such a method. Write a query picking out types that define at least *one* method which is not called “equals”. ♡

*Exercise 4.* Continuing Exercise 1 about `compareTo`. You will have found that one class represents a real bug, whereas the other does not. Refine our earlier query to avoid such false positives. ♡

*Exercise 5.* Write a query to find all types in `JFreeChart` that have a field of type `JFreeChart`. Many of these are test cases; can they be excluded somehow? ♡

## 2.4 .QL Type Hierarchies and Casts

In the previous section we defined a query to find all dependencies between packages, by looking for method calls from one package to another. However, such calls are only one possible way in which a package may depend on another package. For instance, a package might use a type from another package (say with



a field of this type), without calling any methods of this type. This is intuitively a dependency which we would like to record, and indeed there are many more ways in which a package may depend on another. This is encapsulated in a method *getADependency*, defined as part of the *metrics library* for Java programs.

The metrics library, which we shall be using throughout these notes, extends the basic .QL class definitions for Java programs with additional methods to compute information about dependencies in source code, and to evaluate various quantitative metrics to analyse the code. In order to separate these definitions from the basic classes, some .QL classes representing program elements, *e.g.* *RefType*, are extended by counterparts in the metrics library, in this case *MetricRefType*, which contains all methods for computing dependencies and metrics on reference types, in addition to the standard methods defined in *RefType*. The class *MetricRefType* does not, however, restrict the set of elements that it contains — any *RefType* is also a *MetricRefType*, and the metric class merely provides an extended view of the same object. Figure 2 describes the inheritance hierarchy for (part of) the standard .QL library for Java programs, with the metrics classes highlighted. The metrics library makes crucial use of multiple

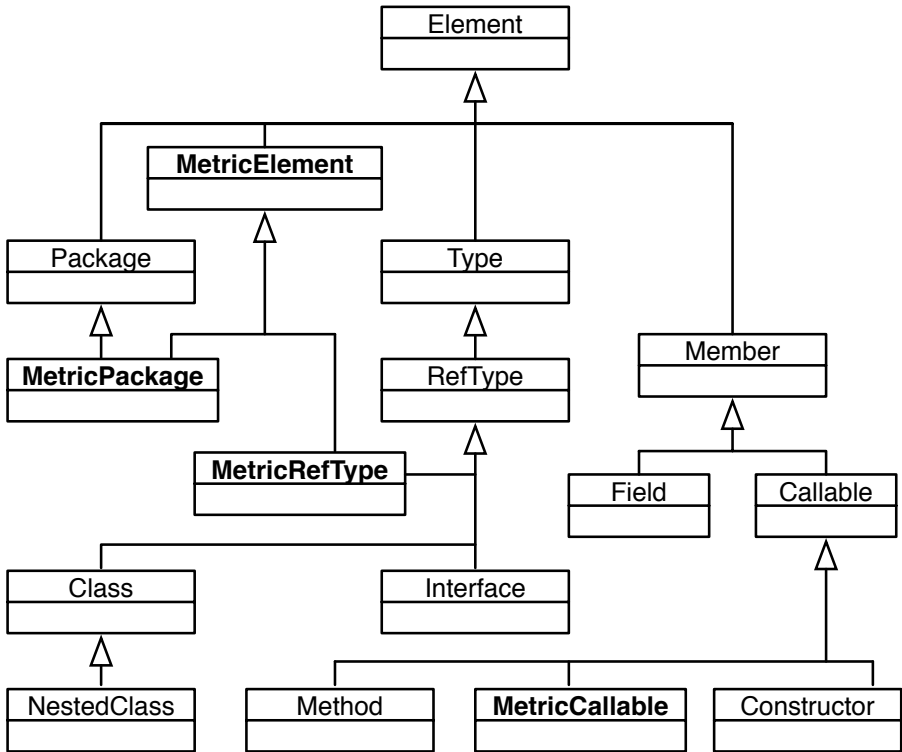
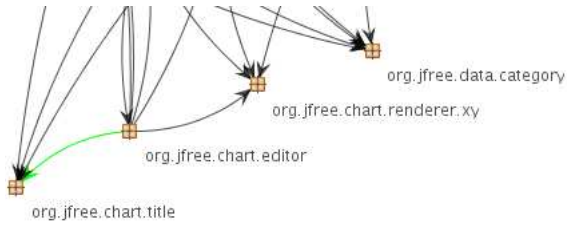


Fig. 2. Standard Library: Inheritance Hierarchy (excerpt)



**Fig. 3.** A fragment of the graph showing inter-package dependencies in JFreeChart

inheritance for `.QL` classes (described later in Section 3) — a *MetricPackage* is both a *Package* and a *MetricElement*.

Using the metrics library it is straightforward to find precise dependencies between packages, as the class *MetricElement* defines the methods *getADependency* to find dependent elements, and *getADependencySrc* to find dependencies from source. The query is shown below:

```
from MetricPackage p
select p, p.getADependencySrc()
```

This query finds all packages *p*, and for each *p* finds those packages defined in source that depend on *p*. The results of this query form a dependency graph, part of which is shown in Figure 3. Suppose, now, that we do not want to inspect just the other packages that *p* depends on, but instead also the types that inhabit such packages. At first you might want to write a query that looks like this:

```
from MetricPackage p
select p, p.getADependencySrc().getARefType() // incorrect!
```

However, that is in fact not type correct, because the result of the method *getADependencySrc* is a *MetricElement*, and *MetricElement* does not have the method *getARefType*. The `.QL` compiler therefore rejects the above query. We must amend it by casting the result of *getADependencySrc* to a *Package*:

```
from MetricPackage p
select p, ((Package) p.getADependencySrc()).getARefType()
```

The cast here will always succeed because when given a package as the receiver, *getADependencySrc* always returns another package. Similarly, starting from a *MetricRefType*, it will always return a *RefType*.

Casts in `.QL` also behave like **instanceof** tests, limiting results to those of a certain type. For instance, this query will filter out all the types that are not an instance of *Class*:

```
from MetricPackage p
select p, (Class) ((Package)p.getADependencySrc()).getARefType()
```

It follows that casts in `.QL` never lead to runtime exceptions as they do in languages like Java: they are merely a test that a logical property (in this case a reference type being a class) is satisfied.

## 2.5 Chaining

The queries that we have seen so far find relatively local properties of program elements, such as the declaring type of a field, or the relationship of one method directly calling another. However, many properties of interest are highly nonlocal, justifying the introduction of *chaining*, also known as transitive closure.

As an example, we shall write a query to find all types that represent AST nodes in a compiler (in this case the Polyglot compiler framework [46]), as suggested previously by our use of the *ASTNode* class. In Polyglot, AST nodes must implement the *Node* interface, and so we are interested in all subtypes of this interface. The standard .QL library for Java provides a convenient *hasSubtype* method to find subtypes of a type, but this only finds *immediate* subtypes, in this case all classes that implement *Node* directly. As we are also interested in classes that are indirect descendents of *Node*, we must use chaining, written in .QL using the + postfix operator:

```

from RefType astNode, RefType rootNode
where rootNode.hasQualifiedName("polyglot.ast", "Node")
        and (rootNode.hasSubtype+(astNode)
            or
            astNode = rootNode)
select astNode

```

The method *hasSubtype+* picks out all direct and indirect subtypes of a type (in this case the *Node* interface). AST nodes are defined as subtypes of *Node*, together with *Node* itself. As this pattern is extremely common, simpler notation is provided for possibly empty chains, as the query is equivalent to:

```

from RefType astNode, RefType rootNode
where rootNode.hasQualifiedName("polyglot.ast", "Node")
        and rootNode.hasSubtype*(astNode)
select astNode

```

The \* operator (known as *reflexive transitive closure* in mathematics) defines possibly empty chains from given relationships, such as the subtype relationship. The symbols +, \* may be familiar from repetition in regular expressions where **a\*** denotes any number of occurrences of **a**, while **a+** denotes at least one occurrence of **a**, justifying the intentional similarity in notation.

*Exercise 6.* There exists a method named *getASuperType* that returns *some* supertype of its receiver, and sometimes this is a convenient alternative to using *hasSubtype*. Uses of methods such as *getASuperType* that return an argument can be chained too. Using *x.getASuperType\**(*y*), write a query for finding all subtypes of `org.jfree.chart.plot.Plot`. Try to use no more than one variable. ♡

*Exercise 7.* When a query returns two program elements plus a string you can view the results as an edge-labelled graph by clicking on the graph button (shown below). To try out that feature, use chaining to write a query to depict the hierarchy above the type `TaskSeriesCollection` in package `org.jfree.data.gantt`.

You may wish to exclude `Object` from the results, as it clutters the picture. Right-clicking on the graph view will give you a number of options for displaying it. ♡



## 2.6 Aggregates

We have so far seen `.QL` used to find elements in a program with certain properties. The language also offers powerful features to aggregate information over a range of values, to compute numerical metrics over query results. These features are substantially more expressive than their SQL counterparts, and allow a wide range of metrics to be computed straightforwardly. As a first example, the following query computes the number of types in each package in a program:

```
from Package p
select p, count(RefType c | c.getPackage() = p)
```

The `count` expression in this query finds those elements  $c$  of type `RefType` (all reference types) satisfying the condition  $c.getPackage() = p$ . The value of the expression is just the number of results, that is the number of reference types in  $p$ .

The above query is a simple example of the aggregate constructs in `.QL`. Aggregates in `.QL` adopt the *Eindhoven Quantifier Notation* [21,34], an elegant notation introduced by Edsger W. Dijkstra and others for the purpose of reasoning about programs. The general syntax for aggregates is

$$\text{aggregateFunction ( localVariables | condition | expression )}$$

The `aggregateFunction` is any function for aggregating sets of values. The functions provided in `.QL` are `count`, `sum`, `max`, `min` and `avg` (average). The `localVariables` define the range of the aggregate expression, namely the variables over the values of which the aggregation is computed. The `condition` restricts the values of interest. In our previous example, the condition was used to restrict counting to those types in the appropriate package. Finally, the `expression` defines the value to be aggregated. In our above example, the expression was omitted. This is always possible when counting, as the value of each result in the aggregation is irrelevant. The `expression` becomes very useful in other aggregates such as summation, however. As an example, the following query computes the average number of methods per type in each package:

```
from Package p
where p.fromSource()
select p, avg(RefType c | c.getPackage() = p | c.getNumberOfMethods())
```

This aggregate finds all reference types in the appropriate package, finds the number of methods for each such type (which itself is easily defined as an aggregate), and averages these numbers of methods.

*Exercise 8.* Display the results of the above query as pie chart, where each slice of the pie represents a package and the size of the slice the average number of methods in that package. To do so, use the *run* button marked with a chart (shown on the next page), and select ‘*as a pie chart*’ from the drop-down menu. ♥



Aggregates may be nested, as the expression whose value is being aggregated is often itself the result of an aggregate. The following example computes the average number of methods per class over an entire project:

```
select avg(Class c
  | c.fromSource()
  | count(Method m | m.getDeclaringType()==c))
```

This query contains two aggregates. The outermost aggregate computes an average over all classes *c* that are defined in source files. For each such class, the value of the innermost aggregate is computed, giving the number of methods in the class, and the resulting values are averaged. This example does not include **from** or **where** clauses, as only one result is returned, so it is not necessary to define output variables.

**Metrics.** An important use of aggregates in program queries is to compute *metrics* over the code. Such metrics may be used to identify problematic areas of the program, such as overly large classes or packages, or classes that do not encapsulate a single abstraction. It is not our aim here to describe the vast library of software metrics that have been proposed (see, for instance, [8,14,18,36,41,56]), but we shall use such metrics as examples of the use of aggregates in .QL.

Many of these metrics are provided as a library, and use the object-oriented features of .QL to achieve encapsulation and reusability, as illustrated in Figure 2. However, as we discuss these features in Section 3, we shall simply express metrics as standalone queries for now.

*Instability.* Instability is a measure of how hard it is to change a package without changing the behaviour of other packages. This is represented as a number between 0 (highly stable) and 1 (highly unstable). Instability is defined as follows:

$$Instability = \frac{EfferentCoupling}{AfferentCoupling + EfferentCoupling}$$

where the *efferent coupling* of a package is the number of types outside the package that the package depends on, while the *afferent coupling* is the number of outside types that depend on this package. Typically a package that has recently been added and is still experimental will have high instability, because it depends on many more established packages, while few other packages depend on the new package. Conversely, a package with many responsibilities that is at the core of an existing project will have low instability, and indeed such packages are hard to modify.

It is easy to define queries to compute efferent and afferent coupling. As these are similar, we present afferent coupling only:

```

from Package p
select p, count(RefType t
    | t.getPackage() != p and
    exists(RefType u |
        u.getPackage() = p and
        depends(t, u)))

```

where the *depends* predicate, part of the metrics library, is fairly straightforward but lengthy, and so is omitted.

We now aim to define the instability metric. This is a clear case for the expressiveness of .QL classes. Without encapsulation mechanisms, there is no easy means of reusing definitions such as afferent coupling. In section 3 we shall see how definitions such as afferent coupling can be defined as methods. These definitions are in fact part of the metrics library and we can write the instability metric in a straightforward way:

```

from MetricPackage p, float efferent, float afferent
where efferent = p.getEfferentCoupling()
and
    afferent = p.getAfferentCoupling()
select p, efferent / (efferent + afferent)

```

Without methods, the aggregate expressions for efferent and afferent coupling would have to be inlined, leading to a far less readable query. The above definition of instability is in fact itself available as a method named *getInstability* on *MetricPackage*, so a shorter version is

```

from MetricPackage p select p, p.getInstability()

```

*Exercise 9.* Not convinced that metrics are any good? Run the above query and display the results as a bar chart—the chart icon mentioned earlier for creating pie charts (shown below) is also used to create bar charts by selecting the appropriate option from the drop-down menu. It will be convenient to display the bars in descending order. To achieve that sorting, add “**as s order by s desc**” at the end of the query. Now carefully inspect the packages with high instability. Sorting the other way round (using **asc** instead of **desc**) allows you to inspect the stable packages. ♥



*Abstractness.* Abstractness measures the proportion of abstract types in a package, as a number between 0 (not at all abstract) and 1 (entirely abstract). Packages should be abstract in proportion to their incoming dependencies, and concrete in proportion to their outgoing dependencies. That way, making changes

is likely to be easy. There is therefore a relationship between abstractness and instability: the more abstract a package is, the lower its instability value should be. A highly abstract, highly stable package is well designed for its purpose and represents a good use of abstraction; conversely, concrete packages may be unstable as nothing depends on concrete packages. Abstract and unstable packages, however, are likely to be useless and represent design flaws.

Abstractness is easy to define: it is just the ratio of abstract classes in a package to all classes in this package. For a package  $p$  this may be written as:

```

from Package p, float abstract, float all
where all = count(Class c | c.getPackage() = p)
      and abstract = count(Class c
                          | c.getPackage() = p and
                          c.hasModifier("abstract"))
      and abstract > 0
      and p.fromSource()
select p, abstract / all
  
```

This query computes the number of types in the variable *all* and the number of abstract types in *abstract*, and for nonempty packages returns the ratio of the two. Again we gave this definition merely for expository reasons, as a method named *abstractness* has already been defined on *MetricPackage*; therefore an alternative query (which also sorts its results in descending order) is:

```

from MetricPackage p where p.fromSource() and p.abstractness() > 0
select p, p.abstractness() as a order by a desc
  
```

As in the previous exercise, this is a suitable query for viewing as a bar chart. The result is shown in Figure 4.

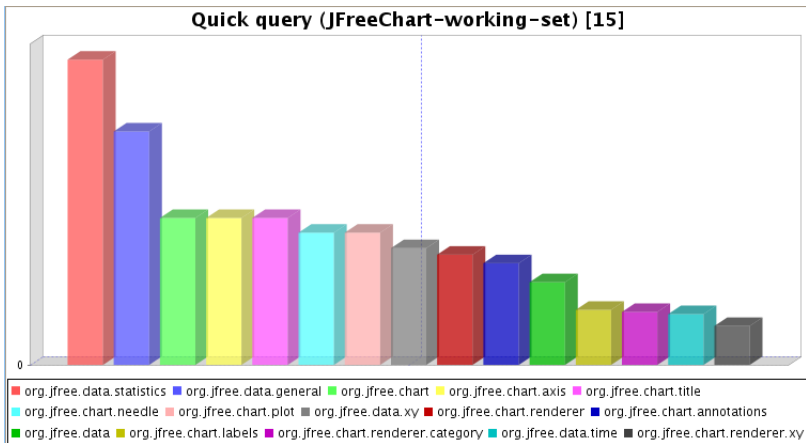


Fig. 4. A bar chart of the *abstractness* of packages in JFreeChart

**Semantics of Aggregation.** Aggregates in .QL are extremely general constructs, and while their use is largely intuitive as our above examples have shown, it is worth describing the exact meaning of aggregate queries in a little more detail. This section may be omitted on first reading, but forms a useful reference for the semantics of aggregate expressions.

An aggregate query of the form

$$\text{aggregate} ( T_1 x_1, T_2 x_2, \dots, T_n x_n \mid \text{condition} \mid \text{expression} )$$

ranges over all tuples  $(x_1, \dots, x_n)$  of appropriately-typed values satisfying **condition**. The **condition** is a conditional formula in which the variables  $x_i$  may appear, and which allows some of the tuples to be excluded. Variables defined outside the aggregate may appear in the condition — the value of such variables is computed outside the aggregate, and the aggregate is evaluated for each possible assignment of values to external variables.

For each tuple  $(x_1, \dots, x_n)$  making the condition true, the **expression** is evaluated. The values of the expression are then collected and aggregated (counted, added, ...). It is important to note that these values are not treated as a set, but allow duplicates. As an example, consider the following expression:

$$\text{sum} ( \text{int } i \mid (i=0 \text{ or } i=1) \mid 2 )$$

Evaluation of this proceeds as described above: the set of integers  $i$  satisfying the condition  $i = 0$  **or**  $i = 1$  is collected, giving just the set  $\{0, 1\}$ . The expression has a constant value of 2, so the values to be summed are *two* copies of 2 — one for the assignment  $i = 0$  and the other for the assignment  $i = 1$ . The result of the aggregate is therefore  $4 = 2 + 2$ .

As another example, consider the following:

$$\begin{aligned} \text{sum} ( \text{int } i, \text{int } j \\ \mid (i=3 \text{ or } i=4) \text{ and } (j=3 \text{ or } j=4) \\ \mid i*i + j*j ) \end{aligned}$$

This sum ranges over four tuples:  $(3, 3)$ ,  $(3, 4)$ ,  $(4, 3)$  and  $(4, 4)$ . The result of the sum is thus  $18 + 25 + 25 + 32 = 100$ .

This notation is convenient, but it would be cumbersome to have to include all parts of the aggregate, including the term and condition, when these are not needed. A number of shorthands are therefore provided:

1. Counting: the expression can always be omitted in a **count** aggregate, as it is irrelevant
2. Numerical values. For other aggregates, such as **sum**, the expression can be omitted in exactly one case, namely if the aggregate defines one local variable of numerical type. For instance, the aggregate

$$\text{sum} ( \text{int } i \mid i=0 \text{ or } i=1 )$$

is simply equivalent to

$$\text{sum} ( \text{int } i \mid i=0 \text{ or } i=1 \mid i )$$



and thus adds the values of  $i$  matching the condition. This obviously cannot be extended to non-numerical variables — it does not make sense to add classes together!

3. Omitting condition: if the condition is not required, it may be omitted altogether. For instance, adding the number of types in each package may be written:

```
sum ( Package p | | p.getNumberOfTypes() )
```

This is particularly simple for counting, as both condition and expression can be omitted. Simply counting the number of packages can be achieved with

```
count ( Package p )
```

*Exercise 10.* The following questions are intended to help reinforce some of the points made above; you could run experiments with SemmlCode to check them, but really they're just for thinking.

1. What is  $\mathbf{sum}(\mathbf{int} \ i \mid i = 0 \ \mathbf{or} \ i = 0 \mid 2)$ ?
2. Under what conditions on  $p$  and  $q$  is this a true equation?

$$\mathbf{sum}(\mathbf{int} \ i \mid p(i) \ \mathbf{or} \ q(i)) = \mathbf{sum}(\mathbf{int} \ i \mid p(i)) + \mathbf{sum}(\mathbf{int} \ i \mid q(i)) \quad \heartsuit$$

### 3 Object-Oriented Queries

So far we have merely written one-off queries, without any form of abstraction to reuse them. To enable reuse, .QL provides classes, including virtual methods and overriding, making it easy to adapt existing queries to new requirements. We present these features in a top-down fashion. First, we discuss some motivating examples, to give the reader a general feel for the way classes are used in practice. Next, we take a step back and examine the semantics of classes and virtual dispatch in some detail through small artificial examples. Finally, we demonstrate how a class hierarchy in .QL can be built on top of a set of simple primitive relations, of the kind found in traditional databases.

#### 3.1 Motivating Examples

**Classes.** A class in .QL is a logical property: when a value satisfies that property, it is a member of the corresponding type. To illustrate, let us define a class for ‘Visible Instance Fields’ in Java, namely fields that are not static and not private. Clearly it is a special kind of normal Java field, so our new class is a subclass of *Field*:

```
class VisibleInstanceField extends Field {
  VisibleInstanceField () {
    not(this.hasModifier("private")) and
    not(this.hasModifier("static"))
  }
}
```

```

predicate readExternally() {
  exists(FieldRead fr |
    fr.getField()=this and
    fr.getSite().getDeclaringType() != this.getDeclaringType())
}
}

```

This class definition states that a *VisibleInstanceField* is a special kind of *Field*. The constructor actually makes the distinguishing property of the new class precise: this field does not have modifier **private** or **static**. The conjunction of the constructor with the defining property of the supertype is called the *characteristic predicate* of a class. It is somewhat misleading to speak of a ‘constructor’ in this context, as nothing is being constructed: it is just a predicate, and naming it the *character* might have been more accurate. However, we adopt the terminology ‘constructor’ because it is familiar to Java programmers.

The above class also defines a predicate, which is a property of some *VisibleInstanceFields*. It checks whether this field is read externally. In order to make that check, it introduces a local variable named *fr* of type *FieldRead*: first we check that *fr* is indeed an access to **this** field, and then we check that the read does not occur in the host type of **this**. In general, a predicate is a relation between its parameters and the special variable **this**.

Newly defined classes can be used directly in select statements. For instance, we might want to find visible instance fields that are *not* read externally. Arguably such fields should have been declared private instead. A query to find such offending fields is:

```

from VisibleInstanceField vif
where vif.fromSource() and
  not(vif.readExternally())
select vif.getDeclaringType().getPackage(),
  vif.getDeclaringType(),
  vif

```

It should now be apparent that all those predicates we have used in previous queries were, in fact, defined in the same way in classes as we defined *readExternally*. We shall shortly see how methods (which can return a result as well as check a property) are defined as class members. It follows that while at first it may appear that .QL is specific to the domain of querying source code, in fact it is a general query language — all the domain-specific notions have been encoded in the query library.

**Classless Predicates.** Sometimes there is no obvious class to put a new predicate, and in fact .QL allows you to define such predicates outside a class. To illustrate, here is a classless predicate for checking that one Java field masks another in a superclass:

```

predicate masks(Field masker, VisibleInstanceField maskee) {
    maskee.getName()==masker.getName() and
    masker.getDeclaringType().hasSupertype+(maskee.getDeclaringType())
}

```

In words, the two fields share the same name, but the *masker* is defined in a subtype of the *maskee*, while the *maskee* is visible. Such field masking is often considered bad practice, and indeed it can lead to confusing programming errors. Indeed, most modern development environments, including Eclipse, provide an option for checking for the existence of masked fields. In .QL, any such coding conventions are easily phrased as queries. In particular, here is a query to find all the visible instance fields that are masked:

```

from Field f, VisibleInstanceField vif
where masks(f,vif)
select f, vif

```

*Exercise 11.* Queries can be useful for identifying refactoring opportunities. For example, suppose we are interested in finding pairs of classes that could benefit by extracting a common interface or by creating a new common superclass.

1. As a first step, we will need to identify *root definitions*: methods that are not overriding some other method in the superclass. Define a new .QL class named *RootDefMethod* for such methods. It only needs to have a constructor, and no methods or predicates.
2. Complete the body of the following classless predicate:

```

predicate similar(RefType t, RefType s, Method m, Method n) { ... }

```

It should check that *m* is a method of *t*, *n* is a method of *s*, and *m* and *n* have the same signature.

3. Now we are ready to write the real query: find all pairs  $(t, s)$  that are in the same package and have more than one root definition in common. All of these are potential candidates for refactoring. If you have written the query correctly, you will find two types in JFreeChart that have 99 root definitions in common!
4. Write a query to list those 99 root definitions in a table. ♡

**Methods.** Often the introduction of a classless predicate is merely a stepping stone towards introducing a new class. Wrapping predicates in a class has several advantages. First, your queries become shorter because you can use method dispatch and so there is no need to name intermediate results. Second, when typing queries you get much better content assist, so you do not need to remember details of all existing predicates.

To illustrate, we introduce a class *MaskedField* as a subclass of the class *VisibleInstanceField* defined earlier:

```

class MaskedField extends VisibleInstanceField {
    MaskedField() { masks(.,this) }
}

```

```

Field getMasker() { masks(result,this) }
string getIconPath() { result = "icons/semmler-logo.png" }
}

```

The constructor for this .QL class consists of the property *masks*(**\_**, **this**) stating that **this** is being masked by some other field. Here, as in many other logic languages, we use the underscore to represent a fresh variable whose value is not relevant. Next the class introduces two methods. The *getMasker*() method returns the masker of **this**. In general, the body of a method is a relation between two special variables named **result** and **this**; the relation may also involve any method parameters. Our new class also defines a method *getIconPath*, which is used to determine the icon that is displayed next to a program element in the results views provided by an implementation. In fact this overrides a method of the same signature in *Field*, and so from now on masked fields will be displayed differently from other fields. Somewhat frivolously, we have decided to give them the Semmler icon.

A query that uses the above class might read:

```
from MaskedField mf select mf,mf.getMasker()
```

and the results will be displayed with the new icon we just introduced.

Note that predicates in a class are really just a special kind of method that returns no result; indeed one could think of them as analogous to void methods in Java. Also note, once again, that methods may be nondeterministic. Indeed, in the above example, it is possible that one field in a Java class *C* is masked by several fields in different subclasses of *C*. Nondeterminism is a natural consequence of the fact that the method body is a relation between **this**, **result** and the method parameters. There is *no* requirement that **result** is uniquely determined.

**Framework-specific Classes.** It is often worthwhile to define new classes that are specific to a particular framework, and we already encountered an example of that earlier, namely *ASTNode* (in Section 2.2). Now we have all the machinery at hand to present the definition of *ASTNode*. We assume the context of the *Polyglot* compiler framework [46], which is intended for experimentation with novel extensions of the Java language. In Polyglot, every kind of *ASTNode* is an implementation of the interface `polyglot.ast.Node`. This can be directly expressed in .QL:

```

class ASTNode extends RefType {
  ASTNode() { this.getASupertype+().
             hasQualifiedName("polyglot.ast","Node") }

  Field getAChild() {
    result = this.getAField() and
    result.getType() instanceof ASTNode
  }
}

```

Note the use of nondeterminism in the constructor: effectively it says that there exists *some* supertype that implements the *Node* interface. The method *getAChild* returns a field of an AST class, that is itself of an AST type. Of course it can happen that no such field exists (if the class represents a terminal in the grammar), or there may be multiple such fields.

In Polyglot, there is a design rule which says that every AST class that has a child must implement its own *visitChildren* method. We now aim to write a query for violations to that rule: we seek AST classes that do *not* declare a method named *visitChildren*, yet a child exists:

```

from ASTNode n
where not(n.declaresMethod("visitChildren"))
select n, n.getAChild()

```

At first it may appear that the condition that a child exists has been omitted, but in fact we do attempt to get a child in the **select** part of the query. If no such child exists then *n.getAChild()* will fail, and so the query will return no results for this value of *n* — exactly what we intended.

This type of coding convention is extremely common in non-trivial frameworks. Normally the conventions are mentioned in the documentation, where they may be ignored or forgotten. Indeed, in our own use of Polyglot in the *abc* compiler, there are no less than 18 violations of the rule. Interestingly, there are no violations in any of the code written by the Polyglot designers themselves — they do as they say. By making the rule explicit as a query, it can be shipped with the library code, thus ensuring that all clients comply with it as well.

As another typical example of a coding convention, consider the use of a factory. Again in Polyglot, all AST nodes must be constructed via such a factory; the only exceptions allowed are super calls in constructors of other AST nodes. Violation of this rule leads to compilers that are difficult to extend with new features.

Definition of a class that captures the essence of an AST node factory in Polyglot can be expressed in .QL as follows:

```

class ASTFactory extends RefType {
  ASTFactory() { this.getASupertype+().
                hasQualifiedName("polyglot.ast", "NodeFactory")
  }
  ConstructorCall getAViolation() {
    result.getType() instanceof ASTNode and
    not(result.getCaller().getDeclaringType()
        instanceof ASTFactory) and
    not(result instanceof SuperConstructorCall)
  }
}

```

The constructor is not interesting; it is just a variation of our earlier example in *ASTNode*. The definition of *getAViolation* is however worth spelling out in detail. We are looking for an AST constructor call which does *not* occur inside

an AST factory, and which is also not a super call from an AST constructor. Again, we successfully used this query to find numerous problems in our own code for the *abc* compiler.

*Exercise 12.* We now explore the use of factories in JFreeChart.

1. Write a query to find types in JFreeChart whose name contains the string “Factory.”
2. Write a class to model the Java type `JFreeChart` and its subtypes.
3. Count the number of constructor calls to such types.
4. Modify the above query to find violations in the use of a factory to construct instances of `JFreeChart`.
5. There are 53 such violations; it is easiest to view them as a table. The interesting ones are those that are not in tests or demos. Inspect these in detail — they reveal a weakness in the above example, namely that we may also wish to make an exception for *this* constructor calls. Modify the code to include that exception. Are all the remaining examples tests or demos? ♥

**Default Constructors.** New .QL classes do not have to define a constructor; when it is not defined, the default constructor is the same as that of the superclass. A .QL class with no constructor of its own does not define a new logical property, but this can often be handy when we want to define a new method that did not exist in the superclass, but which really belongs there.

For instance, suppose that we wish to define a method named *depth* that returns the length of a path from `Object` to a given type in the inheritance hierarchy. That method is not defined in the standard library definition of *RefType*, but it really is a property of any reference type. In .QL, we can add it as such via the definition

```
class RT extends RefType {
  int depth() {
    (this.hasQualifiedName("java.lang", "Object") and result=0)
    or
    (result = ((RT)this.getASupertype()).depth() + 1)
  }
  int maxDepth() {
    result = max(this.depth())
  }
}
```

That is, the depth of *Object* itself is 0. Otherwise, we pick a supertype, compute its depth and add 1 to it. In the recursive step, we cast a *RefType* to a *RT*, just so we can call *depth* on it. That cast will always succeed, because the characteristic predicates of *RT* and *RefType* are identical. Because Java allows multiple inheritance for interfaces, there may be multiple paths from a type to *Object*, and therefore we also define a method for finding the maximum depth of a type. This example was just for illustration and the same result can be obtained via *MetricRefType.getInheritanceDepth()*.

### 3.2 Generic Queries

To conclude our introduction to object-oriented queries, we consider the definition of a metric that exists both on packages and on reference types: the Lakos level [36]. This metric, which was first introduced by John Lakos, is intended to give insight into the *layers* of an application: at the highest level, are the most abstract parts of the program, and at the bottom, utility elements. The Lakos metric is part of the metrics library, and we shall describe how many of the methods from this library that have already been used in earlier sections may be defined.

To appreciate the level metric, consider the well-known drawing framework *JHotDraw*. When arranging packages according to level, the highest point is a package containing sample applications, and a low point is a package of utility classes for recording user preferences. When arranging reference types according to level, most of the high level types are classes containing a `main` method. An example of a low reference type is again a utility class, this time for recording information about a locale.

As illustrated by these examples, Lakos's level metric is useful in sorting the components of a program (be it packages or types) in a top-down fashion, to ease exploration and to gain a bird's-eye view of the structure of a system.

Formally, an element has no level defined if it is cyclically dependent on itself. Otherwise, it has level 0 if it does not depend on any other elements. It has level 1 if it depends on other elements, but those occur in libraries. Finally, if it depends on another element at level  $n$  then it has level  $n+1$ .

Now note that this definition is truly generic: it is the same whether we are talking about dependencies between packages or dependencies between types. Consequently we can define an abstract class, which is a superclass both of reference types and packages. All we need to do to use the metric on particular examples is override the abstract definition of dependency, once in *MetricRefType* and once in *MetricPackage*.

The abstract class (named *MetricElement*) is a subclass of a common super-type of *Package* and *RefType*, namely *Element*. The first method we define is *getADependency*: this returns another element that **this** depends on; and the definition needs to be overridden both in *MetricPackage* and in *MetricRefType*. Next, we define the notion of a *Source Dependency*, simply restricting normal dependency to source elements. We impose that restriction because it does not make sense to trace dependencies through all the libraries: we are interested in the structure of the source itself. It remains to fill in the dots in the class definition below by defining the level metric itself, and we shall do that below.

```
class MetricElement extends Element {
    MetricElement getADependency() {
        result=this // to be overridden
    }
}
```

```

MetricElement getADependencySrc() {
    result = this.getADependency() and result.fromSource()
}
...
}

```

We only define the level of elements in the source. Furthermore, as stated in the above definition, if an element participates in a dependency cycle, then it does not have a level. Here we test that by taking the transitive closure of *getADependencySrc*: in other words, we only consider cycles through source elements. Next come three cases: first, if an element depends on no other elements, it has level 0. Second, if it depends on some other elements but none of those are in source, it has level one. Finally, if it depends on level  $n$ , it has level  $n + 1$ :

```

int getALevel() {
    this.fromSource() and
    not(this.getADependencySrc+=this) and
    ( (not(exists(MetricElement t | t=this.getADependency()))
      and
      result=0)
    or (not(this.getADependency().fromSource()) and
        exists(MetricElement e | this.getADependency() = e) and
        result=1)
    or (result = this.getADependency().getALevel() + 1 )
}

```

Our definition of the Lakos level metric is now almost complete. The above definition of *getALevel* possibly assigns multiple levels to the same element. Therefore, we take the maximum over all those possibilities, and that is the metric we wished to define:

```

int getLevel() {
    result = max(int d | d = this.getALevel())
}

```

*Exercise 13.* The above definition of *getLevel* is in the default library; write queries to display barcharts. Do the high points indeed represent components that you would consider high-level? For types, write a query that calculates how many classes that have maximum level do not define a method named “main”. ♡

### 3.3 Inheritance and Method Dispatch

We have introduced the class mechanism of .QL through a number of motivating examples; it is now time to take a step back and examine more closely what the precise semantics are. In this subsection we shall use minimal examples; they are artificial, but intended to bring out some subtle points in the language design.

**Inheritance.** A class is a predicate of one argument. So for example, we can define a class named *All* that is true just of the numbers 1 through 4:



```

class All {
  All() { this=1 or this=2 or this=3 or this=4}
  string foo() { result="A"}
  string toString() { result = ((int)this).toString() }
}

```

Note that *All* does not have a superclass. Any such class that does not have an ancestor must define *toString*, just to ensure that the results of queries can be displayed. We have also defined a method named *foo*, for illustrating the details of method overriding below. The query

```
from All t select t
```

will return 1, 2, 3 and 4.

Defining a subclass means restricting a predicate by adding new conjuncts. For instance, consider the class definition below:

```

class OneOrTwo extends All {
  OneOrTwo() {this=1 or this=2 or this=5}
  string foo() { result="B"}
}

```

This class consists just of 1 and 2. That is, we take the conjunction of the characteristic predicate of *All* and the constructor. While 5 is mentioned as an alternative in the constructor, it is not satisfied by the superclass *All*. Consequently the query

```
from OneOrTwo t select t
```

returns just 1 and 2. More generally, the predicate corresponding to a class is obtained by taking the conjunction of its constructor, and the predicate corresponding to its superclass.

Because classes are logical properties, they can overlap: multiple properties can be true of the same element simultaneously. For instance, here is another subclass of *All*, which further restricts the set of elements to just 2 and 3.

```

class TwoOrThree extends All {
  TwoOrThree() {this=2 or this=3}
  string foo() { result="C"}
}

```

Note that the element 2 is shared between three classes: *All*, *OneOrTwo* and *TwoOrThree*. The overlap between subclass and superclass is natural, but here *OneOrTwo* and *TwoOrThree* are siblings in the type hierarchy. Overlapping siblings are allowed in .QL, but they can lead to nondeterminism in method dispatch, and we shall discuss that further below.

Summarising our account of classes so far, classes are predicates, and inheritance is conjunction of constructors. It is easy to see what multiple inheritance means in this setting: it is again conjunction. So for example, the following class is satisfied only by the number 2, because that is the only element that its superclasses have in common:

```

class OnlyTwo extends OneOrTwo, TwoOrThree {
  OnlyTwo() { any() }
  string foo() { result = "D" }
}

```

As remarked previously, in cases like this where the constructor is just *true*, its definition may be omitted.

A precise definition of what the predicate corresponding to a class definition can now be stated as: that predicate is the conjunction of all constructors of all its supertypes in the type hierarchy. It is not allowed to define a circular type hierarchy in .QL, so this notion is indeed well-defined. Figure 5 summarises the example so far, showing for each class what elements are satisfied, and what the value returned by *foo* is.

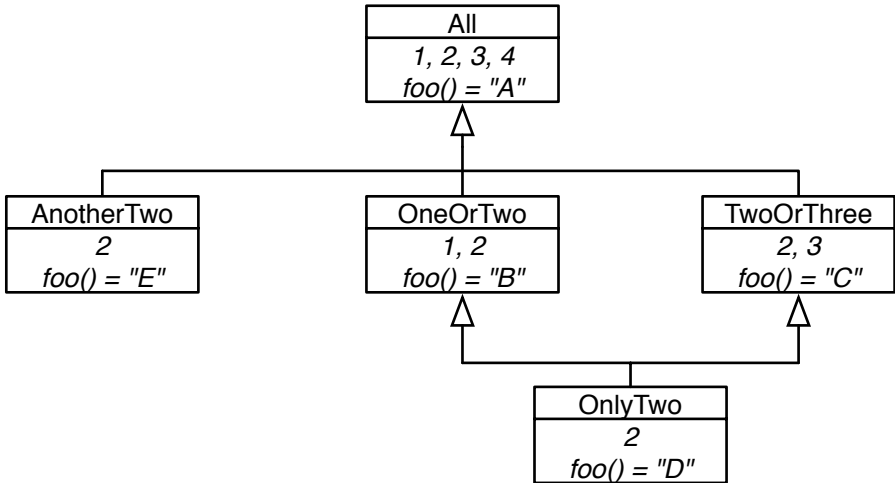


Fig. 5. Example Classes: Inheritance Hierarchy

**Method Dispatch.** Let us now consider the definition of method dispatch. A method definition *m* of class *C* is invoked on a value *x* if *x* satisfies the defining property of *C*, and there is no subclass *D* of *C* which defines a method *m* of the same signature, and *x* also satisfies *D*. In words, we always apply the most specialised definition.

In the above example, the query

```

from All t select t.foo()

```

returns “A”, “B”, “C” and “D”. It returns “A” because 4 satisfies *All*, but none of the other classes. It returns “B” because 1 satisfies *OneOrTwo* but none of the other classes. Next, “C” is returned because 3 satisfies *TwoOrThree*, but not

any of its subclasses. Finally, “D” appears because *OnlyTwo* is the most specific class of 2.

What happens if there are multiple most specific types? This can easily occur, as illustrated by

```
class AnotherTwo extends All {
  AnotherTwo() {this=2}
  string foo() { result="E" }
}
```

Now the number 2 has two most specific types, namely *OnlyTwo* and *AnotherTwo*. In such cases *all* most specific implementations are tried. In particular the query

```
from OneOrTwo t select t.foo()
```

returns “B”, “D”, and “E”. It is quite rare for such nondeterminism to be intended, and it is therefore important to take care when designing a class hierarchy that few unintended overlaps between siblings occur. Of course it is always possible to resolve the nondeterminism by introducing another subclass that simultaneously extends all the overlapping subclasses.

Programmers who are familiar with object-oriented programming in Java may find it at first disconcerting that dispatch is entirely based on logical properties. The inheritance hierarchy is used only to build up those logical properties via conjunction and more primitive predicates. The semantics of runtime dispatch is however entirely in terms of the semantics of classes as predicates. Upon reflection, that is analogous to the way method dispatch works in Java, based on the runtime type of objects, and not at all influenced by static typing. The design of .QL is thus consistent with traditional notions of object-orientation, in that static type-checking and runtime semantics are not intertwined.

There is one small exception to the principle that dispatch is entirely a runtime phenomenon, to avoid unwanted confusion between method signatures. In deciding what method definitions to consider as candidates for dispatch, at compile time the compiler inspects the static type of the receiver (*i.e.*  $x$  in a call  $x.bar(..)$ ) and finds the root definitions of the corresponding method: those are definitions (of  $bar$ ) in supertypes of the receiver type that do not override a definition in another superclass themselves. All definitions of  $bar$  in subtypes of the root definitions are possible candidates. As said, this is just a device to avoid accidental confusion of method names, and it is not a key element of the semantics of .QL.

In summary, method dispatch occurs in two stages, one static and one dynamic. To resolve a call  $x.bar(..)$ , at compile-time we determine the static type of  $x$ , say  $T$ . We then determine all root definitions of  $bar$  above  $T$  (methods with the same signature that do not themselves override another definition). This is the set of candidates considered for dispatch at runtime. At runtime itself, each of the candidates applies only if the value of  $x$  satisfies the corresponding predicate, and there is no more specific type that  $x$  also satisfies.

*Exercise 14.* Suppose the class *OnlyTwo* does not override *foo*. Does that make sense? What does the *.QL* implementation do in such cases? ♡

*Exercise 15.* Construct an example to demonstrate how dispatch depends on the static type of the receiver. ♡

### 3.4 Database Schema

We have claimed earlier that *.QL* is a general query language, which is specialised to a particular application by constructing a class hierarchy. Indeed, it is our claim that *.QL* can be used on *any* relational database. A key ingredient of that argument is still missing, however, and that is how the class mechanism interacts with information stored in such a relational database, and that is explained now.

**Column types.** The primitive relations store information about the type hierarchy, class definitions, method definitions and so on. The schema for these relations is just like that found in a normal relational database, giving field names and types. The twist needed to create class definitions is that every field in fact has two types: one for the use of the underlying database (a *representation type*), and one for *.QL* (a *column type*).

For example, here is the schema for the table that represents method declarations.

```
methods(int id: @method,
        varchar(100) nodeName: string ref,
        varchar(900) signature: string ref,
        int typeid: @type ref,
        int parentid: @reftype ref,
        int cuid: @cu ref,
        int location: @location ref);
```

In words, we store a unique identifier for each method, a name, a signature, the return type, the declaring type, the compilation unit it lives in, and its location. The first type for each field (set in teletype font) is its representation type. For example, the unique method identifier happens to be an integer. Representation types describe the values stored in the database, but are not exposed to *.QL* programs, since it is undesirable to leak such low-level implementation details. As a result, each field has another type (the column type) for use in *.QL*, shown in italics above. Conventionally, column types start with the character '@', except for primitive types such as *string* or *int*.

The declaration of the *methods.id* field doubles as the declaration of the type *@method*: we define that type to be any value occurring in this column of the methods table. Such a type defined simultaneously with a field is called a *column type*. All the other fields have types that are references to column types that already exist elsewhere. For instance, the *cuid* field (short for Compilation Unit Identifier) is a reference to the *@cu* type; and that type is defined in the table that represents compilation units.

Not all column types are introduced via a field declaration, however. Some of these are defined as the union of other types. For example:

```
@reftype = @interface | @class | @array | @typevariable;
```

This defines the notion of a reference type: it is an interface, or a class, or an array, or a type variable.

### 3.5 From Primitives to Classes

Now suppose we wish to write a new class for querying Java methods. As we have seen, there is a primitive relation *methods* one can build on. Furthermore, classes can extend column types, and this is the key that makes the connection between the two worlds. The characteristic predicate of a column type is just that a value occurs in its defining column. We can therefore define

```
class MyMethod extends @method {
  string getName() {methods(this,result,--,--,--,-)}
  string toString() {result=this.getName()}
}
```

Note how we can refer to primitive relations in the same way as we refer to classless predicates.

It should now be apparent that the design of the .QL language is independent of its application to querying Java code, of even querying source code more generally. There is a collection of primitive relations that comes with the application, and those primitive relations have been annotated with column types. In turn, those column types then form the basis of a class library that is specific to the application in hand. In principle, any existing relational database can be queried via .QL.

Of course annotating the database schema and constructing a good class library is not a trivial exercise. In the case of querying Java, the current distribution of .QL has a schema that consists of about forty primitive relations, and approximately fifty column types (there are more column types than relations because some column types are unions of others). The corresponding library of classes contains 70 class definitions, and amounts to 941 lines of .QL code (excluding white space and comments).

*Exercise 16.* Extend the above class definition with *getDeclaringType*. ♡

## 4 Implementation

In earlier sections we have seen the .QL query language, providing a convenient and expressive formalism in which to write queries over complex data. We then discussed the object-oriented features of .QL, which allow complex queries to be packaged up and reused in a highly flexible fashion. These features are essential to build up a library of queries over programs, but this begs the question of how .QL may be implemented, and it is the aim of this last section to describe

the implementation strategy. We first describe the intermediate language used for .QL queries, a deductive query language known as Datalog. We then sketch the translation of .QL programs into Datalog, before briefly outlining the implementation of Datalog queries over relational databases.

## 4.1 Datalog

.QL is based on a simple form of logic programming known as Datalog, originally designed as an expressive language for database queries [26]. All .QL programs can be translated into Datalog, and the language draws on the clear semantics and efficient implementation strategies for Datalog. In this section we describe the Datalog language before outlining how .QL programs may be translated into Datalog. Datalog is essentially a subset of .QL, and as such we shall be using .QL syntax for Datalog programs.

**Predicates.** A Datalog program is a set of *predicates* defining logical relations. These predicates may be recursive, which in particular allows the transitive closure operations to be implemented. A Datalog predicate definition is of the form:

**predicate**  $p(T_1\ x_1, \dots, T_n\ x_n) \{ \text{formula} \}$

This defines a named predicate  $p$  with variables  $x_1, \dots, x_n$ . In a departure from classical Datalog each variable is given a type. These restrict the range of the relation, which only contains tuples  $(x_1, \dots, x_n)$  where each  $x_i$  has the type  $T_i$ .

The body of a Datalog predicate is a logical formula over the variables defined in the head of the clause. These formulas can be built up as follows:

$$\begin{aligned} \text{formula} ::= & \text{predicate}(\text{variable}, \dots, \text{variable}) \\ & | \text{test}(\text{variable}, \dots, \text{variable}) \\ & | \text{variable} = \text{expr} \\ & | \mathbf{not}(\text{formula}) \\ & | \text{formula} \mathbf{or} \text{formula} \\ & | \text{formula} \mathbf{and} \text{formula} \\ & | \mathbf{exists}(\text{Type variable} \mid \text{formula}) \end{aligned}$$

That is, a formula is built up from uses of predicates through the standard logical operations of negation, disjunction, conjunction and existential quantification. In addition to predicates, *tests* are allowed in Datalog programs. A test is distinct from a predicate in that it can only be used to test whether results are valid, not generate results. An example of a test is a regular expression match. The test  $X$  **matches** "C%" is intended to match all strings beginning with "C". Evidently such a test cannot be used to generate strings, as there are infinitely many possible results, but may constrain possible values for  $X$ . In contrast, a predicate such as  $depends(A, B)$  may generate values — in this case, the variables  $A$  and  $B$  are bound to each pair of elements for which  $A$  depends on  $B$ . In a

manner of speaking, variable occurrences in a test are non-binding: such variables must also occur in a predicate.

Arguments to predicates are simply variables in Datalog, but *expressions* allow the computation of arbitrary values. Expressions are introduced through formula such as  $X = Y + 1$  defining the value of a variable, and include all arithmetic and string operators. In addition, expressions allow aggregates to be introduced.

$$\begin{aligned} \text{expr} ::= & \text{variable} \\ & | \text{constant} \\ & | \text{expr} + \text{expr} \\ & | \text{expr} \times \text{expr} \\ & | \dots \\ & | \text{aggregate} \end{aligned}$$

Our definition of Datalog differs from usual presentations of the language in several respects. The first difference is largely inessential. While we allow arbitrary use of logical operators in formulas, most presentations requires Datalog predicates to be in *disjunctive normal form*, where disjunction can only appear at the top level of a predicate and the only negated formulas are individual predicates. However, any formula may be converted to disjunctive normal form, so this does not represent a major departure from pure Datalog. Expressions, on the other hand, are crucial in increasing the expressiveness of the language. In pure Datalog expressions are not allowed, and this extension to pure Datalog is nontrivial, with an impact on the semantics of the language.

**Datalog Programs.** A Datalog program contains three parts:

1. A *query*. This is just a Datalog predicate defining the relation that we wish to compute.
2. A set of user-defined, or *intensional* predicates. These predicates represent user-defined relations to be computed to evaluate the query.
3. A set of *extensional* predicates. These represent the elements stored in the database to be queried.

The general structure of a Datalog program therefore mirrors that of a .QL program. The query predicate corresponds to the query in a .QL program, while classes and methods may be translated to intensional predicates. Finally, in the context of program queries the extensional predicates define the information that it stored about the program. Examples may include the inheritance hierarchy, for instance represented as a table *hasSubtype* of each type and its direct subtypes; or the set of classes in the program.

**Semantics and Recursion.** The semantics of Datalog program are very straightforward, in particular in comparison to other forms of logic programming such as Prolog. A key property is that termination of Datalog queries is not an issue. The simplicity of the semantics of Datalog programs (and by implication of .QL programs) is an important factor in its choice as an intermediate

query language, as it is straightforward to generate Datalog code. It is worth exploring the semantics in a little more detail, however, as a few issues crop up when assigning meaning to arbitrary Datalog programs.

For our purposes, the meaning of a Datalog program is that each predicate defines a relation, or set of tuples, between its arguments. Other, more general, interpretations of Datalog programs are possible [58], but this will suffice for our purposes. An important feature is that these relations should be finite, so that they may be represented explicitly in a database or in memory. It is customary to enforce this through *range restriction*, that is to say ensuring that each variable that is an argument to a predicate should be restricted to a finite set. In our case, this is largely straightforward, as each variable is typed. Types such as `@class` or `@reftype` restrict variables to certain kinds of information already in the database, in this case the sets of classes or reference types in the program. As there can only be finitely many of these, any variable with such a type is automatically restricted. However, primitive types such as `int` are more troublesome. Indeed it is easy to write a predicate involving such variables that defines an infinite relation:

```
predicate p(int X, int Y) { X = Y }
```

This predicate contains all pairs  $(X, X)$ , where  $X$  is an integer, which is infinite and therefore disallowed. As a result, the type system of .QL ensures that any variable of primitive type is always constrained by a predicate, restricting its range to a finite set.

In the absence of recursion, the semantics of a Datalog program is very straightforward. The program can be evaluated bottom-up, starting with the extensional predicates, and working up to the query. Each relation, necessarily finite by range-restriction, can be computed from the relations it depends on by simple logical operations, and so the results of the query can be found.

The situation is more interesting in the presence of recursion. Unlike other logic programs in which evaluation of a recursive predicate may fail to terminate, in Datalog the meaning of a recursive predicate is simply given by the least fixed point of the recursive equation it defines. As an example, consider the recursive predicate

```
predicate p(int X, int Y) { q(X, Y) or (p(X,Z) and q(Z,Y)) }
```

where  $q$  denotes (say) the relation  $\{(1, 2), (2, 3), (3, 4)\}$ . Then  $p$  denotes the solution of the relation equation  $P = q \cup P; q$ , in which  $;$  stands for relational composition. This is just the transitive closure of  $q$ , so the relation  $p$  is simply

$$p = \{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)\}$$

This least fixed point interpretation of Datalog programs makes it easy to find the value of any predicate. For instance, consider

```
predicate p(int X) { p(X) }
```

This predicate would be nonterminating as a Prolog program. However, in Datalog this is just the least solution of the equation  $P = P$ . As every relation satisfies this equation, the result is just the empty relation!



More precisely, the meaning of a Datalog program can be defined as follows. First, break the program up into components, where each component represents a recursive cycle between predicates (formally, a strongly-connected component in the call graph). Evaluation proceeds bottom-up, starting with extensional predicates and computing each layer as a least fixed point as above.

There are two technical restrictions to the use of recursion in Datalog. The first is known as *stratification*, and is necessary to deal with negation properly. The problem can be illustrated by this simple example:

```
predicate p(@class X) { not(p(X)) }
```

What should this predicate mean? It is defined as its own complement, so a class lies in  $p$  iff it does not lie in  $p$ . There is no relation satisfying this property, so we cannot assign a simple relational interpretation to this program. To avoid this issue, we only consider stratified Datalog. In this fragment of Datalog, negation cannot be used inside a recursive cycle. That is, a cycle through mutually recursive predicates cannot include negation. This is not a problem in practice, and stratification is not a substantial obstacle to expressiveness.

A similar problem is posed by our use of expressions, which does not lie in the scope of classical Datalog. While expressions increase the power of the language, their interaction with recursion is problematic. For instance, consider the following:

```
predicate p(int Y) { Y = 0 or (Y = Z+1 and p(Z)) }
```

Clearly 0 lies in  $p$ . Therefore 1 must also lie in  $p$  from the recursive clause, and in this manner every number  $n$  lies in  $p$ . The use of expressions in recursive calls may therefore lead to infinite relations, and thus nontermination. In .QL this may also lead to nonterminating queries, and so care must be used when using arithmetic expressions in recursive calls — if, as in the above example, the expression can create new values for each recursive call, then the query may be nonterminating.

## 4.2 Translating .QL

The precise semantics of .QL programs are defined by their translation into Datalog programs. The outline of this translation is quite straightforward, as the overall structure of .QL programs closely mirrors that of Datalog programs. In particular, the query in a .QL program is translated into a Datalog query, while methods and classless predicates are translated to Datalog intensional predicates.

**Translating Queries.** The general form of a .QL() query (ignoring **order by** clauses, which merely amount to a post-processing step) is:

```
from T1 x1, T2 x2, ..., Tn xn
where formula
select e1, e2, ..., ek
```

where each  $e_i$  is an expression, and each  $x_i$  is a declared variable of type  $T_i$ .

It is straightforward to translate this to a Datalog query, which is just a standard predicate. The resulting relation has  $k$  parameters (one for each selected expression), and so the query predicate has  $k$  parameters. The variables  $x_1$  through  $x_n$  can be introduced as local variables, defined by an existential quantifier. As a result, the Datalog translation of the above query, omitting types, is:

```
predicate p(res1, res2, ..., resk) {
  exists (T1 x1, T2 x2, ..., Tn xn |
    formula2
    and res1 = e1
    and res2 = e2
    and ...
    and resk = ek
  )
}
```

where *formula2* is obtained from *formula* by translating away all non-Datalog features of .QL, and in particular method calls, as described below

**Translating Classes.** Classes are translated into individual Datalog predicates, representing constructors, methods and class predicates. In most cases the translation is straightforward, the key aspect being the translation of method calls.

A .QL method is merely a particular kind of Datalog predicate involving two special variables — **this** and **result**. The **this** variable holds the value that is a member of the class, while the **result** variable holds the result of the method. As an example, consider the following method to compute a string representation of the fully qualified name of a type:

```
class RefType {
  ...

  string getQualifiedName() {
    result = this.getPackage() + "." + this.getName()
  }

  ...
}
```

This is translated into the following Datalog predicate

```
predicate RefType_getQualifiedName(RefType this, string result) {
  exists(string package, string type |
    RefType_getName(this, type)
    and RefType_getPackage(this, package)
    and result = type + "." + package
  )
}
```

This extends to methods taking an arbitrary number of parameters, in which case the two parameters **this** and **result** are simply added to the list of parameters. Apart from the translation of method calls, which we will describe shortly, there are few differences between the body of the method and the body of the generated predicate. Class predicates are similar, but as predicates do not return a value, the **result** variable is not used. For instance, the method

```
class RefType {
  ...

  predicate declaresField(string name) {
    this.getAField().getName() = name
  }

  ...
}
```

is translated to the following Datalog predicate:

```
predicate RefType_declaresField(RefType this, string name) {
  exists(Field field |
    RefType_getAField(this, field)
    and Field_getName(field, name)
  )
}
```

Both examples highlight one of the crucial advantages of .QL methods over Datalog predicates, in addition to extensibility. In Datalog, it is necessary to name each intermediate result, as is the case with the field in the above example. In contrast, methods returning (many) values allow queries to be written in a much more concise and readable manner.

Finally, constructors are simply translated to Datalog predicates denoting the character of each class. For instance, consider the definition of anonymous Java classes:

```
class AnonymousClass extends NestedClass {
  AnonymousClass() { this.isAnonymous() }
}
```

The constructor for this class is translated into a predicate defining precisely those elements that are nested classes. These are the Java elements that are nested classes, additionally satisfying the *isAnonymous* predicate:

```
predicate AnonymousClass(NestedClass this) {
  NestedClass_isAnonymous(this)
}
```

In the above, the type of **this** enforces the fact that an anonymous class must be nested. When a class inherits from multiple classes, the translation is a little more complicated. Consider the class *Interface*, with no constructor:

```
class Interface extends RefType, @interface {
  ...
}
```

This class extends both *RefType* and the column type *@interface*, and thus an element is an *Interface* exactly when it is both a *RefType* and an *@interface*. This is encoded in the generated constructor for *Interface*:

```
Interface(RefType this) { @interface(this) }
```

Despite the fact that *Interface* does not define a constructor, it restricts the range of values that it encompasses by inheritance, and thus this characteristic predicate must be generated.

**Translating Method Calls.** In the above, we have described the translation of methods into Datalog predicates with extra arguments **this** and **result**, and informally shown some method calls translated into calls to the generated predicates. In our examples, the translation was straightforward, as the type of the receiver was known, and so it was immediately apparent which predicate should be called. However, as .QL uses virtual dispatch, the method that is actually used depends on the value it is invoked on, and this translation scheme cannot work in general.

To illustrate the translation of method dispatch in .QL, let us recall the class hierarchy defined in Section 3, simplified for this example:

```
class All {
  All() { this=1 or this=2 or this=3 or this=4 }
  string foo() { result = "A" }
}
```

```
class OneOrTwo extends All {
  OneOrTwo() { this=1 or this=2 }
  string foo() { result = "B" }
}
```

```
class TwoOrThree extends All {
  TwoOrThree() { this=2 or this=3 }
  string foo() { result="C" }
}
```

As we have seen previously, each of the implementations of *foo* is translated into a Datalog predicate:

```
predicate All_foo(All this, string result) { result = "A" }
predicate OneOrTwo_foo(OneOrTwo this, string result) { result = "B" }
predicate TwoOrThree_foo(TwoOrThree this, string result) { result = "C" }
```

However, when a call to the *foo* method is encountered, the appropriate methods must be chosen, depending on the value of the receiver of the call. .QL method

dispatch selects the most specific methods, of which there may be several due to overlapping classes, and returns results from all most specific methods. Only the most specific methods are considered, so that a method is not included if it is overridden by a matching method.

This virtual dispatch mechanism is implemented by defining a *dispatch predicate* for each method, testing the receiver against the relevant types and choosing appropriate methods. Testing the type of the receiver is achieved by invoking the characteristic predicate for each possible class, leading to the following dispatch method for *foo*:

```
predicate Dispatch_foo(All this, string result) {
    OneOrTwo_foo(this, result)
  or TwoOrThree_foo(this, result)
  or (not(OneOrTwo(this)) and not(TwoOrThree(this))
      and All_foo(this, result))
}
```

Let us examine this dispatch predicate a little more closely. The parameter **this** is given type *All*, as this is the most general possible type in this case. The body of the predicate consists of three possibly overlapping cases. In the first case, the *foo* method from *OneOrTwo* is called. Note that this only applies when **this** has type *OneOrTwo*, due to the type of the **this** parameter in *OneOrTwo*. As *OneOrTwo* does not have any subclasses, its *foo* method cannot be overridden and whenever it is applicable it is necessarily the most specific. The second case is symmetrical, considering the class *TwoOrThree*. These cases are overlapping, if **this** = 2, and so the method can return several results. Finally, the third case is the “default” case. If **this** did not match either of the specific classes *OneOrTwo* or *TwoOrThree*, the default implementation in *All* is chosen.

Suppose now that we extend the example to the full class hierarchy shown in Figure 5, as follows:

```
class OnlyTwo extends OneOrTwo, TwoOrThree {
  foo() { result = "D" }
}
class AnotherTwo extends All {
  AnotherTwo() { this = 2 }
  foo() { result = "E" }
}
```

In this new hierarchy, we added two classes with exactly the same characteristic predicate. This changes method dispatch whenever **this** = 2, as the newly introduced methods are more specific than previous methods for this case. To extend the previous example with these new classes, we simply lift out the new implementations of *foo*:

```
predicate OnlyTwo_foo(OnlyTwo this, string result) { result = "D" }
predicate AnotherTwo_foo(AnotherTwo this, string result) { result = "E" }
```

and change the dispatch predicate accordingly:

```
predicate Dispatch_foo(All this, string result) {
  OnlyTwo_foo(this, result)
  or AnotherTwo_foo(this, result)
  or (not(OnlyTwo(this))
      and OneOrTwo_foo(this, result))
  or (not(OnlyTwo(this))
      and TwoOrThree_foo(this, result))
  or (not(OneOrTwo(this))
      and not(TwoOrThree(this))
      and not (AnotherTwo(this))
      and All_foo(this, result))
}
```

The only changes, apart from the introduction of cases for the two new classes, is that the existing cases for *OneOrTwo*, *TwoOrThree* and *All* must be amended to check whether the method is indeed the most specific one.

### 4.3 Implementing Datalog Queries

**Database Implementation.** The use of Datalog as an intermediate language for .QL has two benefits. The first is the simplicity of Datalog, making it straightforward to define the semantics of .QL by translation to Datalog. In addition, Datalog was designed as a query language over relational databases, and can be implemented efficiently over familiar relational query languages, in particular SQL.

A .QL program ranges over a database schema defining the relations that queries can inspect. In the translated Datalog program these just form the extensional predicates, while intensional predicates define new relations that are computed by querying this data. Such Datalog queries can be translated directly into SQL statements, and the aim of this section is to introduce this translation.

For each defined predicate, say

```
predicate p(A x, B y) {
  exists (C z | q(x, z) and r(z, y))
}
```

a new table (also called **p**) is created. The table **p** has columns **x** and **y**, corresponding to the query fields. The types of these columns can be deduced from the .QL column types, but are not identical: .QL allows for rich user-defined column types such as *@class*, while databases typically only provided simple scalar types such as integers or characters. Primitive types can be represented directly in the database, naturally, but for user-defined types some representation (typically based on unique identifiers) must be chosen.

This table is then populated with the result of the query, as computed by an SQL **SELECT** statement. The first step of this translation is to make the variable types explicit. Recall that variable types restrict the range of values that

a variable can take, which must be represented in the SQL query. We therefore make these types explicit in the Datalog query, resulting in the following (un-typed) query:

```
predicate p(x, y) {
    A(x) and B(y)
    and exists(z | C(z) and q(x,z) and r(z,y))
}
```

This relation is essentially a join of the **q** and **r** relations, together with the type restrictions on variables. This may be computed by the following SQL statement, assuming that tables **q(a,b)** and **r(c,d)** have already been computed, as have all type tables **A(x)**, **B(x)** and **C(x)**:

```
SELECT DISTINCT q.a, r.d
FROM q
    INNER JOIN r
        ON r.c = q.b
    INNER JOIN C
        ON C.x = q.b
    INNER JOIN A
        ON q.a = A.x
    INNER JOIN B
        ON r.d = B.x
```

The first line of this query selects the **x** and **y** variables from tables **q** and **r**. The **DISTINCT** modifier is used to guarantee that the result is a set and does not contain duplicates, as SQL queries otherwise produce bags of results. The relation constructed in the **FROM** clause is simply the join of all predicates conjoined together in the predicate *p*, joining on any variables that appear in several predicates.

This implementation strategy allows arbitrary Datalog predicates to be implemented as SQL queries. A conjunction may, as we have seen above, simply be translated as an SQL join. More general formulas can be implemented by converting the body of each predicate to disjunctive normal form, in which the formula is expressed as a disjunction of conjunctions. As an example, consider the following predicate (in disjunctive normal form), ignoring types for concision:

```
predicate p(x, y) {
    exists (z | q(x,z) and r(z,y))
    or ( q(x, y) and not(t(y)) )
}
```

This may be translated into the following SQL query, in which the disjunction is simply turned into a union, where in addition to previous tables **t(e)** has been computed:

```
SELECT DISTINCT q.a, r.d
FROM q
```

```

INNER JOIN r
  ON r.c = q.b
UNION

SELECT DISTINCT q.a, q.b
FROM q
WHERE NOT EXISTS
  (SELECT t.e
   WHERE t.e = q.b)

```

These examples illustrate the principles behind the translation of Datalog queries, and thus .QL programs, to SQL. The only Datalog feature that we have not considered are the use of expressions and aggregates, which are beyond the scope of these notes (note, however, that both are present in SQL, and so do not give rise insurmountable obstacles). This translation is crucial for the efficient implementation of .QL on very large data sets, thanks to the efficiency of database query optimisers. However, it is clear that .QL is far better suited to writing queries over complex data sets, such as the representations of programs, than SQL.

**Recursion.** The translation from Datalog to SQL requires the program to be evaluated bottom-up, so that a relation is computed only when all the relations it depends on have themselves been evaluated. However, this is only possible for nonrecursive programs. Any recursive predicate will depend on itself, and thus the evaluation strategy is a little more involved. To conclude our description of the implementation of .QL we therefore outline the translation of recursive predicates. For simplicity, we exclude mutual recursion and consider only a single recursive predicate.

The most straightforward translation of recursive queries is to use recursive SQL queries as a direct translation. The SQL:1999 standard specifies *common table expressions*, with which queries that refer to their own result set may be written. However, support for common table expressions among widespread database management systems is patchy, and available implementations suffer from performance problems. As recursive queries are common when analysing programs, this application of .QL requires good performance in the implementation of recursion. As a result, we use our own implementation, based on well-known algorithms for evaluating recursive equations.

A recursive query, say (omitting types):

**predicate**  $p(x, y)$  {  $q(x, y)$  **or exists** ( $z \mid q(x, z)$  **and**  $p(z, y)$ ) }

gives rise to a recursive equation of the form  $p = F(p)$ , where  $F$  is a function from relations to relations. In the above case the function is simply:

$$F(R) = q \cup q; R$$

That is, this function simply computes the value of the body of the predicate, replacing the recursive occurrence of  $p$  with the parameter  $R$ . The semantics



of Datalog then prescribe that the value of  $p$  should be the *least* solution of the equation  $p = F(p)$ . To compute this, we may appeal to the *Knaster-Tarski fixpoint theorem*, which asserts that such a least solution exists, as long as  $F$  is monotonic (guaranteed in the absence of negated recursive calls), and that the solution can be obtained by iterating the  $F$  function, starting with the empty relation:

$$p = \lim_{n \rightarrow \infty} F^n(\emptyset)$$

This suggests an algorithm for computing the fixpoint:

```

1  old =  $\emptyset$ 
2  p = F(old)
3  while (p  $\neq$  old)
4    old = p
5    p = F(old)

```

The assignment  $p = F(\text{old})$  can be computed as a nonrecursive SQL query, this clearly provides an implementation strategy. However, it is not optimal. The successive iterations of this algorithm give the following values for  $p$ :

$$\begin{aligned}
p &= \emptyset \\
p &= F(\emptyset) = q \cup q; \emptyset = q \\
p &= F(q) = q \cup q; q = q \cup q^2 \\
p &= F(q \cup q^2) = q \cup q; (q \cup q^2) = q \cup q^2 \cup q^3 \\
&\dots
\end{aligned}$$

In general, after  $n$  iterations the value of  $p$  is  $q \cup q^2 \cup \dots \cup q^n$ . The difference between the results for iterations  $n$  and  $n + 1$  is therefore just  $q^{n+1}$ . However, the relations  $q$  to  $q^n$  are recomputed anyway, making this algorithm expensive.

The inefficiency of the naive algorithm for evaluating recursion leads to the so-called “semi-naive” algorithm presented below [6]. The idea is to observe that at each step, we need only apply the function  $F$  to values that were newly created at the previous step. In our example, the new tuples at step  $n$  are those of  $q^n$ . In step  $n + 1$  we thus only need to add the relation  $F(q^n)$ , and keep all other tuples in the accumulated relation.

The semi-naive evaluation strategy is almost always applicable, but does impose a restriction on the predicates it is used for. More precisely, the function  $F$  corresponding to this predicate must be *distributive*, in the sense that

$$F(A \cup B) = F(A) \cup F(B)$$

This is always guaranteed for (safe) predicates with *linear* recursion, that is predicates in which there is only one recursive call per disjunct in the disjunctive normal form representation. Such predicates form the overwhelming majority of recursive predicates, apart from artificial examples, and so this is not a great restriction. In any other cases the naive strategy may be used.

The semi-naive keeps a *frontier* of tuples that were added in the last step:

```

1  p = ∅
2  frontier = F(p)
3  while (frontier ≠ ∅)
4    p = p ∪ frontier
5    newFrontier = F(frontier)
6    frontier = newFrontier \ p

```

At each step, the current frontier is added to the accumulated relation, while the new frontier is computed by applying  $F$  to the frontier from the previous iteration. This is guaranteed to contain all new tuples, but may contain some tuples already in the accumulated in the relation  $p$ . The last statement of the loop therefore removes any such tuples. The algorithm stops when no more tuples can be added. A proof of correctness of this algorithm may be found in [28].

To illustrate semi-naive evaluation, the following shows its iterations for our example predicate:

Iteration	$p$	$newFrontier$	$frontier$
0	$\emptyset$	$q$	$q$
1	$q$	$q \cup q^2$	$q^2$
2	$q \cup q^2$	$q \cup q^3$	$q^3$
3	$q \cup q^2 \cup q^3$	$q \cup q^4$	$q^4$
4	$q \cup q^2 \cup q^3 \cup q^4$	$q \cup q^5$	$q^5$
...	...	...	...

This example illustrates the efficiency gain offered by semi-naive evaluation. While the accumulated relation  $p$  naturally grows at each iteration, the frontier remains relatively constant as it contains only new tuples. The efficiency gain arises because the possibly expensive function  $F$  is only applied to the frontier, while the accumulated  $p$  is only used in inexpensive union and difference operations. Semi-naive evaluation is therefore crucial to the efficient implementation of recursion in .QL.

## 5 Related Work

.QL builds on a wealth of previous work by others, and it is impossible to survey all of that here. We merely point out the highlights, and give sources for further reading.

### 5.1 Code Queries

The idea to use code queries for analysing source code has emerged from at least three different communities: software maintenance, program analysis and aspect-oriented programming. We discuss each of those below.

*Software maintenance.* As early as 1984, Linton proposed the use of a relational database to store programs [39]. His system was called Omega, and implemented on top of INGRES, a general database system with a query language named QUEL. Crucially, QUEL did not allow recursive queries, and as we have seen in these notes, recursion is indispensable when exploring the hierarchical structures that naturally occur in software systems. Furthermore, Linton already observed extremely poor performance. In retrospect, that is very likely to have been caused by the poor state of database optimisers in the 1980s. Furthermore, in the implementation of .QL, we have found it essential to apply a large number of special optimisations (which are proprietary to Semmler and the subject of patent applications) in the translation from .QL to SQL.

Linton's work had quite a large impact on the software maintenance community as witnessed by follow-up papers like that on CIA (the C Information Abstraction system) [13]. Today there are numerous companies that market products based on these ideas, usually under the banner of "application mining" or "application portfolio management". For instance, Paris-based CAST has a product named the 'Application Intelligence Platform' that stores a software system in a relational database [11]. Other companies offering similar products include ASG [3], BluePhoenix [9], EZLegacy [25], Metalec [43], Microfocus [44], Relativity [49] and TSRI [51]. A more light-weight system, which does however feature its own SQL-like query language (again, however, without recursion), is NDepend [55].

The big difference between SemmlerCode and all these other industrial systems is the emphasis on agility: with .QL, *all* quality checks are concise queries that can be adapted at will, by anyone involved in the development process. Some of the other systems mentioned above have however one big advantage over the free Java-only version of SemmlerCode: they offer parsers for many different languages, making it possible to store programs in relational form in the database. Indeed, large software systems are often heterogeneous, and so the same code query technology must work for many different object languages. We shall return to this point below.

Meanwhile, the drive for more expressive query languages, better suited to the application domain of searching code, gathered pace. Starting with the XL C++ Browser [32], many researchers have advocated the use of the logic programming language Prolog. In our view, there are several problems with the use of Prolog. First, it is notoriously difficult to predict whether Prolog queries terminate. Second, today's in-memory implementations of Prolog are simply not up to the job of querying the vast amounts of data in software systems. When querying the complete code for the *bonita* workflow system, the number of tuples gathered by SemmlerCode is 4,349,156. In a very recent paper, Costa has demonstrated that none of the leading Prolog implementations is capable of dealing with datasets of that size. That confirms our own experiments with the XSB system, reported in [29]. A few months ago, however, Kniesel *et al.* reported some promising preliminary experiments with special optimisations in a Prolog-based system for querying large software systems [35].

A modern system that uses logic programming for code querying is JQuery, a source-code querying plugin for Eclipse [30,42]. It uses a general-purpose language very similar to Prolog, but crucially, its use of tabling guarantees much better termination properties. It is necessary to annotate predicate definitions with mode annotations to achieve reasonable efficiency. We have resolutely excluded any such annotation features from .QL, leaving all the optimisation work to our compiler and the database optimiser. Despite the use of annotations, JQuery’s performance does not scale to substantial Java projects.

Instead of using a general logic programming language like Prolog, it might be more convenient to use a language that is more specific to the domain. For instance Consens *et al.* proposed GraphLog [16], a language for querying graph structures, and showed that it has advantages over Prolog in the exploration of software systems. Further examples of domain-specific languages for code search are the relational query algebra of Paul and Prakash [48], Jarzabek’s PQL [31] and Crew’s ASTLog [17]. A very recent proposal in this tradition is JTL (the Java Tools Language) of Cohen *et al.* [15]. Not only is this query language specific to code querying, it is specific to querying Java code. That has the advantage that some queries can be quite concise, with concrete Java syntax embedded in queries.

By contrast, there is nothing in .QL that is specific to the domain of code querying, because its designers preferred to have a simple, orthogonal language design. This is important if one wishes to use .QL for querying large, heterogeneous systems with artifacts in many different object languages. Furthermore, the creation of dedicated class libraries goes a long way towards tailoring .QL towards a particular domain. We might, however, consider the possibility of allowing the embedding of shorthand syntax in scripts themselves. There is a long tradition of allowing such user-defined syntactic extensions in a query language, for instance [10].

*Program Analysis.* Somewhat independently, the program analysis community has also explored the use of logic programming, for dataflow analyses rather than the structural analyses of the software maintenance community. The first paper to make that connection is one by Reps [50], where he showed how the use of the so-called ‘magic sets’ transformation [7] helps in deriving demand-driven program analyses from specifications in a restricted subset of Prolog, called *Datalog* (the variant of Datalog employed here incorporates certain extensions, *e.g.* expressions and aggregates).

Dawson *et al.* [19] demonstrate how many analyses can be expressed conveniently in Prolog — assuming it is executed with tabling (like JQuery mentioned above). Much more recently Michael Eichberg *et al.* demonstrated how such analyses can be incrementalised directly, using existing techniques for incrementalisation of logic programs [23]. While this certainly improves response times in an interactive environment for small datasets, it does not overcome the general scalability problem with Prolog implementations outlined above.

Whaley *et al.* [37,59] also advocate the use of Datalog to express program analyses. However, their proposed implementation model is completely different,

namely Binary Decision Diagrams (BDDs). This exploits the fact that in many analyses, there are a large number similar sets (for instance of allocation sites), and BDDs can exploit such similarity by sharing in their representation. Lhoták *et al.* [38] have independently made the same observation; their system is based on relational algebra rather than Datalog.

We have not yet experimented with expressing these types of program analysis in .QL, because the Eclipse plugin does not yet store information about control flow.

*Aspect-oriented programming.* Of course all these independent developments have not gone unnoticed, and many of the ideas are brought together in research on aspect-oriented programming. Very briefly, an ‘aspect’ instruments certain points in program execution. To identify those points, one can use any of the search techniques reviewed above.

One of the pioneers who made the connection between code queries and aspects was de Volder [20]. More recently, others have convincingly demonstrated that indeed the use of logic programming is very attractive for identifying the instrumentation points [27,47]. A mature system building on these ideas is LogicAJ [52]. In [5], the patterns used in AspectJ (an aspect-oriented extension of Java) are given a semantics by translation into Datalog queries [5].

The connection is of course also exploited in the other direction, suggesting new query mechanisms based on applications in aspect-oriented programming. For example, in [24], Eichberg *et al.* propose that XQuery is an appropriate notation for expressing many of the queries that arise in aspects. It appears difficult, however, to achieve acceptable performance on large systems, even with considerable effort [22].

Earlier this year, Morgan *et al.* proposed a static aspect language for checking design rules [45], which is partly inspired by AspectJ, marrying it with some of the advantages of code querying systems. In many ways, it is similar to JTTL, which we mentioned above. Like JTTL, it is tailored to the particular application, allowing concrete object syntax to be included in the queries. As said earlier, because large software systems are often heterogeneous, written in many different languages, we believe the query language itself should not be specific to the object language. Many users of .QL at first believe it to be domain-specific as well, because of the library of queries that tailor it to a particular application such as querying Java in Eclipse.

## 5.2 Object-Oriented Query Languages

.QL is a general query language, and we have seen how one can build a class library on top of any relational database schema, by annotating its fields with column types. There exists a long tradition of research on object-oriented query languages, so it behooves us to place .QL in that context.

In the 1980s, there was a surge of interest in so-called *deductive databases*, which used logic programming as the query language. The most prominent of these query languages was Datalog, which we mentioned above. In essence,

Datalog is Prolog, but without any data structures [26]; it thus lacks any object-oriented features.

Since the late 80s saw a boom in object-oriented programming, it was only natural that many attempts were made to integrate the idea of deductive databases and objects. Unfortunately a smooth combination turned out to be hard to achieve, and in a landmark paper, Ullman [57] even went so far as to state that a perfect combination is impossible.

Abiteboul *et al.* [1] proposed a notion of ‘virtual classes’ that is somewhat reminiscent of our normal classes [1]. However, the notion of dispatch is very different, using a ‘closest match’ rather than the ‘root definitions’ employed in .QL. Their definition of dispatch leads to brittle queries, where the static type of the receiver can significantly change the result. In our experience, such a design makes the effective use of libraries nearly impossible.

Most later related work went into modelling the notion of object-identity in the framework of a query language, *e.g.* [2,40]. In .QL that question is side-stepped because there is no object identity: a class is just a logical property. From that then follows the definition of inheritance as conjunction, and the disarmingly simple definition of virtual dispatch. Previous works have had much difficulty in defining an appropriate notion of multiple inheritance: here it is just conjunction.

## 6 Conclusion

We have presented .QL, a general object-oriented query language, through the particular application of software quality assessment. While this is the only concrete application we discussed, it was shown how, through appropriate annotation of the fields in a normal relational database schema with column types, one can build a library of queries on top of any relational database.

The unique features of .QL include its class mechanism (where inheritance is just logical ‘and’), its notion of virtual method dispatch, nondeterministic expressions, and its adoption of Dijkstra’s quantifier notation for aggregates. Each of these features contributes to the fun of playing with queries in .QL.

We hope to have enthused the reader into further exploring the use of .QL. A rich and interesting application area is the encoding of rules that are specific to an application domain. We have already done so for J2EE rules [53], but that only scratches the surface. Another application, which we hinted at in one of the exercises, is the use of .QL to identify opportunities for refactoring.

## References

1. Abiteboul, S., Lausen, G., Uphoff, H., Waller, E.: Methods and rules. In: Buneman, P., Jaodia, S. (eds.) Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, pp. 32–41. ACM Press, New York (1993)
2. Afrati, F.N.: On inheritance in object oriented datalog. In: International Workshop on Issues and Applications of Database Technology (IADT), pp. 280–289 (1998)

3. ASG. ASG-becubic<sup>TM</sup> for understanding and managing the enterprise's application portfolio. Product description on company website (2007), [http://asg.com/products/product\\_details.asp?code=BSZ](http://asg.com/products/product_details.asp?code=BSZ)
4. Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: An extensible AspectJ compiler. In: Rashid, A., Akşit, M. (eds.) *Transactions on Aspect-Oriented Software Development*. LNCS, vol. 3880, pp. 293–334. Springer, Heidelberg (2006)
5. Avgustinov, P., Hajiyev, E., Ongkingco, N., de Moor, O., Sereni, D., Tibble, J., Verbaere, M.: Semantics of static pointcuts in AspectJ. In: Felleisen, M. (ed.) *Principles of Programming Languages (POPL)*, pp. 11–23. ACM Press, New York (2007)
6. Balbin, I., Ramamohanarao, K.: A generalization of the differential approach to recursive query evaluation. *Journal of Logic Programming* 4(3), 259–262 (1987)
7. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.D.: Magic sets and other strange ways to implement logic programs. In: *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, 1986, pp. 1–16. ACM Press, New York (1986)
8. Basili, V., Brand, L., Melo, W.: A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering* 22(10), 751–760 (1996)
9. BluePhoenix. IT discovery. Product description available from company (2004), <http://www.bphx.com/Discovery.cfm>
10. Cardelli, L., Matthes, F., Abadi, M.: Extensible grammars for language specialization. In: Beeri, C., Ogori, A., Shasha, D. (eds.) *Database Programming Languages*, pp. 11–31. Springer, Heidelberg (1993)
11. Cast. Application intelligence platform. Product description on company website at, <http://www.castsoftware.com> (2007)
12. Checkstyle. Eclipse-cs: Eclipse checkstyle plug-in. Documentation and download at, <http://eclipse-cs.sourceforge.net/> (2007)
13. Chen, Y., Nishimoto, M., Ramamoorthy, C.V.: The C information abstraction system. *IEEE Transactions on Software Engineering* 16(3), 325–334 (1990)
14. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering* 20(6), 476–493 (1994)
15. Cohen, T., Gil, J., Maman, I.: JTL - the Java Tools Language. In: *21st Annual Conference on Object-oriented Programming, systems languages and applications (OOPSLA 2006)*, pp. 89–108. ACM Press, New York (2006)
16. Consens, M., Mendelzon, A., Ryman, A.: Visualizing and querying software structures. In: *ICSE 1992: Proceedings of the 14th international conference on Software engineering*, pp. 138–156. ACM Press, New York (1992)
17. Crew, R.F.: ASTLOG: A language for examining abstract syntax trees. In: *USENIX Conference on Domain-Specific Languages*, pp. 229–242 (1997)
18. Darcy, D.P., Slaughter, S.A., Kemerer, C.F., Tomayko, J.E.: The structural complexity of software: an experimental test. *IEEE Transactions on Software Engineering* 31(11), 982–995 (2005)
19. Dawson, S., Ramakrishnan, C.R., Warren, D.S.: Practical program analysis using general purpose logic programming systems. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 117–126. ACM Press, New York (1996)
20. d. Volder, K.: Aspect-oriented logic meta-programming. In: Cointe, P. (ed.) *Reflection 1999*. LNCS, vol. 1616, pp. 250–272. Springer, Heidelberg (1999)



21. Dijkstra, E.W., Scholten, C.S.: Predicate Calculus and Program Semantics. Texts and Monographs in Computer Science. Springer, Heidelberg (1990)
22. Eichberg, M.: Open Integrated Development and Analysis Environments. PhD thesis, Technische Universität Darmstadt (2007), <http://elib.tu-darmstadt.de/diss/000808/>
23. Eichberg, M., Kahl, M., Saha, D., Mezini, M., Ostermann, K.: Automatic incrementalization of prolog based static analyses. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 109–123. Springer, Heidelberg (2007)
24. Eichberg, M., Mezini, M., Ostermann, K.: Pointcuts as functional queries. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 366–381. Springer, Heidelberg (2004)
25. EZLegacy. EZ Source<sup>TM</sup>. Product description on company website at, <http://www.ezlegacy.com> (2007)
26. Gallaire, H., Minker, J.: Logic and Databases. Plenum Press, New York (1978)
27. Gybels, K., Brichau, J.: Arranging language features for more robust pattern-based crosscuts. In: 2nd International Conference on Aspect-Oriented Software Development, pp. 60–69. ACM Press, New York (2003)
28. Hajiyev, E.: CodeQuest: Source Code Querying with Datalog. MSc Thesis, Oxford University Computing Laboratory (September 2005), <http://progtools.comlab.ox.ac.uk/projects/codequest/>
29. Hajiyev, E., Verbaere, M., de Moor, O.: CodeQuest: scalable source code queries with Datalog. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 2–27. Springer, Heidelberg (2006)
30. Janzen, D., de Volder, K.: Navigating and querying code without getting lost. In: 2nd International Conference on Aspect-Oriented Software Development, pp. 178–187 (2003)
31. Jarzabek, S.: Design of flexible static program analyzers with PQL. IEEE Transactions on Software Engineering 24(3), 197–215 (1998)
32. Javey, S., Mitsui, K., Nakamura, H., Ohira, T., Yasuda, K., Kuse, K., Kamimura, T., Helm, R.: Architecture of the XL C++ browser. In: CASCON 1992: Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research, pp. 369–379. IBM Press (1992)
33. JFreeChart. Website with documentation and downloads (2007), <http://www.jfree.org/jfreechart/>
34. Kaldewaij, A.: The Derivation of Algorithms. Prentice-Hall, Englewood Cliffs (1990)
35. Kniesel, G., Hannemann, J., Rho, T.: A comparison of logic-based infrastructures for concern detection and extraction. In: LATE R 2007 – Linking Aspect Technology and Evolution. ACM, New York (2007), <http://www.cs.uni-bonn.de/~gk/papers/knieselHannemannRho-late07.pdf>
36. Lakos, J.: Large-Scale C++ Software Design. Addison-Wesley, Reading (1996)
37. Lam, M.S., Whaley, J., Livshits, V.B., Martin, M.C., Avots, D., Carbin, M., Unkel, C.: Context-sensitive program analysis as database queries. In: Proceedings of PODS, pp. 1–12. ACM Press, New York (2005)
38. Lhoták, O., Hendren, L.: Jedd: A BDD-based relational extension of Java. In: Programming Language Design and Implementation (PLDI), pp. 158–169 (2004)
39. Linton, M.A.: Implementing relational views of programs. In: Henderson, P.B. (ed.) Software Development Environments (SDE), pp. 132–140 (1984)
40. Liu, M., Dobbie, G., Ling, T.W.: A logical foundation for deductive object-oriented databases. ACM Transactions on Database Systems 27(1), 117–151 (2002)



41. Martin, R.C.: Agile Software Development, Principles, Patterns and Practices. Prentice-Hall, Englewood Cliffs (2002)
42. McCormick, E., De Volder, K.: JQuery: finding your way through tangled code. In: Companion to OOPSLA, pp. 9–10. ACM Press, New York (2004)
43. Metalect. IQ server. Product description on company website at, <http://www.metalect.com/what-we-offer/technology/> (2007)
44. MicroFocus. Application portfolio management. Product description on company website at, <http://www.microfocus.com/Solutions/APM/> (2007)
45. Morgan, C., De Volder, K., Wohstadter, E.: A static aspect language for checking design rules. In: De Moor, O. (ed.) Aspect-Oriented Software Development (AOSD 2007), pp. 63–72 (2007)
46. Nystrom, N., Clarkson, M.R., Myers, A.C.: Polyglot: An extensible compiler framework for Java. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 138–152. Springer, Heidelberg (2003)
47. Ostermann, K., Mezini, M., Bockish, C.: Expressive pointcuts for increased modularity. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 214–240. Springer, Heidelberg (2005)
48. Paul, S., Prakash, A.: Querying source code using an algebraic query language. IEEE Transactions on Software Engineering 22(3), 202–217 (1996)
49. Relativity. Application analyzer<sup>TM</sup>. Product description on company website at, <http://www.relativity.com/pages/applicationanalyzer.asp> (2007)
50. Reps, T.W.: Demand interprocedural program analysis using logic databases. In: Ramakrishnan, R. (ed.) Applications of Logic Databases. International Series in Engineering and Computer Science, vol. 296, pp. 163–196. Kluwer, Dordrecht (1995)
51. The Software Revolution. Janus technology<sup>TM</sup>. Product description on company website (2007), <http://www.softwarerevolution.com/>
52. Rho, T., Kniesel, G., Appeltauer, M., Linder, A.: LogicAJ (2006), <http://roots.iai.uni-bonn.de/research/logicaj/people>
53. Semmler Ltd. Company website with free downloads, documentation, and discussion forums (2007), <http://semmler.com>
54. Semmler Ltd. Installation instructions for this tutorial (2007), <http://semmler.com/gttse-07>
55. Smacchia, P.: NDepend. Product description on company website at, <http://www.ndepend.com> (2007)
56. Spinellis, D.D.: Code Quality: the Open Source Perspective. Addison-Wesley, Reading (2007)
57. Ullman, J.D.: A comparison between deductive and object-oriented database systems. In: 2nd International Conference on Deductive and Object-Oriented Databases. Springer Lecture Notes in Computer Science, pp. 263–277 (1991)
58. van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. Journal of the ACM 38(3), 620–650 (1991)
59. Whaley, J., Avots, D., Carbin, M., Lam, M.S.: Using datalog and binary decision diagrams for program analysis. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 97–118. Springer, Heidelberg (2005)

## Appendix: Answers to the Exercises

*Exercise 1.* Run the query to find suspicious declarations of `compareTo` in `JFreeChart`. You can do that in a number of ways, but here the nicest way

to look at the results is as a table, so use the run button marked with a table at the top right-hand side of the Quick Query window. You will get two results, and you can navigate to the relevant locations in the source by double-clicking. Are both of them real bugs? ♡

*Answer:* The table is not shown here. One of the matches, the class named `PieLabelRecord`, is indeed an example where `compareTo` and `equals` are inconsistent. The `compareTo` method will return 0 whenever the `baseY` values are equal, but `equals` is inherited from `Object` and so compares object identity. The other match `Outlier` is not a bug: in fact consistency between `equals` and `compareTo` is clearly assured because `compareTo` calls `equals`.

---

*Exercise 2.* Write a query to find all methods named `main` in packages whose names end with the string `demo`. You may find it handy to use the predicate `string.matches("%demo")` (as is common in query languages, `%` is a wildcard matching any string). ♡

*Answer:* We want to select a method, so that is what goes in the **from** clause. Next, we want those methods to have name “main” and occur in a package with a name that matches the given pattern. Note the repeated use of `dispatch` on the result of methods. If you tried to write the same query in Prolog, you would have to give a name to each of those intermediate results, considerably cluttering the query.

```

from Method m
where m.hasName(" main") and
    m.getDeclaringType().getPackage().getName().matches("%demo")
select m.getDeclaringType().getPackage(),
    m.getDeclaringType(),
    m

```

---

*Exercise 3.* The above queries show how to find types that define a method named “equals”, and how to find types that do not have such a method. Write a query picking out types that define at least *one* method which is not called “equals”. ♡

*Answer:* This query is more verbose, but straightforward. We use **exists** to find a method and test that its name is not “equals”:

```

from Class c
where exists (Method m | m = c.getACallable()
    and not (m.hasName("equals")))
select m

```

Note that `getACallable` returns several results, so this succeeds if at least one of the methods is not called “equals”.

*Exercise 4.* Continuing Exercise 2.1. You will have found that one class represents a real bug, whereas the other does not. Refine our earlier query to avoid such false positives. ♡

*Answer:* We exclude declarations of `compareTo` that make a call to `equals`:

```

from Class c, Method compare
where compare.getDeclaringType()=c and
      compare.hasName("compareTo") and
      not(c.declaresMethod("equals")) and
      not(compare.getACall().hasName("equals"))
select c.getPackage(),c,compare

```

An interesting point concerns the fact that the method `getACall` is nondeterministic. Negating the nondeterministic call means that *none* of the methods called by `compare` has name "equals".

*Exercise 5.* Write a query to find all types in `JFreeChart` that have a field of type `JFreeChart`. Many of these are test cases; can they be excluded somehow? ♡

*Answer:* Inspecting the results of the obvious query (the one below without the extra conjunct in the **where** clause), it is easy to see that all of the test cases are in fact subtypes of `TestCase`, so that is the condition we use to exclude them:

```

from RefType t
where t.getAField().getType().hasName("JFreeChart")
      and
      not t.getASupertype().hasName("TestCase")
select t

```

*Exercise 6.* There exists a method named `getASuperType` that returns *some* supertype of its receiver, and sometimes this is a convenient alternative to using `hasSubtype`. Uses of methods such as `getASuperType` that return an argument can be chained too. Using `x.getASuperType*()`, write a query for finding all subtypes of `org.jfree.chart.plot.Plot`. Try to use no more than one variable. ♡

*Answer:* Again, note how the use of nondeterministic methods leads to very concise queries:

```

from RefType s
where s.getASuperType*().hasName("Plot")
select s

```

---

*Exercise 7.* When a query returns two program elements plus a string you can view the results as an edge-labelled graph by clicking on the graph button (shown below). To try out that feature, use chaining to write a query to depict the hierarchy above the type `TaskSeriesCollection` in package `org.jfree.data.gantt`. You may wish to exclude `Object` from the results, as it clutters the picture. Right-clicking on the graph view will give you a number of options for displaying it. ♡

*Answer:* First, find the `TaskSeriesCollection` type, and name it *tsc*. Now we want to find pairs *s* and *t* that are supertypes of *tsc*, such that furthermore *t* is a direct supertype of *s*. Finally, we don't want to consider `Object`, so that is our final conjunct. If we now select the pair (*s*, *t*) that becomes an edge in the depicted graph:

```

from RefType tsc, RefType s, RefType t
where tsc.hasQualifiedName("org.jfree.data.gantt", "TaskSeriesCollection")
      and
      s.hasSubtype*(tsc)
      and
      t.hasSubtype(s)
      and
      not(t.hasName("Object"))
select s, t

```

---

*Exercise 8.* Display the results of the above query as pie chart, where each slice of the pie represents a package and the size of the slice the average number of methods in that package. To do so, use the *run* button marked with a chart, and select 'pie chart' from the drop-down menu. ♡

*Answer:* No comment; just an exercise to play with!

---

*Exercise 9.* Not convinced that metrics are any good? Run the above query; it will be convenient to display the results as a bar chart, with the bars in descending order. To achieve that sorting, add "**as s order by s desc**" at the end. Now carefully inspect the packages with high instability. Sorting the other way round (using **asc** instead of **desc**) allows you to inspect the stable packages. ♡

*Answer:* The most unstable packages are precisely the *experimental* ones in JFreeChart. The most stable package of all is `java.lang`. Amazing that such a simple metric can make such accurate predictions!

*Exercise 10.* The following questions are intended to help reinforce some of the subtle points about aggregates; you could run experiments with SemmlCode to check them, but really they're just for thinking.

1. What is `sum(int i | i = 0 or i = 0|2)`?
2. Under what conditions on  $p$  and  $q$  is this a true equation?

$$\text{sum}(\text{int } i \mid p(i) \text{ or } q(i)) = \text{sum}(\text{int } i \mid p(i)) + \text{sum}(\text{int } i \mid q(i)) \quad \heartsuit$$

*Answer:*

1. It's just 2. You can use normal logical equivalences to manipulate the range condition in an aggregate.
2. This equation is true only if  $p$  and  $q$  are disjoint, that is:  $\forall i : \neg(p(i) \wedge q(i))$ .

*Exercise 11.* Queries can be useful for identifying refactoring opportunities. For example, suppose we are interested in finding pairs of classes that could benefit by extracting a common interface or by creating a new common superclass.

1. As a first step, we will need to identify *root definitions*: methods that are not overriding some other method in the superclass. Define a new .QL class named `RootDefMethod` for such methods. It only needs to have a constructor, and no methods or predicates.
2. Complete the body of the following classless predicate:

```
predicate similar(RefType t, RefType s, Method m, Method n) { ... }
```

It should check that  $m$  is a method of  $t$ ,  $n$  is a method of  $s$ , and  $m$  and  $n$  have the same signature.

3. Now we are ready to write the real query: find all pairs  $(t, s)$  that are in the same package, and have more than one root definition in common. All of these are potential candidates for refactoring. If you have written the query correctly, you will find two types in `JFreeChart` that have 99 root definitions in common.
4. Write a query to list those 99 commonalities. ♥

*Answer:*

1. The class for root definitions is:

```
class RootDefMethod extends Method {  
    RootDefMethod() { not exists(Method m | overrides(this, m)) }  
}
```

2. The definition of the predicate can be completed as follows:

```
predicate similar(RefType t, RefType s, Method m, Method n) {
  m.getDeclaringType() = t and n.getDeclaringType() = s
  and m.getSignature() = n.getSignature()
}
```

3. Finally, the required query is shown below. To try out the answer, just type the class definition, the predicate and the query all together in the Quick Query window. (Warning: this query takes a while to execute.)

```
from RefType t, RefType s, int c
where t.getPackage() = s.getPackage()
  and
  t.getQualifiedName() < s.getQualifiedName()
  and
  c = count(RootDefMethod m, RootDefMethod n | similar(t,s,m,n))
  and
  c > 1
select c, t.getPackage(), t,s order by c desc
```

4. This is a simple re-use of the predicate *similar* defined above:

```
from RefType t, RefType s, RootDefMethod m, RootDefMethod n
where t.hasName("CategoryPlot") and s.hasName("XYPlot")
  and
  t.getPackage() = s.getPackage()
  and
  similar(t,s,m,n)
select m,n
```

*Exercise 12.* We now explore the use of factories in JFreeChart.

1. Write a query to find types in JFreeChart whose name contains the string "Factory."
2. Write a class to model the Java type `JFreeChart` and its subtypes.
3. Count the number of constructor calls to such types.
4. Modify the above query to find violations in the use of a `ChartFactory` to construct instances of `JFreeChart`.
5. There are 53 such violations; it is easiest to view them as a table. The interesting ones are those that are not in tests or demos. Inspect these in detail — they reveal a weakness in the above example, namely that we may also wish to make an exception for *this* constructor calls. Modify the code to include that exception. Are all the remaining examples tests or demos? ♥

*Answer:*

1. Here is a query to find factories in JFreeChart:

```

from RefType t
where t.getName().matches("%Factory%")
select t

```

We shall use the first result, `ChartFactory`, in the remainder of this exercise.

- The class just has a constructor and no methods or predicates. The constructor says that **this** has a supertype named `JFreeChart`. If desired, that could be refined by using a qualified name rather than a simple name.

```

class JFreeChart extends RefType {
  JFreeChart() { this.getASupertype*().hasName("JFreeChart") }
}

```

- We want calls where the callee is a constructor of a *JFreeChart* type:

```

select count(Call c | c.getCallee() instanceof Constructor and
  c.getCallee().getDeclaringType() instanceof JFreeChart)

```

A shorter alternative (which does however require you to know the class hierarchy quite well) is

```

select count(ConstructorCall c | c.getCallee().getDeclaringType()
  instanceof
  JFreeChart)

```

The answer is 88.

- The definitions are very similar to the ones in the *ASTFactory* example:

```

class ChartFactory extends RefType {
  ChartFactory() { this.getASupertype*().hasName("ChartFactory") }
  ConstructorCall getAViolation() {
    result.getType() instanceof JFreeChart and
    not(result.getCaller().getDeclaringType()
      instanceof ChartFactory) and
    not(result instanceof SuperConstructorCall)
  }
}

```

```

from ChartFactory f, Call c
where c = f.getAViolation()
select c.getCaller().getDeclaringType().getPackage(),
  c.getCaller().getDeclaringType(),
  c.getCaller(),
  c

```

- Change the *getAViolation* definition to:

```

ConstructorCall getAViolation() {
  result.getType() instanceof JFreeChart and
  not(result.getCaller().getDeclaringType()
    instanceof ChartFactory) and

```

```

not(result instanceof SuperConstructorCall or
      result instanceof ThisConstructorCall)
}

```

No, there are still two matches in the package `org.jfree.chart.plot`. One of them says “An initial quick and dirty”; both matches seem to be real mistakes. The other 49 are all in packages that do not use the factory at all, so that is probably intended.

*Exercise 13.* The above definition of `getLevel` is in the default library; write queries to display barcharts. Do the high points indeed represent components that you would consider high-level? For types, write a query that calculates how many classes that have maximum level do not define a method named “main”. ♡

*Answer:* The level metric is surprisingly effective in finding components that are high-level in the intuitive sense.

```

from MetricPackage p, float c
where p.fromSource() and c = p.getLevel()
select p, c order by c desc

```

The following query calculates what proportion of the highest-level types do not define a method named “main”:

```

predicate maxLevel(MetricRefType t) {
  t.fromSource() and
  t.getLevel() = max(MetricRefType t | | t.getLevel())
}

```

```

from float i, float j
where
  i = count(MetricRefType t | maxLevel(t) and
            not(t.getACallable().hasName("main")))
  and
  j = count(MetricRefType t | maxLevel(t))
select i/j

```

About 24% of high-level matches do not define a “main” method.

*Exercise 14.* Suppose the class `OnlyTwo` does not override `foo`. Does that make sense? What does the `.QL` implementation do in such cases? ♡

*Answer:* There is then a choice of two different implementations that could be overridden. At first it might seem that it makes sense to take their disjunction, but clearly that is wrong as subclassing means conjunction. The implementation forbids such cases and insists that `foo` be overridden to ensure a unique definition is referenced.



---

*Exercise 15.* Construct an example to demonstrate how dispatch depends on the static type of the receiver. ♡

*Answer:* We need two root definitions that have the same signature. For instance, in the class hierarchy below, there are root definitions of *foo* both in class *B* and in class *C*:

```
class A {
  A() { this=1 }
  string toString() { result="A" }
}
```

```
class B extends A {
  string foo() { result="B" }
}
```

```
class C extends A {
  string foo() { result="C" }
}
```

```
from C c select c.foo()
```

The answer of the query is just “C”. If *foo* was also declared in class *A*, then that would be the single root definition, and “B” would also be an answer.

---

*Exercise 16.* Extend the above class definition with *getDeclaringType*. ♡

*Answer:* The definition of *getDeclaringType* is just a minor variation on the definition of *getName* we saw earlier:

```
class MyMethod extends @method {
  string getName() { methods(this,result,,-,-,-,-,-) }
  string toString() { result = this.getName() }
  RefType getDeclaringType() { methods(this,,-,-,-,-,-,result,,-,-) }
}
```

# Transforming Data by Calculation

José N. Oliveira

CCTC, Universidade do Minho, 4700-320 Braga, Portugal  
jno@di.uminho.pt

**Abstract.** This paper addresses the foundations of data-model transformation. A catalog of *data mappings* is presented which includes abstraction and representation relations and associated constraints. These are justified in an algebraic style via the *pointfree-transform*, a technique whereby predicates are lifted to binary relation terms (of the algebra of programming) in a two-level style encompassing both data and operations. This approach to data calculation, which also includes transformation of recursive data models into “flat” database schemes, is offered as alternative to standard database design from abstract models. The calculus is also used to establish a link between the proposed transformational style and bidirectional *lenses* developed in the context of the classical *view-update problem*.

**Keywords:** Theoretical foundations, mapping scenarios, transformational design, refinement by calculation.

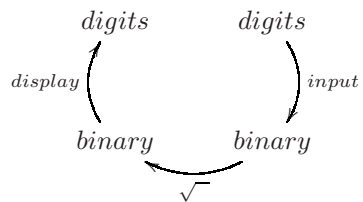
## 1 Introduction

Watch yourself using a pocket calculator: every time a digit key is pressed, the corresponding digit is displayed on the LCD display once understood by the calculator, a process which includes representing it internally in binary format:



This illustrates the main ingredients of one’s everyday interaction with machines: the abstract objects one has in mind (eg. digits, numbers, etc) need to be *represented* inside the machine before this can perform useful calculations, eg. square root, as displayed in the diagram below.

However, it may happen that our calculator is faulty. For instance, sometimes the digit displayed is not the one whose key was just pressed; or *nothing* at all is displayed; or even the required operation (such as triggered by the square root key) is not properly computed. It is the designer’s responsibility to ensure that the machine we are using never misbehaves and can thus be trusted.



When using machines such as computers or calculators, one is *subcontracting* mechanical services. Inside the machine, the same subcontracting process happens again and again: complex routines accomplish their tasks by subcontracting (simpler) routines, and so on and so forth. So, the data representation process illustrated above for the (interaction with a) pocket calculator happens inside machines every time a routine is called: input data are to be made available in the appropriate format to the subcontracted routine, the result of which may need to change format again before it reaches its caller.

Such data *represent/retrieve* processes (analogue to the *input/display* process above) happen an uncountable number of times even in simple software systems. *Subcontracting* thus being the essence of computing (as it is of any organized society), much trouble is to be expected once *represent/retrieve* contracts fail: the whole service as subcontracted from outside is likely to collapse.

Three kinds of fault have been identified above: loss of data, confusion among data and wrong computation. The first two have to do with *data representation* and the third with *data processing*. Helping in preventing any of these from happening in software designs is the main aim of this paper.

We will see that most of the work has to do with *data transformation*, a technique which the average programmer is often unaware of using when writing, most often in an ‘ad hoc’ way, middleware code to “bridge the gap” between two different technology layers. The other part of the story — ensuring the overall correctness of software subcontracts — has to do with *data refinement*, a well established branch of the software sciences which is concerned with the relationship between (stepwise) specification and implementation.

*Structure of the paper.* This paper is organized as follows. Section 2 presents the overall spirit of the approach and introduces a simple running example. Section 3 reviews the binary relation notation and calculus, referred to as the *pointfree (PF) transform*. Section 4 shows how to denote the meaning of data in terms of such unified notation. Section 5 expresses data impedance mismatch in the PF-style. While sections 6 to 8 illustrate the approach in the context of (database) relational modeling, recursive data modeling is addressed from section 9 onwards. Then we show how to handle cross-paradigm impedance by calculation (section 10) and how to transcribe operations from recursive to flat data models (section 11). Section 12 addresses related work. In particular, it establishes a link between data mappings and bidirectional *lenses* developed in the context of the *view-update problem* and reviews work on a library for data transformations (2LT) which is strongly related to the current paper. Finally, section 13 concludes and points out a number of research directions in the field.

*Technical sketch of the paper.* This text puts informal, technology dependent approaches to data transformation together with data calculation formalisms which are technology agnostic. It is useful to anticipate how such schools of thought are related along the paper, while pinpointing the key formal concepts involved.

The main motivation for data calculation is the need for *data-mappings* as introduced in section 2: one needs to ensure that data flow unharmed across the boundaries of software layers which use different technologies and/or adopt different data models. On

the technical side, this is handled (in section 2) by ordering data formats by degree of abstraction and writing  $A \leq B$  wherever format  $A$  is safely implemented by format  $B$ . Technically,  $\leq$  is a *preorder* and  $\leq$ -facts are witnessed by *relations* telling how data should flow back and forth between formats  $A$  and  $B$ .

The need for handling such relations in a compositional, calculational way leads to the relational calculus and the pointfree transform. The whole of section 3 is devoted to providing a summary of the required background, whose essence lies in a number of laws which can be used to calculate with relations directly (instead of using set theory to indirectly convey the same results). The fact that all relations are binary is not a handicap: they can be thought of as arrows of the form  $A \xrightarrow{R} B$  which express data flow in a natural way and can be composed with each other to express more complex data flows. Data filtering is captured by relations of a particular kind, known as *coreflexives*, which play a prominent role throughout the whole calculus.

The bridge between formal and informal data structuring becomes more apparent from section 4 onwards, where typical data structures are shown to be expressible not only in terms of abstract constructs such as Cartesian product ( $A \times B$ ), disjoint sum ( $A + B$ ) and equations thereof (as in the case of recursive types), but also in terms of typed finite relations, thus formalizing the way data models are recorded by entity-relationship diagrams or UML class diagrams, for instance.

Further to structure, *constraints* (also known as *invariants*) are essential to data modeling, making it possible to enforce semantic properties on data. Central to such data constraints is *membership*, a relation of type  $A \xleftarrow{\in} T A$  which is able to tell which data elements can be found in a particular data structure of shape  $T$ . The key ingredient at this point is the fact that set-theoretic membership can be extended to data containers other than sets.

Sections 5 and 6 are central to the whole paper: they show how to calculate complex data mappings by combining a number of  $\leq$ -rules which are proposed and justified using (pointfree) relation calculus. Compositionality is achieved in two ways: by transitivity, suitably typed  $\leq$ -rules can be chained; by monotonicity, they can be promoted from the parameters of a parametric type  $T$  to the whole type, for instance by inferring  $T A \leq T B$  from  $A \leq B$ . The key of the latter result consists in regarding  $T$  as a *relator*, a concept which traverses relation calculus from beginning to end and explains, in the current paper, data representation techniques such as those involving dynamic heaps and pointer dereferencing. On the practical side, a number of  $\leq$ -facts are shown to be applicable to calculating database schemata from abstract models (sections 6 and 7) and reasoning about entity-relationship diagrams (section 8).

Abstract (and language-based) data models often involve recursive data which pose challenges of their own to data mapping formalization. Sections 9 to 11 show how the calculus of fixpoint solutions to relational equations (known as *hylomorphisms*) offers a basis for refining recursive data structures. This framework is set to work in section 10 where it is applied to the paper's running example, the `PTree` recursive model of pedigree trees, which is eventually mapped onto a flat, non-recursive model, after stepping through a pointer-based representation. The layout of calculations not only captures the  $\leq$  relationships among source, intermediate and target data models, but

also the abstraction and representation relations implicit in each step, which altogether synthesize two overall ‘map forward’ and ‘map backward’ data transformations.

Section 11 addresses the *transcription level*, the third component of a *mapping scenario*. This has to do with refining operations whose input and output data formats have changed according to such big-step ‘map forward’ and ‘map backward’ transformations. Technically, this can be framed into the discipline of data refinement. The examples given, which range from transcribing a query over `PTree` down to the level of its flat version (obtained in section 10) to calculating low level operations handling heaps and pointers, show once again the power of data calculation performed relationally, and in particular the usefulness of so-called *fusion*-properties.

Finally, section 12 includes a sketch of how  $\leq$ -diagrams can be used to capture bidirectional (asymmetric) transformations known as *lenses* and their properties.

## 2 Context and Motivation

*On data representation.* The theoretical foundation of *data representation* can be written in few words: what matters is the *no loss/no confusion* principle hinted above. Let us explain what this means by writing  $c R a$  to denote the fact that *datum c represents datum a* (assuming that  $a$  and  $c$  range over two given data types  $A$  and  $C$ , respectively) and the converse fact  $a R^\circ c$  to denote that *a is the datum represented by c*. The use of definite article “*the*” instead of “*a*” in the previous sentence is already a symptom of the **no confusion** principle — we want  $c$  to represent *only one* datum of interest:

$$\langle \forall c, a, a' :: c R a \wedge c R a' \Rightarrow a = a' \rangle \quad (1)$$

The **no loss** principle means that no data are lost in the representation process. Put in other words, it ensures that every datum of interest  $a$  is representable by some  $c$ :

$$\langle \forall a :: \langle \exists c :: c R a \rangle \rangle \quad (2)$$

Above we mention the converse  $R^\circ$  of  $R$ , which is the relation such that  $a(R^\circ)c$  holds iff  $c R a$  holds. Let us use this rule in re-writing (1) in terms of  $F = R^\circ$ :

$$\langle \forall c, a, a' :: a F c \wedge a' F c \Rightarrow a = a' \rangle \quad (3)$$

This means that  $F$ , the converse of  $R$ , can be thought of as an *abstraction relation* which is *functional* (or deterministic): two outputs  $a, a'$  for the same input  $c$  are bound to be the same.

Before going further, note the notation convention of writing the outputs of  $F$  on the left hand side and its inputs on the right hand side, as suggested by the usual way of declaring functions in ordinary mathematics,  $y = f x$ , where  $y$  ranges over outputs (cf. the vertical axis of the Cartesian plane) and  $x$  over inputs (cf. the other, horizontal axis). This convention is adopted consistently throughout this text and is extended to relations, as already seen above 1.

<sup>1</sup> The fact that  $a F c$  is written instead of  $a = F c$  reflects the fact that  $F$  is not a total function, in general. See more details about notation and terminology in section 3.

Expressed in terms of  $F$ , (2) becomes

$$\langle \forall a :: \langle \exists c :: a F c \rangle \rangle \tag{4}$$

meaning that  $F$  is *surjective*: every abstract datum  $a$  is reachable by  $F$ . In general, it is useful to let the abstraction relation  $F$  to be larger than  $R^\circ$ , provided that it keeps properties (3,4) — being functional and surjective, respectively — and that it stays *connected* to  $R$ . This last property is written as

$$\langle \forall a, c :: c R a \Rightarrow a F c \rangle$$

or, with less symbols, as

$$R^\circ \subseteq F \tag{5}$$

by application of the rule which expresses relational inclusion:

$$R \subseteq S \equiv \langle \forall b, a :: b R a \Rightarrow b S a \rangle \tag{6}$$

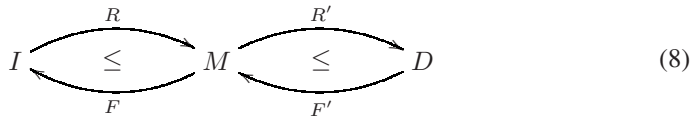
(Read  $R \subseteq S$  as “ $R$  is at most  $S$ ”, meaning that  $S$  is either more defined or less deterministic than  $R$ .)

To express the fact that  $(R, F)$  is a *connected* representation/abstraction pair we draw a diagram of the form



where  $A$  is the datatype of data *to be represented* and  $C$  is the chosen datatype of representations [2]. In the data refinement literature,  $A$  is often referred to as *the abstract type* and  $C$  as *the concrete one*, because  $C$  contains more information than  $A$ , which is *ignored* by  $F$  (a non-injective relation in general). This explains why  $F$  is referred to as *the abstraction relation* in a  $(R, F)$  pair.

*Layered representation.* In general, it will make sense to chain several layers of abstraction as in, for instance,

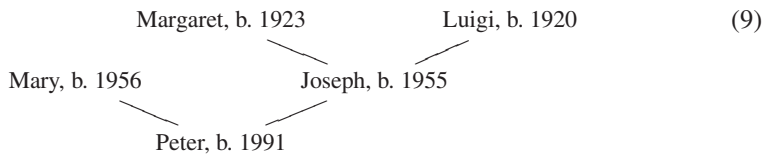


where letters  $I$ ,  $M$  and  $D$  have been judiciously chosen so as to suggest the words *interface*, *middleware* and *dataware*, respectively.

<sup>2</sup> Diagrams such as (7) should not be confused with commutative diagrams expressing properties of the relational calculus, as in eg. [11], since the ordering  $\leq$  in the diagram is an ordering on objects and not on arrows.

In fact, data become “more concrete” as they go down the traditional layers of software architecture: the contents of interactive, handy objects at the interface level (often pictured as trees, combo boxes and the like) become pointer structures (eg. in C++/C#) as they descend to the middleware, from where they are channeled to the data level, where they live as persistent database records. A popular picture of diagram (8) above is given in figure 1 where layers  $I$ ,  $M$  and  $D$  are represented by concentric circles.

As an example, consider an interface ( $I$ ) providing direct manipulation of pedigree trees, common in genealogy websites:



Trees — which are the users’ mental model of recursive structures — become pointer structures (figure 2a) once channeled to the middleware ( $M$ ). For archival purposes, such structures are eventually buried into the dataware level ( $D$ ) in the form of very concrete, persistent records of database files (cf. figure 2b).

Modeling pedigree trees will be our main running example throughout this paper.

*Mapping scenarios.* Once materialized in some technology (eg. XML, C/C++/Java, SQL, etc), the layers of figure 1 stay apart from each other in different programming paradigms (eg. markup languages, object-orientated databases, relational databases, etc) each requiring its own skills and programming techniques.

As shown above, different data models can be compared via abstraction/representation pairs. These are expected to be more complex once the two models under comparison belong to different paradigms. This kind of complexity is a measure of the *impedance mismatches between the various data-modeling and data-processing paradigms*<sup>3</sup>, in the words of reference [42] where a thorough account is given of the many problems which hinder software technology in this respect. Still quoting [42]:

*Whatever programming paradigm for data processing we choose, data has the tendency to live on the other side or to eventually end up there. (...) This myriad of inter- and intra-paradigm data models calls for a good understanding of techniques for mappings between data models, actual data, and operations on data. (...)*

<sup>3</sup> According to [3], the label *impedance mismatch* was coined in the early 1990’s to capture (by analogy with a similar situation in electrical circuits) the technical gap between the object and relational technologies. Other kinds of impedance mismatch are addressed in [42, 67].

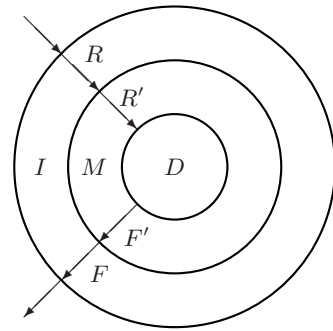


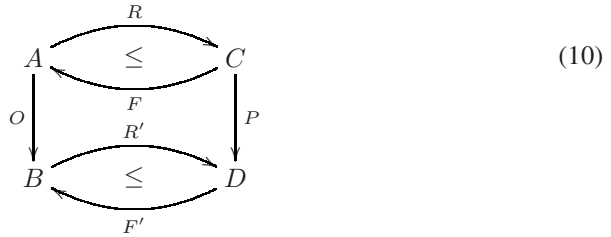
Fig. 1. Layered software architecture

Given the fact that IT industry is fighting with various impedance mismatches and data-model evolution problems for decades, it seems to be safe to start a research career that specifically addresses these problems.

The same reference goes further in identifying three main ingredients (levels) in *mapping scenarios*:

- The *type-level* mapping of a source data model to a target data model;
- Two maps (“map forward” and “map backward”) between source / target data;
- The *transcription level* mapping of source operations into target operations.

Clearly, diagram (7) can be seen as a succinct presentation of the two first ingredients, the former being captured by the  $\leq$ -ordering on data models and the latter by the  $(R, F)$  pair of relations. The third can easily be captured by putting two instances of (7) together, in a way such that the input and output types of a given operation, say  $O$ , are *wrapped* by forward and backward data maps:



The (safe) transcription of  $O$  into  $P$  can be formally stated by ensuring that the picture is a commutative diagram. A typical situation arises when  $A$  and  $B$  are the same (and so are  $C$  and  $D$ ), and  $O$  is regarded as a state-transforming operation of a software component, eg. one of its CRUD (“Create, Read, Update and Delete”) operations. Then the diagram will ensure correct refinement of such an operation across the change of state representation.

*Data refinement.* The theory behind diagrams such as (10) is known as *data refinement*. It is among the most studied formalisms in software design theory and is available from several textbooks — see eg. [20, 38, 49].

The fact that state-of-the-art software technologies don’t enforce such formal design principles in general leads to the unsafe technology which we live on today, which is hindered by permanent cross-paradigm impedance mismatch, loose (untyped) data mappings, unsafe CRUD operation transcription, etc. Why is this so? Why isn’t data refinement widespread? Perhaps because it is far too complex a discipline for most software practitioners, a fact which is mirrored on its prolific terminology — cf. *downward, upward* refinement [31], *forwards, backwards* refinement [31, 48, 70], *S, SP, SC*-refinement [21] and so on. Another weakness of these theories is their reliance on *invent & verify (proof)* development strategies which are hard to master and get involved once facing “real-sized” problems. What can we do about this?

The approach we propose to follow in this paper is different from the standard in two respects: first, we adopt a *transformational* strategy as opposed to invention-followed-by-verification; second, we adopt a *calculational* approach throughout our data transformation steps. What do we mean by “calculational”?



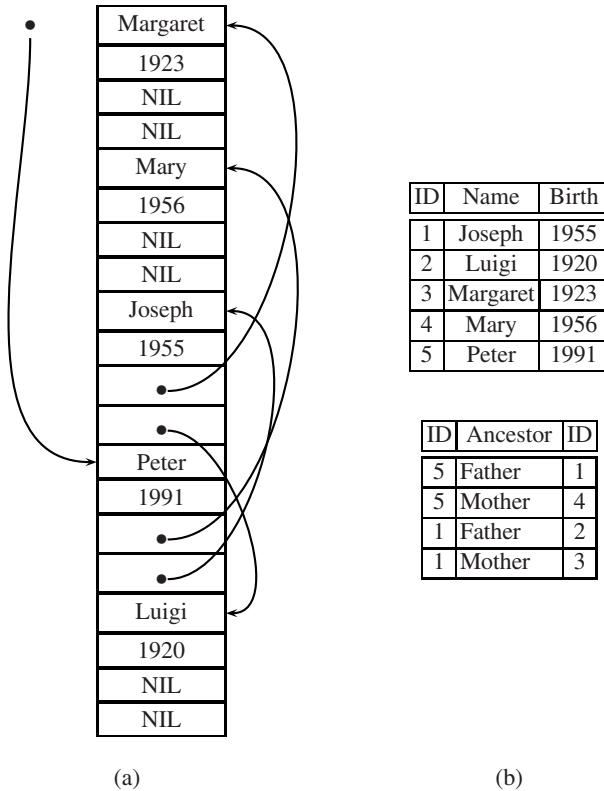


Fig. 2. Middleware (a) and dataware (b) formats for family tree sample data [9]

*Calculational techniques.* Let us briefly review some background. The idea of using mathematics to reason about and transform programs is an old one and can be traced back to the times of McCarthy's work on the foundations of computer programming [46] and Floyd's work on program meaning [26]. A so-called *program transformation* school was already active in the mid 1970s, see for instance references [16, 19]. But program transformation becomes *calculational* only after the inspiring work of J. Backus in his *algebra of (functional) programs* [7] where the emphasis is put on the calculus of functional combinators rather than on the  $\lambda$ -notation and its variables, or *points*. This is why Backus' calculus is said to be *point-free*.

Intensive research on the (pointfree) program calculation approach in the last thirty years has led to the *algebra of programming* discipline [5, 11]. The priority of this discipline has been, however, mostly on reasoning about *algorithms* rather than *data structures*. Our own attempts to set up a *calculus of data structures* date back to [51, 52, 53] where the  $\leq$ -ordering and associated rules are defined. The approach, however, was not agile enough. It is only after its foundations are stated in the pointfree style [54, 56] that succinct calculations can be performed to derive data representations.

*Summary.* We have thus far introduced the topic of data representation framed in two contexts, one practical (data mapping scenarios) and the other theoretical (data refinement). In the remainder of the paper the reader will be provided with strategies and tools for handling mapping scenarios by calculation. This is preceded by the section which follows, which settles basic notation conventions and provides a brief overview of the binary relational calculus and the pointfree-transform, which is essential to understanding data calculations to follow. Textbook [11] is recommended as further reading.

### 3 Introducing the Pointfree Transform

By *pointfree transform* [60] (“PF-transform” for short) we essentially mean the conversion of predicate logic formulæ into binary relations by removing bound variables and quantifiers — a technique which, initiated by De Morgan in the 1860s [61], eventually led to what is known today as the *algebra of programming* [5, 11]. As suggested in [60], the PF-transform offers to the predicate calculus what the Laplace transform [41] offers to the differential/integral calculus: the possibility of changing the underlying mathematical space in a way which enables agile algebraic calculation.

Theories “refactored” via the PF-transform become more general, more structured and simpler [58, 59, 60]. Elegant expressions replace lengthy formulæ and easy-to-follow calculations replace pointwise proofs with lots of “ $\dots$ ” notation, case analyses and natural language explanations for “obvious” steps.

The main principle of the PF-transform is that “*everything is a binary relation*” once logical expressions are PF-transformed; one thereafter resorts to the powerful calculus of binary relations [5, 11] until proofs are discharged or solutions are found for the original problem statements, which are mapped back to logics if required.

*Relations.* Let arrow  $B \xleftarrow{R} A$  denote a binary relation on datatypes  $A$  (source) and  $B$  (target). We will say that  $B \longleftarrow A$  is the *type* of  $R$  and write  $b R a$  to mean that pair  $(b, a)$  is in  $R$ . Type declarations  $B \xleftarrow{R} A$  and  $A \xrightarrow{R} B$  will mean the same.

$R \cup S$  (resp.  $R \cap S$ ) denotes the union (resp. intersection) of two relations  $R$  and  $S$ .  $\top$  is the largest relation of its type. Its dual is  $\perp$ , the smallest such relation (the empty one). Two other operators are central to the relational calculus: composition ( $R \cdot S$ ) and converse ( $R^\circ$ ). The latter has already been introduced in section 2. Composition is defined in the usual way:  $b(R \cdot S)c$  holds wherever there exists some mediating  $a$  such that  $bRa \wedge aSc$ . Thus we get one of the kernel rules of the PF-transform:

$$b(R \cdot S)c \equiv \langle \exists a :: bRa \wedge aSc \rangle \quad (11)$$

Note that converse is an involution

$$(R^\circ)^\circ = R \quad (12)$$

and commutes with composition:

$$(R \cdot S)^\circ = S^\circ \cdot R^\circ \quad (13)$$

All these relational operators are  $\subseteq$ -monotonic, where  $\subseteq$  is the inclusion partial order (6). Composition is the basis of (sequential) factorization. Everywhere  $T = R \cdot S$  holds, the replacement of  $T$  by  $R \cdot S$  will be referred to as a “factorization” and that of  $R \cdot S$  by  $T$  as “fusion”. Every relation  $B \xleftarrow{R} A$  allows for two trivial factorizations,  $R = R \cdot id_A$  and  $R = id_B \cdot R$  where, for every  $X$ ,  $id_X$  is the identity relation mapping every element of  $X$  onto itself. (As a rule, subscripts will be dropped wherever types are implicit or easy to infer.) Relational equality can be established by  $\subseteq$ -antisymmetry:

$$R = S \equiv R \subseteq S \wedge S \subseteq R \tag{14}$$

*Coreflexives and orders.* Some standard terminology arises from the *id* relation: a (endo) relation  $A \xleftarrow{R} A$  (often called an *order*) will be referred to as *reflexive* iff  $id \subseteq R$  holds and as *coreflexive* iff  $R \subseteq id$  holds. Coreflexive relations are fragments of the identity relation which model predicates or sets. They are denoted by uppercase Greek letters (eg.  $\Phi, \Psi$ ) and obey a number of interesting properties, among which we single out the following, which prove very useful in calculations:

$$\Phi \cdot \Psi = \Phi \cap \Psi = \Psi \cdot \Phi \tag{15}$$

$$\Phi^\circ = \Phi \tag{16}$$

The PF-transform of a (unary) *predicate*  $p$  is the coreflexive  $\Phi_p$  such that

$$b \Phi_p a \equiv (b = a) \wedge (p a)$$

that is, the relation that maps every  $a$  which satisfies  $p$  (and only such  $a$ ) onto itself. The PF-meaning of a set  $S$  is  $\Phi_{\lambda a.a \in S}$ , that is,  $b \Phi_S a$  means  $(b = a) \wedge a \in S$ .

Preorders are reflexive and transitive relations, where  $R$  is transitive iff  $R \cdot R \subseteq R$  holds. Partial orders are anti-symmetric preorders, where  $R$  being anti-symmetric means  $R \cap R^\circ \subseteq id$ . A preorder  $R$  is an *equivalence* if it is symmetric, that is, if  $R = R^\circ$ .

*Taxonomy.* Converse is of paramount importance in establishing a wider taxonomy of binary relations. Let us first define two important notions: the *kernel* of a relation  $R$ ,  $\ker R \stackrel{\text{def}}{=} R^\circ \cdot R$  and its dual,  $\text{img } R \stackrel{\text{def}}{=} R \cdot R^\circ$ , the *image* of  $R$  (12, 13). From (12, 13) one immediately draws

$$\ker (R^\circ) = \text{img } R \tag{17}$$

$$\text{img } (R^\circ) = \ker R \tag{18}$$

Kernel and image lead to the following terminology:

	<i>Reflexive</i>	<i>Coreflexive</i>	
$\ker R$	entire $R$	injective $R$	(19)
$\text{img } R$	surjective $R$	simple $R$	

<sup>4</sup> As explained later on, these operators are relational extensions of two concepts familiar from set theory: the image of a function  $f$ , which corresponds to the set of all  $y$  such that  $(\exists x :: y = f x)$ , and the kernel of  $f$ , which is the equivalence relation  $b(\ker f)a \equiv f b = f a$ . (See exercise 3).

In words: a relation  $R$  is said to be *entire* (or total) iff its kernel is reflexive and to be *simple* (or functional) iff its image is coreflexive. Dually,  $R$  is *surjective* iff  $R^\circ$  is entire, and  $R$  is *injective* iff  $R^\circ$  is simple.

Recall that part of this terminology has already been mentioned in section 2. In this context, let us check formula (1) against the definitions captured by (19) as warming-up exercise in pointfree-to-pointwise conversion:

$$\begin{aligned}
& \langle \forall c, a, a' :: c R a \wedge c R a' \Rightarrow a = a' \rangle \\
\equiv & \quad \{ \text{rules of quantification [5] and converse} \} \\
& \langle \forall a, a' : \langle \exists c :: a R^\circ c \wedge c R a' \rangle : a = a' \rangle \\
\equiv & \quad \{ \text{(11) and rules of quantification} \} \\
& \langle \forall a, a' :: a(R^\circ \cdot R)a' \Rightarrow a = a' \rangle \\
\equiv & \quad \{ \text{(6) and definition of kernel} \} \\
& \ker R \subseteq id
\end{aligned}$$

*Exercise 1.* Derive (2) from (19). □

*Exercise 2.* Resort to (17)(18) and (19) to prove the following four rules of thumb:

- Converse of *injective* is *simple* (and vice-versa)
- Converse of *entire* is *surjective* (and vice-versa)
- Smaller than injective (simple) is injective (simple)
- Larger than entire (surjective) is entire (surjective) □

A relation is said to be a *function* iff it is both simple and entire. Following a widespread convention, functions will be denoted by lowercase characters (eg.  $f, g, \phi$ ) or identifiers starting with lowercase characters. Function application will be denoted by juxtaposition, eg.  $f a$  instead of  $f(a)$ . Thus  $bfa$  means the same as  $b = f a$ .

The overall taxonomy of binary relations is pictured in figure 3 where, further to the standard classes, we add *representations* and *abstractions*. As seen already, these are the relation classes involved in  $\leq$ -rules (7). Because of  $\subseteq$ -antisymmetry,  $\text{img } F = id$  wherever  $F$  is an *abstraction* and  $\ker R = id$  wherever  $R$  is a *representation*.

Bijections (also referred to as isomorphisms) are functions, abstractions and representations at the same time. A particular bijection is  $id$ , which also is the smallest equivalence relation on a particular data domain. So,  $b id a$  means the same as  $b = a$ .

*Functions and relations.* The interplay between functions and relations is a rich part of the binary relation calculus [11]. For instance, the PF-transform rule which follows, involving two functions ( $f, g$ ) and an arbitrary relation  $R$

$$b(f^\circ \cdot R \cdot g)a \equiv (f b)R(g a) \quad (20)$$

plays a prominent role in the PF-transform [4]. The pointwise definition of the kernel of a function  $f$ , for example,

$$b(\ker f)a \equiv f b = f a \quad (21)$$

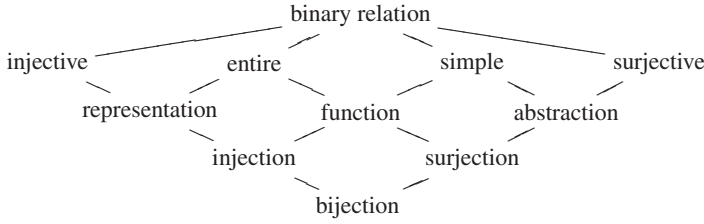


Fig. 3. Binary relation taxonomy

stems from (20), whereby it is easy to see that  $\top$  is the kernel of every constant function,  $1 \xleftarrow{!} A$  included. (Function  $!$  — read “!” as “bang” — is the unique function of its type, where 1 denotes the singleton data domain.)

Exercise 3. Given a function  $B \xleftarrow{f} A$ , calculate the pointwise version (21) of  $\ker f$  and show that  $\text{img } f$  is the coreflexive associated to predicate  $p \ b = \langle \exists a :: b = f \ a \rangle$ .  $\square$

Given two preorders  $\leq$  and  $\sqsubseteq$ , one may relate arguments and results of pairs of suitably typed functions  $f$  and  $g$  in a particular way,

$$f \circ \cdot \sqsubseteq = \leq \cdot g \tag{22}$$

in which case both  $f, g$  are monotone and said to be *Galois connected*. Function  $f$  (resp.  $g$ ) is referred to as the *lower* (resp. *upper*) adjoint of the connection. By introducing variables in both sides of (22) via (20), we obtain, for all  $a$  and  $b$

$$(f \ b) \sqsubseteq a \equiv b \leq (g \ a) \tag{23}$$

Quite often, the two adjoints are *sections* of binary operators. Given a binary operator  $\theta$ , its two sections  $(a\theta)$  and  $(\theta b)$  are unary functions  $f$  and  $g$  such that, respectively:

$$f = (a\theta) \equiv f \ b = a \ \theta \ b \tag{24}$$

$$g = (\theta b) \equiv g \ a = a \ \theta \ b \tag{25}$$

Galois connections in which the two preorders are relation inclusion ( $\leq, \sqsubseteq := \subseteq, \subseteq$ ) and whose adjoints are sections of relational combinators are particularly interesting because they express universal properties about such combinators. Table 1 lists connections which are relevant for this paper.

It is easy to recover known properties of the relation calculus from table 1. For instance, the entry marked “*shunting rule*” leads to

$$h \cdot R \subseteq S \equiv R \subseteq h^\circ \cdot S \tag{26}$$

for all  $h, R$  and  $S$ . By taking converses, one gets another entry in table 1, namely

$$R \cdot h^\circ \subseteq S \equiv R \subseteq S \cdot h \tag{27}$$

**Table 1.** Sample of Galois connections in the relational calculus. The general formula given on top is a logical equivalence universally quantified on  $S$  and  $R$ . It has a left part involving lower adjoint  $f$  and a right part involving upper adjoint  $g$ .

$(f R) \subseteq S \equiv R \subseteq (g S)$			
Description	$f$	$g$	Obs.
converse	$(-)^{\circ}$	$(-)^{\circ}$	
<i>shunting</i> rule	$(h \cdot)$	$(h^{\circ} \cdot)$	$h$ is a function
“converse” <i>shunting</i> rule	$(\cdot h^{\circ})$	$(\cdot h)$	$h$ is a function
domain	$\delta$	$(\top \cdot)$	left $\subseteq$ restricted to coreflexives
range	$\rho$	$(\cdot \top)$	left $\subseteq$ restricted to coreflexives
difference	$(- - R)$	$(R \cup )$	

These equivalences are popularly known as “shunting rules” [11]. The fact that *at most* and equality coincide in the case of functions

$$f \subseteq g \equiv f = g \equiv f \supseteq g \tag{28}$$

is among many beneficial consequences of these rules (see eg. [11]).

It should be mentioned that some rules in table 1 appear in the literature under different guises and usually not identified as GCs [5]. For a thorough presentation of the relational calculus in terms of GCs see [1, 5]. There are *many* advantages in such an approach: further to the systematic tabulation of operators (of which table 1 is just a sample), GCs have a rich algebra of properties, namely:

- Both adjoints  $f$  and  $g$  in a GC are monotonic;
- Lower adjoint  $f$  commutes with join and upper-adjoint  $g$  commutes with meet, wherever these exist;
- Two cancellation laws hold,  $b \leq g(f b)$  and  $f(g a) \sqsubseteq a$ , respectively known as *left-cancellation* and *right-cancellation*.

It may happen that a cancellation law holds up to equality, for instance  $f(g a) = a$ , in which case the connection is said to be *perfect* on the particular side [1].

*Simplicity.* Simple relations (that is, partial functions) will be particularly relevant in the sequel because of their ubiquity in software modeling. In particular, they will be used in this paper to model data *identity* and any kind of data structure “embodying a functional dependency” [58] such as eg. relational database tables, memory segments (both static and dynamic) and so on.

In the same way simple relations generalize functions (figure 3), *shunting* rules [26, 27] generalize to

$$S \cdot R \subseteq T \equiv (\delta S) \cdot R \subseteq S^{\circ} \cdot T \tag{29}$$

$$R \cdot S^{\circ} \subseteq T \equiv R \cdot \delta S \subseteq T \cdot S \tag{30}$$

<sup>5</sup> For instance, the *shunting* rule is called *cancellation law* in [70].

for  $S$  simple. These rules involve the *domain* operator ( $\delta$ ) whose GC, as mentioned in table 11 involves coreflexives on the lower side:

$$\delta R \subseteq \Phi \equiv R \subseteq \top \cdot \Phi \quad (31)$$

We will draw harpoon arrows  $B \xleftarrow{R} A$  or  $A \xrightarrow{R} B$  to indicate that  $R$  is simple. Later on we will need to describe simple relations at pointwise level. The notation we shall adopt for this purpose is borrowed from VDM [38], where it is known as *mapping comprehension*. This notation exploits the applicative nature of a simple relation  $S$  by writing  $b S a$  as  $a \in \text{dom } S \wedge b = S a$ , where  $\wedge$  should be understood non-strict on the right argument<sup>6</sup> and  $\text{dom } S$  is the set-theoretic version of coreflexive  $\delta S$  above, that is,

$$\delta S = \Phi_{\text{dom } S} \quad (32)$$

holds (cf. the isomorphism between sets and coreflexives). In this way, relation  $S$  itself can be written as  $\{a \mapsto S a \mid a \in \text{dom } S\}$  and projection  $f \cdot S \cdot g^\circ$  as

$$\{g a \mapsto f(S a) \mid a \in \text{dom } S\} \quad (33)$$

provided  $g$  is injective (thus ensuring simplicity).

*Exercise 4.* Show that the union of two simple relations  $M$  and  $N$  is simple *iff* the following condition holds:

$$M \cdot N^\circ \subseteq \text{id} \quad (34)$$

(Suggestion: resort to universal property  $(R \cup S) \subseteq X \equiv R \subseteq X \wedge S \subseteq X$ .) Furthermore show that (34) converts to pointwise notation as follows,

$$\langle \forall a :: a \in (\text{dom } M \cap \text{dom } N) \Rightarrow (M a) = (N a) \rangle$$

— a condition known as (map) *compatibility* in VDM terminology [25].  $\square$

*Exercise 5.* It will be useful to order relations with respect to how defined they are:

$$R \preceq S \equiv \delta R \subseteq \delta S \quad (35)$$

From  $\top = \text{ker}!$  draw another version of (35),  $R \preceq S \equiv ! \cdot R \subseteq ! \cdot S$ , and use it to derive

$$R \cdot f^\circ \preceq S \equiv R \preceq S \cdot f \quad (36)$$

$\square$

*Operator precedence.* In order to save parentheses in relational expressions, we define the following precedence ordering on the relational operators seen so far:

$$-^\circ > \{\delta, \rho\} > (\cdot) > \cap > \cup$$

Example:  $R \cdot \delta S^\circ \cap T \cup V$  abbreviates  $((R \cdot (\delta (S^\circ))) \cap T) \cup V$ .

<sup>6</sup> VDM embodies a logic of partial functions (LPF) which takes this into account [38].

*Summary.* The material of this section is adapted from similar sections in [59, 60], which introduce the reader to the essentials of the PF-transform. While the notation adopted is standard [11], the presentation of the associated calculus is enhanced via the use of Galois connections, a strategy inspired by two (still unpublished) textbooks [1, 5]. There is a slight difference, perhaps: by regarding the underlying mathematics as that of a *transform* to be used wherever a “hard” formula  $\psi$  needs to be reasoned about, the overall flavour is more practical and not that of a *fine art* only accessible to the initiated — an aspect of the recent evolution of the calculus already stressed in [40].

The table below provides a summary of the PF-transform rules given so far, where left-hand sides are logical formulæ ( $\psi$ ) and right-hand sides are the corresponding PF equivalents ( $\llbracket \psi \rrbracket$ ):

$\psi$	$\llbracket \psi \rrbracket$
$\langle \forall a, b :: b R a \Rightarrow b S a \rangle$	$R \subseteq S$
$\langle \forall a :: f a = g a \rangle$	$f \subseteq g$
$\langle \forall a :: a R a \rangle$	$id \subseteq R$
$\langle \exists a :: b R a \wedge a S c \rangle$	$b(R \cdot S)c$
$b R a \wedge b S a$	$b(R \cap S) a$
$b R a \vee b S a$	$b(R \cup S) a$
$(f b) R (g a)$	$b(f^\circ \cdot R \cdot g) a$
TRUE	$b \top a$
FALSE	$b \perp a$

(37)

*Exercise 6.* Prove that relational composition preserves *all* relational classes in the taxonomy of figure 3. □

## 4 Data Structures

One of the main difficulties in studying data structuring is the number of disparate (inc. graphic) notations, programming languages and paradigms one has to deal with. Which should one adopt? While graphical notations such as the UML [15] are gaining adepts everyday, it is difficult to be precise in such notations because their semantics are, as a rule, not formally defined.

Our approach will be rather minimalist: we will *map* such notations to the PF-notation whose rudiments have just been presented. By the word “map” we mean a light-weight approach in this paper: presenting a fully formal semantics for the data structuring facilities offered by any commercial language or notation would be more than one paper in itself.

The purpose of this section is two fold: on the one hand, to show how overwhelming data structuring notations can be even in the case of simple data models such as our family tree (running) example; on the other hand, to show how to circumvent such disparity by expressing the same models in PF-notation. Particular emphasis will be put on describing Entity-relationship diagrams [30]. Later on we will go as far as capturing recursive data models by least fixpoints over polynomial types. Once again we warn the

<sup>7</sup> To use the words of Kreyszig [41] in his appreciation of the Laplace transform.



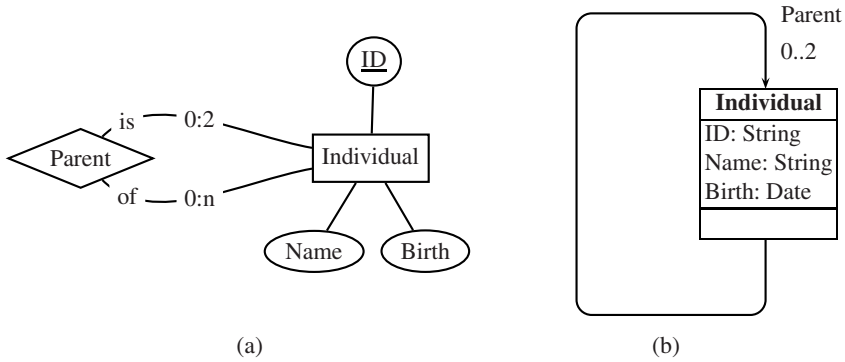


Fig. 4. ER and UML diagrams proposed for *genealogies*. Underlined identifiers denote keys.

reader that types and data modeling constructs in current programming languages are rather more complex than their obvious cousins in mathematics. For the sake of simplicity, we deliberately don't consider aspects such as non-strictness, lazy-evaluation, infinite data values [65] etc.

*Back to the running example.* Recall the family tree displayed in (9) and figure 2. Suppose requirements ask us to provide CRUD operations on a genealogy database collecting such family trees. How does one go about describing the data model underlying such operations?

The average database designer will approach the model via *entity-relationship* (ER) diagrams, for instance that of figure 4(a). But many others will regard this notation too old-fashioned and will propose something like the UML diagram of figure 4(b) instead.

Uncertain of what such drawings *actually mean*, many a programmer will prefer to go straight into code, eg. C

```
typedef struct Gen {
    char *name          /* name is a string */
    int  birth         /* birth year is a number */
    struct Gen *mother; /* genealogy of mother (if known) */
    struct Gen *father; /* genealogy of father (if known) */
};
```

— which matches with figure 2a — or XML, eg.

```
<!-- DTD for genealogical trees -->
<!ELEMENT tree (node+)>
<!ELEMENT node (name, birth, mother?, father?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT birth (#PCDATA)>
<!ELEMENT mother EMPTY>
<!ELEMENT father EMPTY>
<!ATTLIST tree
    ident ID #REQUIRED>
```

```
<!ATTLIST mother
    refid IDREF #REQUIRED>
<!ATTLIST father
    refid IDREF #REQUIRED>
```

— or plain SQL, eg. (fixing some arbitrary sizes for datatypes)

```
CREATE TABLE INDIVIDUAL (
    ID NUMBER (10) NOT NULL,
    Name VARCHAR (80) NOT NULL,
    Birth NUMBER (8) NOT NULL,
    CONSTRAINT INDIVIDUAL_pk PRIMARY KEY(ID)
);

CREATE TABLE ANCESTORS (
    ID VARCHAR (8) NOT NULL,
    Ancestor VARCHAR (8) NOT NULL,
    PID NUMBER (10) NOT NULL,
    CONSTRAINT ANCESTORS_pk PRIMARY KEY (ID,Ancestor)
);
```

— which matches with figure 2b.

What about functional programmers? By looking at pedigree tree (9) where we started from, an inductive data type can be defined, eg. in Haskell,

```
data PTree = Node {
    name    :: [ Char ],
    birth   :: Int    ,
    mother  :: Maybe PTree,
    father  :: Maybe PTree
}
(38)
```

whereby (9) would be encoded as data value

```
Node
{name = "Peter", birth = 1991,
 mother = Just (Node
  {name = "Mary", birth = 1956,
   mother = Nothing,
   father = Nothing}),
 father = Just (Node
  {name = "Joseph", birth = 1955,
   mother = Just (Node
    {name = "Margaret", birth = 1923,
     mother = Nothing, father = Nothing}),
   father = Just (Node
    {name = "Luigi", birth = 1920,
     mother = Nothing, father = Nothing})})})}
```

Of course, the same tree can still be encoded in XML notation eg. using DTD

```
<!-- DTD for genealogical trees -->
<!ELEMENT tree (name, birth, tree?, tree?)>
```

```
<!ELEMENT name (#PCDATA)>
<!ELEMENT birth (#PCDATA)>
```

As well-founded structures, these trees can be pretty-printed as in (9). However, how can one ensure that the same *print-family-tree* operation won't loop forever while retrieving data from eg. figure 2b? This would clearly happen if, by mistake, record 

1	Father	2
---	--------	---

 in figure 2b were updated to 

1	Father	5
---	--------	---

: *Peter* would become a descendant of himself!

Several questions suggest themselves: are all the above data models “equivalent”? If so, in what sense? If not, how can they be ranked in terms of “quality”? How can we tell apart the *essence* of a data model from its technology wrapping?

To answer these questions we need to put some effort in describing the notations involved in terms of a single, abstract (ie. technology free) unifying notation. But syntax alone is not enough: the ability to *reason* in such a notation is essential, otherwise different data models won't be comparable. Thus the reason why, in what follows, we choose the PF-notation as unifying framework<sup>8</sup>.

*Records are inhabitants of products.* Broadly speaking, a database is that part of an information system which collects *facts* or *records* of particular situations which are subject to retrieving and analytical processing. But, what is a record?

Any row in the tables of figure 2b is a record, ie. *records a fact*. For instance, record 

5	Peter	1991
---	-------	------

 tells: *Peter, whose ID number is 5, was born in 1991*. A mathematician would have written  $(5, Peter, 1991) \in \mathbb{N} \times String \times \mathbb{N}$  instead of *drawing* the tabular stuff and would have inferred  $(5, Peter, 1991) \in \mathbb{N} \times String \times \mathbb{N}$  from  $5 \in \mathbb{N}, Peter \in String$  and  $1991 \in \mathbb{N}$ , where, given two types  $A$  and  $B$ , their (Cartesian) product  $A \times B$  is the set  $\{(a, b) \mid a \in A \wedge b \in B\}$ . So records can be regarded as *tuples* which inhabit *products* of types.

Product datatype  $A \times B$  is essential to information processing and is available in virtually every programming language. In Haskell one writes  $(A, B)$  to denote  $A \times B$ , for  $A$  and  $B$  two given datatypes. This syntax can be decorated with names, eg.

```
data C = C { first :: A, second :: B }
```

as is the case of PTree (38). In the C programming language, the  $A \times B$  datatype is realized using “struct”s, eg.

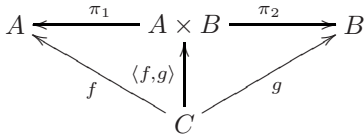
```
struct { A first; B second; };
```

The diagram below is suggestive of what product  $A \times B$  actually means, where  $f$  and  $g$  are functions, the two projections  $\pi_1, \pi_2$  are such that

$$\pi_1(a, b) = a \wedge \pi_2(a, b) = b \tag{39}$$

---

<sup>8</sup> The “everything is a relation” motto implicit in this approach is also the message of Alloy [36], a notation and associated model-checking tool which has been successful in *alloying* a number of disparate approaches to software modeling, namely model-orientation, object-orientation, etc. Quoting [36]: (...) “the Alloy language and its analysis are a Trojan horse: an attempt to capture the attention of software developers, who are mired in the tar pit of implementation technologies, and to bring them back to thinking deeply about underlying concepts”.



and function  $\langle f, g \rangle$  (read: “*f split g*”) is defined by  $\langle f, g \rangle c \stackrel{\text{def}}{=} (f\ c, g\ c)$ . The diagram expresses the two cancellation properties,  $\pi_1 \cdot \langle f, g \rangle = f$  and  $\pi_2 \cdot \langle f, g \rangle = g$ , which follow from a more general (universal) property,

$$k = \langle f, g \rangle \equiv \pi_1 \cdot k = f \wedge \pi_2 \cdot k = g \tag{40}$$

which holds for arbitrary (suitably typed) functions  $f, g$  and  $k$ . This tells that, given functions  $f$  and  $g$ , each producing inhabitants of types  $A$  and  $B$ , respectively, there is a unique function  $\langle f, g \rangle$  which combines  $f$  and  $g$  so as to produce inhabitants of product type  $A \times B$ . Read in another way: any function  $k$  delivering results into type  $A \times B$  can be uniquely decomposed into its two left and right components.

It can be easily checked that the definition of  $\langle f, g \rangle$  given above PF-transforms to  $\langle f, g \rangle = \pi_1^\circ \cdot f \cap \pi_2^\circ \cdot g$ . (Just re-introduce variables and simplify, thanks to (39), (20), etc.) This provides a hint on how to generalize the *split* combinator to relations  $\mathbb{R}$ :

$$\langle R, S \rangle = \pi_1^\circ \cdot R \cap \pi_2^\circ \cdot S \tag{41}$$

To feel the meaning of the extension we introduce variables in (41) and simplify:

$$\begin{aligned} \langle R, S \rangle &= \pi_1^\circ \cdot R \cap \pi_2^\circ \cdot S \\ &\equiv \{ \text{introduce variables; (37)} \} \\ (a, b)\langle R, S \rangle c &\equiv (a, b)(\pi_1^\circ \cdot R)c \wedge (a, b)(\pi_2^\circ \cdot S)c \\ &\equiv \{ \text{(20) twice} \} \\ (a, b)\langle R, S \rangle c &\equiv \pi_1(a, b)\ R\ c \wedge \pi_2(a, b)\ S\ c \\ &\equiv \{ \text{projections (39)} \} \\ (a, b)\langle R, S \rangle c &\equiv a\ R\ c \wedge b\ S\ c \end{aligned}$$

So, relational splits enable one to PF-transform logical formulæ involving more than two variables.

A special case of *split* will be referred to as *relational product*:

$$R \times S \stackrel{\text{def}}{=} \langle R \cdot \pi_1, S \cdot \pi_2 \rangle \tag{42}$$

So we can add two more entries to table (37):

$\psi$	$\llbracket \psi \rrbracket$
$a\ R\ c \wedge b\ S\ c$	$(a, b)\langle R, S \rangle c$
$b\ R\ a \wedge d\ S\ c$	$(b, d)(R \times S)(a, c)$

Finally note that binary product can be generalized to *n-ary product*  $A_1 \times A_2 \times \dots \times A_n$  involving projections  $\{\pi_i\}_{i=1, n}$  such that  $\pi_i(a_1, \dots, a_n) = a_i$ .

<sup>9</sup> Read more about this construct (which is also known as a *fork algebra* [28]) in section 7 and, in particular, in exercise 27.

Exercise 7. Identify which types are involved in the following bijections:

$$flatr(a, (b, c)) \stackrel{\text{def}}{=} (a, b, c) \tag{43}$$

$$flatl((b, c), d) \stackrel{\text{def}}{=} (b, c, d) \tag{44}$$

□

Exercise 8. Show that the side condition of the following *split-fusion law* <sup>10</sup>

$$\langle R, S \rangle \cdot T = \langle R \cdot T, S \cdot T \rangle \Leftarrow R \cdot (\text{img } T) \subseteq R \vee S \cdot (\text{img } T) \subseteq S \tag{45}$$

can be dispensed with in (at least) the following situations: (a)  $T$  is simple; (b)  $R$  or  $S$  are functions. □

Exercise 9. Write the following cancellation law with less symbols assuming that  $R \preceq S$  and  $S \preceq R$  <sup>(35)</sup> hold:

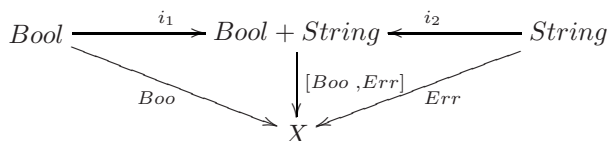
$$\pi_1 \cdot \langle R, S \rangle = R \cdot \delta S \wedge \pi_2 \cdot \langle R, S \rangle = S \cdot \delta R \tag{46}$$

□

*Data type sums.* The following is a declaration of a date type in Haskell which is inhabited by *either* Booleans or error strings:

```
data X = Boo Bool | Err String
```

If one queries a Haskell interpreter for the types of the `Boo` and `Err` constructors, one gets two functions which fit in the following diagram



where  $Bool + String$  denotes the sum (disjoint union) of types  $Bool$  and  $String$ , functions  $i_1, i_2$  are the necessary *injections* and  $[Boo, Err]$  is an instance of the “*either*” relational combinator :

$$[R, S] = (R \cdot i_1^{\circ}) \cup (S \cdot i_2^{\circ}) \quad \text{cf.} \quad \begin{array}{ccccc} A & \xrightarrow{i_1} & A + B & \xleftarrow{i_2} & B \\ & \searrow^R & \downarrow [R, S] & \swarrow_S & \\ & & C & & \end{array} \tag{47}$$

In pointwise notation,  $[R, S]$  means

$$c[R, S]x \equiv \langle \exists a :: c R a \wedge x = i_1 a \rangle \vee \langle \exists b :: c S a \wedge x = i_2 b \rangle$$

<sup>10</sup> Theorem 12.30 in [1].

In the same way *split* was used above to define relational product  $R \times S$ , *either* can be used to define *relational sums*:

$$R + S = [i_1 \cdot R, i_2 \cdot S] \tag{48}$$

As happens with products,  $A + B$  can be generalized to *n-ary sum*  $A_1 + A_2 + \dots + A_n$  involving  $n$  injections  $\{i_i\}_{i=1,n}$ .

In most programming languages, sums are not primitive and need to be programmed on purpose, eg. in C (using unions)

```

struct {
    int tag; /* eg. 1,2 */
    union {
        A ifA;
        B ifB;
    } data;
};

```

where explicit integer tags are introduced so as to model injections  $i_1, i_2$ .

*(Abstract) pointers.* A particular example of a datatype sum is  $1 + A$ , where  $A$  is an arbitrary type and  $1$  is the singleton type. The “amount of information” in this kind of structure is that of a pointer in C/C++: one “pulls a rope” and either gets nothing ( $1$ ) or something useful of type  $A$ . In such a programming context “nothing” above means a predefined value NIL. This analogy supports our preference in the sequel for NIL as canonical inhabitant of datatype  $1$ . In fact, we will refer to  $1 + A$  (or  $A + 1$ ) as the “pointer to  $A$ ” datatype [1]. This corresponds to the Maybe type constructor in Haskell.

*Polynomial types, grammars and languages.* Types involving arbitrary nesting of products and sums are called *polynomial* types, eg.  $1 + A \times B$  (the “pointer to struct” type). These types capture the abstract contents of generative grammars (expressed in extended BNF notation) once non-terminal symbols are identified with types and terminal symbols are filtered. The conversion is synthesized by the following table,

BNF NOTATION		POLYNOMIAL NOTATION
$\alpha \mid \beta$	$\mapsto$	$\alpha + \beta$
$\alpha\beta$	$\mapsto$	$\alpha \times \beta$
$\epsilon$	$\mapsto$	$1$
$a$	$\mapsto$	$1$

(49)

applicable to the right hand side of BNF-productions, where  $\alpha, \beta$  range over sequences of terminal or non-terminal symbols,  $\epsilon$  stands for *empty* and  $a$  ranges over terminal symbols. For instance, production  $X \rightarrow \epsilon \mid a A X$  (where  $X, A$  are non-terminals and  $a$  is terminal) leads to equation

$$X = 1 + A \times X \tag{50}$$

---

<sup>11</sup> Note that we are abstracting from the reference/dereference semantics of a *pointer* as understood in C-like programming languages. This is why we refer to  $1 + A$  as an *abstract* pointer. The explicit introduction of references (pointers, keys, identities) is deferred to section 9.

which has  $A^*$  — the “sequence of  $A$ ” datatype — as least solution. Since  $1 + A \times X$  can also be regarded as instance of the “pointer to struct” pattern, one can encode the same equation as the following (suitably sugared) type declaration in C:

```
typedef struct x {
    A data;
    struct x *next;
} Node;

typedef Node *X;
```

*Recursive types.* Both the interpretation of grammars [68] and the analysis of datatypes with pointers [69] lead to systems of polynomial equations, that is, to mutually recursive datatypes. For instance, the two *typedefs* above lead to  $Node = A \times X$  and to  $X = 1 + Node$ . It is the substitution of  $Node$  by  $A \times X$  in the second equation which gives rise to (50). There is a slight detail, though: in dealing with recursive types one needs to replace *equality* of types by *isomorphism* of types, a concept to be dealt with later on in section 5. So, for instance, the *PTree* datatype illustrated above in the XML and Haskell syntaxes is captured by the equation

$$PTree \cong Ind \times (PTree + 1) \times (PTree + 1) \quad (51)$$

where  $Ind = Name \times Birth$  packages the information relative to name and birth year, which don't participate in the recursive machinery and are, in a sense, parameters of the model. Thus one may write  $PTree \cong G(Ind, PTree)$ , in which  $G$  abstracts the particular pattern of recursion chosen to model family trees

$$G(X, Y) \stackrel{\text{def}}{=} X \times (Y + 1) \times (Y + 1)$$

where  $X$  refers to the parametric information and  $Y$  to the inductive part [12].

Let us now think of the operation which fetches a particular individual from a given *PTree*. From (51) one is intuitively led to an algorithm which *either* finds the individual (*Ind*) at the root of the tree, *or* tries and finds it in the left sub-tree (*PTree*) *or* tries and finds it in the right sub-tree (*PTree*). Why is this strategy “the natural” and obvious one? The answer to this question leads to the notion of datatype *membership* which is introduced below.

*Membership.* There is a close relationship between the *shape* of a data structure and the algorithms which fetch data from it. Put in other words: every instance of a given datatype is a kind of *data container* whose mathematical structure determines the particular *membership* tests upon which such algorithms are structured.

Sets are perhaps the best known data containers and purport a very intuitive notion of membership: everybody knows what  $a \in S$  means, wherever  $a$  is of type  $A$  and  $S$  of type  $\mathcal{P}A$  (read: “the powerset of  $A$ ”). Sentence  $a \in S$  already tells us that (set) membership has type  $A \xleftarrow{\in} \mathcal{P}A$ . Now, lists are also *container types*, the intuition

<sup>12</sup> Types such as *PTree*, which are structured around another datatype (cf.  $G$ ) which captures its structural “shape” are often referred to as *two-level types* in the literature [66].

being that  $a$  belongs (or occurs) in list  $l \in A^*$  iff it can be found in any of its positions. In this case, membership has type  $A \xleftarrow{\in} A^*$  (note the overloading of symbol  $\in$ ). But even product  $A \times A$  has membership too:  $a$  is a member of a pair  $(x, y)$  of type  $A \times A$  iff it can be found in either sides of that pair, that is  $a \in (x, y)$  means  $a = x \vee a = y$ . So it makes sense to define a *generic* notion of membership, able to fully explain the overloading of symbol  $\in$  above.

Datatype membership has been extensively studied [11, 32, 59]. Below we deal with polynomial type membership, which is what it required in this paper. A polynomial type expression may involve the composition, product, or sum of other polynomial types, plus the identity ( $\text{Id } X = X$ ) and constant types ( $F X = K$ , where  $K$  is any basic datatype, eg. the Booleans, the natural numbers, etc). Generic membership is defined, in the PF-style, over the structure of polynomial types as follows:

$$\in_K \stackrel{\text{def}}{=} \perp \tag{52}$$

$$\in_{\text{Id}} \stackrel{\text{def}}{=} \text{id} \tag{53}$$

$$\in_{F \times G} \stackrel{\text{def}}{=} (\in_F \cdot \pi_1) \cup (\in_G \cdot \pi_2) \tag{54}$$

$$\in_{F+G} \stackrel{\text{def}}{=} [\in_F, \in_G] \tag{55}$$

$$\in_{F.G} \stackrel{\text{def}}{=} \in_G \cdot \in_F \tag{56}$$

*Exercise 10.* Calculate the membership of type  $F X = X \times X$  and convert it to pointwise notation, so as to confirm the intuition above that  $a \in (x, y)$  holds iff  $a = x \vee a = y$ .  $\square$

Generic membership will be of help in specifying data structures which depend on each other by some form of *referential integrity* constraint. Before showing this, we need to introduce the important notion of *reference*, or *identity*.

*Identity.* Base clause (53) above clearly indicates that, sooner or later, equality plays its role when checking for polynomial membership. And equality of complex objects is cumbersome to express and expensive to calculate. Moreover, checking two objects for equality based on their properties alone may not work: it may happen that two physically different objects have the same properties, eg. two employees with exactly the same age, name, born in the same place, etc.

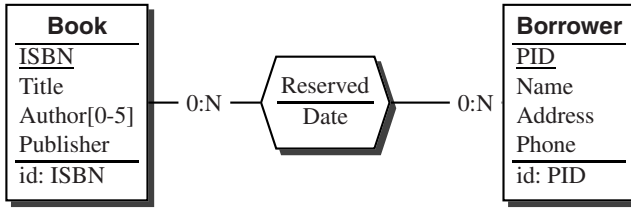
This *identification* problem has a standard solution: one associates to the objects in a particular collection *identifiers* which are unique in that particular context, cf. eg. identifier *ID* in figure 2b. So, instead of storing a collection of objects of (say) type  $A$  in a set of (say) type  $\mathcal{P}A$ , one stores an association of unique names to the original objects, usually thought of in tabular format — as is the case in figure 2b.

However, thinking in terms of *tabular relations* expressed by sets of tuples where particular attributes ensure unique identification<sup>13</sup>, as is typical of database theory [45], is neither sufficiently general nor agile enough for reasoning purposes. References [56, 58] show that relational *simplicity*<sup>14</sup> is what matters in unique identification. So

<sup>13</sup> These attributes are known as *keys*.

<sup>14</sup> Recall that a relation is simple wherever its image is coreflexive (19).





**Fig. 5.** Sample of GER diagram (adapted from [30]). Underlined identifiers denote keys.

it suffices to regard collections of uniquely identified objects  $A$  as simple relations of type

$$K \rightarrow A \quad (57)$$

where  $K$  is a nonempty datatype of *keys*, or identifiers. For the moment, no special requirements are put on  $K$ . Later on,  $K$  will be asked to provide for a countably infinite supply of identifiers, that is, to behave such as *natural number objects* do in category theory [47].

Below we show that simplicity and membership are what is required of our PF-notation to capture the semantics of data modeling (graphical) notations such as *Entity-Relationship* diagrams and UML class diagrams.

*Entity-relationship diagrams.* As the name tells, Entity-Relationship data modeling involves two basic concepts: *entities* and *relationships*. Entities correspond to *nouns* in natural language descriptions: they describe classes of objects which have identity and exhibit a number of properties or attributes. Relationships can be thought of as *verbs*: they record (the outcome of) actions which engage different entities.

A few notation variants and graphical conventions exist for these diagrams. For its flexibility, we stick to the *generic entity-relationship* (GER) proposal of [30]. Figure 5 depicts a GER diagram involving two entities: **Book** and **Borrower**. The latter possesses attributes *Name*, *Address*, *Phone* and identity *PID*. As anticipated above where discussing how to model object identity, the semantic model of **Borrower** is a simple relation of type  $T_{PID} \rightarrow T_{Name} \times T_{Address} \times T_{Phone}$ , where by  $T_a$  we mean the type where attribute  $a$  takes values from. For notation economy, we will drop the  $T...$  notation and refer to the type  $T_a$  of attribute  $a$  by mentioning  $a$  alone:

$$\text{Borrowers} \stackrel{\text{def}}{=} PID \rightarrow Name \times Address \times Phone$$

Entity **Book** has a multivalued attribute (*Author*) imposing at most 5 authors. The semantics of such attributes can be also captured by (nested) simple relations:

$$\text{Books} \stackrel{\text{def}}{=} ISBN \rightarrow Title \times (5 \rightarrow Author) \times Publisher \quad (58)$$

Note the use of number 5 to denote the initial segment of the natural numbers ( $\mathbb{N}$ ) up to 5, that is, set  $\{1, 2, \dots, 5\}$ .

Books can be reserved by borrowers and there is no limit to the number of books the latter can reserve. The outcome of a reservation at a particular date is captured by relationship *Reserved*. Simple relations also capture relationship formal semantics, this time involving the identities of the entities engaged. In this case:

$$Reserved \stackrel{\text{def}}{=} ISBN \times PID \rightarrow Date$$

Altogether, the diagram specifies datatype  $Db \stackrel{\text{def}}{=} Books \times Borrowers \times Reserved$  inhabited by triples of simple relations.

In summary, Entity-Relationship diagrams describe data models which are concisely captured by simple binary relations. But we are not done yet: the semantics of the problem include the fact that only *existing* books can be borrowed by *known* borrowers. So one needs to impose a semantic constraint (invariant) on datatype *Db* which, written pointwise, goes as follows

$$\phi(M, N, R) \stackrel{\text{def}}{=} \langle \forall i, p, d :: d R (i, p) \Rightarrow \langle \exists x :: x M i \rangle \wedge \langle \exists y :: y M p \rangle \rangle \quad (59)$$

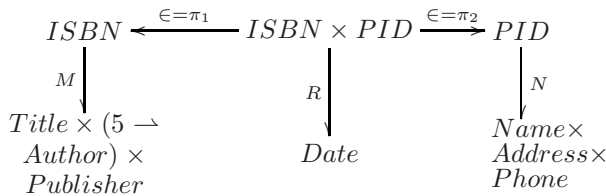
where  $i, p, d$  range over *ISBN*, *PID* and *Date*, respectively.

Constraints of this kind, which are implicitly assumed when interpreting *relationships* in these diagrams, are known as *integrity constraints*. Being invariants at the semantic level, they bring along with them the problem of ensuring their preservation by the corresponding CRUD operations. Worse than this, their definition in the predicate calculus is not agile enough for calculation purposes. Is there an alternative?

Space constraints preclude presenting the calculation which would show (59) *equivalent* to the following, much more concise PF-definition:

$$\phi(M, N, R) \stackrel{\text{def}}{=} R \cdot \in^\circ \preceq M \wedge R \cdot \in^\circ \preceq N \quad (60)$$

cf. diagram



To understand (60) and the diagram above, the reader must recall the definition of the  $\preceq$  ordering (35) — which compares the domains of two relations — and inspect the types of the two memberships,  $ISBN \xleftarrow{\in=\pi_1} ISBN \times PID$  in the first instance and  $PID \xleftarrow{\in=\pi_2} ISBN \times PID$  in the second. We check the first instance, the second being similar:

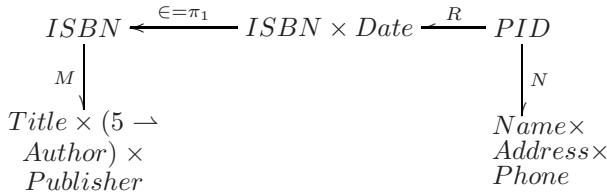
$$ISBN \xleftarrow{\in} ISBN \times PID$$

$$\begin{aligned}
 &= \{ \text{polynomial decomposition, membership of product (54)} \} \\
 &\quad (\in_{id} \cdot \pi_1) \cup (\in_{PID} \cdot \pi_2) \\
 &= \{ (52) \text{ and } (53) \} \\
 &\quad id \cdot \pi_1 \cup \perp \cdot \pi_2 \\
 &= \{ \text{trivia} \} \\
 &\quad \pi_1
 \end{aligned}$$

Multiplicity labels 0:N in the diagram of figure 5 indicate that there is no limit to the number of books borrowers can reserve. Now suppose the library decrees the following rule: *borrowers can have at most one reservation active*. In this case, label 0:N on the **Book** side must be restricted to 0:1. These so-called many-to-one relationships are once again captured by simple relations, this time of a different shape:

$$Reserved \stackrel{\text{def}}{=} PID \multimap ISBN \times Date \tag{61}$$

Altogether, note how clever use of simple relations dispenses with explicit cardinality invariants, which would put spurious weight on the data model. However, referential integrity is still to be maintained. The required pattern is once again nicely built up around membership,  $\phi(M, N, R) \stackrel{\text{def}}{=} (\in \cdot R)^\circ \preceq M \wedge R \preceq N$ , see diagram:



In retrospect, note the similarity in shape between these diagrams and the corresponding Entity-Relationship diagrams. The main advantage of the former resides in their richer semantics enabling formal reasoning, as we shall see in the sequel.

*Name spaces and “heaps”*. Relational database referential integrity can be shown to be an instance of a more general issue which traverses computing from end to end: *name space* referential integrity (NSRI). There are so many instances of NSRI that *genericity* is the only effective way to address the topic<sup>15</sup>. The issue is that, whatever programming language is adopted, one faces the same (ubiquitous) syntactic ingredients: (a) source code is made of units; (b) units refer to other units; (c) units need to be named.

For instance, a software package is a (named) collection of modules, each module being made of (named) collections of data type declarations, of variable declarations, of function declarations etc. Moreover, the package won’t compile in case name spaces don’t integrate with each other. Other examples of name spaces requiring NSRI are XML DTDs, grammars (where nonterminals play the role of names), etc.

<sup>15</sup> For further insight into *naming* see eg. Robin Milner’s interesting essay *What’s in a name? (in honour of Roger Needham)* available from <http://www.cl.cam.ac.uk/~rm135>.

In general, one is led to heterogeneous (typed) collections of (mutually dependent) *name spaces*, nicely modeled as simple relations again

$$N_i \rightarrow F_i(T_i, N_1, \dots, N_j, \dots, N_{n_i})$$

where  $F_i$  is a parametric type describing the particular pattern which expresses how names of type  $N_i$  depend on names of types  $N_j$  ( $j = 1, n_i$ ) and where  $T_i$  aggregates all types which don't participate in NSRI.

Assuming that all such  $F_i$  have membership, we can draw diagram

$$\begin{array}{ccc}
 N_i & \xrightarrow{S_i} & F_i(T_i, N_1, \dots, N_j, \dots, N_{n_i}) \\
 & \searrow_{\in_{i,j} \cdot S_i} & \downarrow_{\in_{i,j}} \\
 & & N_j
 \end{array}$$

where  $\in_{i,j} \cdot S_i$  is a name-to-name relation, or *dependence graph*. Overall NSRI will hold iff

$$\langle \forall i, j :: (\in_{i,j} \cdot S_i)^\circ \preceq S_j \rangle \tag{62}$$

which, once the definition order  $\preceq$  (35) is spelt out, converts to the pointwise:

$$\langle \forall n, m : n \in \text{dom } S_i : m \in_{i,j} (S_i n) \Rightarrow m \in \text{dom } S_j \rangle$$

Of course, (62) includes self referential integrity as a special case ( $i = j$ ).

NSRI also shows up at low level, where data structures such as *caches* and *heaps* can also be thought of as name spaces: at such a low level, names are *memory addresses*. For instance,  $\mathbb{N} \xrightarrow{H} F(T, \mathbb{N})$  models a heap “of shape”  $F$  where  $T$  is some datatype of interest and addresses are natural numbers ( $\mathbb{N}$ ). A heap satisfies NSRI iff it has no dangling pointers. We shall be back to this model of heaps when discussing how to deal with recursive data models (section 9).

*Summary.* This section addressed data-structuring from a double viewpoint: the one of programmers wishing to build data models in their chosen programming medium and the one of the software analyst wishing to bridge between models in different notations in order to eventually control data impedance mismatch. The latter entailed the abstraction of disparate data structuring notations into a common unifying one, that of binary relations and the PF-transform. This makes it possible to study data impedance mismatch from a formal perspective.

## 5 Data Impedance Mismatch Expressed in the PF-Style

Now that both the PF-notation has been presented and that its application to describing the semantics of data structures has been illustrated, we are better positioned to restate and study diagram (7). This expresses the *data impedance mismatch* between two data

models  $A$  and  $B$  as witnessed by a *connected* representation/abstraction pair  $(R, F)$ . Formally, this means that:

$$\left\{ \begin{array}{l} - R \text{ is a representation } (\ker R = id) \\ - F \text{ is an abstraction } (\text{img } F = id) \\ - R \text{ and } S \text{ are connected: } R \subseteq F^\circ \end{array} \right. \quad (63)$$

The higher the mismatch between  $A$  and  $B$  the more complex  $(R, F)$  are. The least impedance mismatch possible happens between a datatype and itself:

$$A \begin{array}{c} \xrightarrow{id} \\ \leq \\ \xleftarrow{id} \end{array} A \quad (64)$$

Another way to read (64) is to say that the  $\leq$ -ordering on data models is *reflexive*. It turns up that  $\leq$  is also *transitive*,

$$A \begin{array}{c} \xrightarrow{R} \\ \leq \\ \xleftarrow{F} \end{array} B \wedge B \begin{array}{c} \xrightarrow{S} \\ \leq \\ \xleftarrow{G} \end{array} C \Rightarrow A \begin{array}{c} \xrightarrow{S \cdot R} \\ \leq \\ \xleftarrow{F \cdot G} \end{array} C \quad (65)$$

that is, data impedances compose. The calculation of (65) is immediate: composition respects abstractions and representations (recall exercise 6) and  $(F \cdot G, S \cdot R)$  are connected:

$$\begin{aligned} & S \cdot R \subseteq (F \cdot G)^\circ \\ \equiv & \quad \{ \text{converses (13)} \} \\ & S \cdot R \subseteq G^\circ \cdot F^\circ \\ \Leftarrow & \quad \{ \text{monotonicity} \} \\ & S \subseteq G^\circ \wedge R \subseteq F^\circ \\ \equiv & \quad \{ \text{since } S, G \text{ and } R, F \text{ are assumed connected} \} \\ & \text{TRUE} \end{aligned}$$

*Right-invertibility.* A most beneficial consequence of (63) is the *right-invertibility* property

$$F \cdot R = id \quad (66)$$

which, written in predicate logic, expands to

$$\langle \forall a', a :: \langle \exists b :: a' F b \wedge b R a \rangle \equiv a' = a \rangle \quad (67)$$

The PF-calculation of (66) is not difficult:

$$F \cdot R = id$$

$$\begin{aligned}
 &\equiv \{ \text{equality of relations (14)} \} \\
 &\quad F \cdot R \subseteq id \wedge id \subseteq F \cdot R \\
 &\equiv \{ \text{img } F = id \text{ and } \ker R = id \text{ (63)} \} \\
 &\quad F \cdot R \subseteq F \cdot F^\circ \wedge R^\circ \cdot R \subseteq F \cdot R \\
 &\equiv \{ \text{converses} \} \\
 &\quad F \cdot R \subseteq F \cdot F^\circ \wedge R^\circ \cdot R \subseteq R^\circ \cdot F^\circ \\
 &\Leftarrow \{ (F \cdot) \text{ and } (R^\circ \cdot) \text{ are monotone} \} \\
 &\quad R \subseteq F^\circ \wedge R \subseteq F^\circ \\
 &\equiv \{ \text{trivia} \} \\
 &\quad R \subseteq F^\circ \\
 &\equiv \{ R \text{ and } F \text{ are connected (63)} \} \\
 &\text{TRUE}
 \end{aligned}$$

Clearly, this *right-invertibility* property matters in data representation:  $id \subseteq F \cdot R$  ensures the **no loss** principle and  $F \cdot R \subseteq id$  ensures the **no confusion** principle.

While (as we have just seen)  $F \cdot R = id$  is entailed by (63), the converse entailment *does not hold*:  $F \cdot R = id$  ensures  $R$  a representation and  $F$  surjective, but not simple. It may be also the case that  $F \cdot R = id$  and  $R \subseteq F^\circ$  does not hold, as the following counter-example shows:  $R = !^\circ$  and  $\perp \subseteq F \subseteq !$ .

*Exercise 11.* The reader may be interested to compare the calculation just above with the corresponding proof carried out at pointwise level using quantified logic expressions. This will amount to showing that (67) is entailed by the *pointwise* statement of  $(R, F)$  as a connected abstraction/representation pair.  $\square$

*Exercise 12.* Consider two data structuring patterns: “*pointer to struct*”  $(A \times B + 1)$  and “*pointer in struct*”  $((A + 1) \times B)$ . The question is: which of these data patterns represents the other? We suggest the reader checks the validity of

$$A \times B + 1 \begin{array}{c} \xrightarrow{R} \\ \leq \\ \xleftarrow{f} \end{array} (A + 1) \times B \quad (68)$$

where  $R \stackrel{\text{def}}{=} [i_1 \times id, \langle i_2, !^\circ \rangle]$  and  $f = R^\circ$ , that is,  $f$  satisfying clauses  $f(i_1 a, b) = i_1(a, b)$  and  $f(i_2 \text{ NIL}, b) = i_2 \text{ NIL}$ , where NIL denotes the unique inhabitant of type 1.  $\square$

Right-invertibility happens to be *equivalent* to (63) wherever both the abstraction and the representation are *functions*, say  $f, r$ :

$$A \begin{array}{c} \xrightarrow{r} \\ \leq \\ \xleftarrow{f} \end{array} C \quad \equiv \quad f \cdot r = id \quad (69)$$

Let us show that  $f \cdot r = id$  is equivalent to  $r \subseteq f^\circ$  and entails  $f$  surjective and  $r$  injective:

$$\begin{aligned}
 & f \cdot r = id \\
 \equiv & \quad \{ \text{(28)} \} \\
 & f \cdot r \subseteq id \\
 \equiv & \quad \{ \text{shunting (26)} \} \\
 & r \subseteq f^\circ \\
 \Rightarrow & \quad \{ \text{composition is monotonic} \} \\
 & f \cdot r \subseteq f \cdot f^\circ \wedge r^\circ \cdot r \subseteq r^\circ \cdot f^\circ \\
 \equiv & \quad \{ f \cdot r = id ; \text{converses} \} \\
 & id \subseteq f \cdot f^\circ \wedge r^\circ \cdot r \subseteq id \\
 \equiv & \quad \{ \text{definitions} \} \\
 & f \text{ surjective} \wedge r \text{ injective}
 \end{aligned}$$

The right invertibility property is a handy way of spotting  $\leq$  rules. For instance, the following cancellation properties of product and sum hold [11]:

$$\pi_1 \cdot \langle f, g \rangle = f, \pi_2 \cdot \langle f, g \rangle = g \tag{70}$$

$$[g, f] \cdot i_1 = g, [g, f] \cdot i_2 = f \tag{71}$$

Suitable instantiations of  $f, g$  to the identity function in both lines above lead to

$$\pi_1 \cdot \langle id, g \rangle = id, \pi_2 \cdot \langle f, id \rangle = id$$

$$[id, f] \cdot i_1 = id, [g, id] \cdot i_2 = id$$

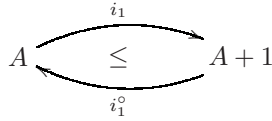
Thus we get — via (69) — the following  $\leq$ -rules

$$\begin{array}{ccc}
 \begin{array}{ccc}
 & \langle id, g \rangle & \\
 A & \xrightarrow{\leq} & A \times B \\
 & \pi_1 & \\
 \end{array} & \begin{array}{ccc}
 & \langle f, id \rangle & \\
 B & \xrightarrow{\leq} & A \times B \\
 & \pi_2 & \\
 \end{array} & (72)
 \end{array}$$

$$\begin{array}{ccc}
 \begin{array}{ccc}
 & i_1 & \\
 A & \xrightarrow{\leq} & A + B \\
 & [id, f] & \\
 \end{array} & \begin{array}{ccc}
 & i_2 & \\
 B & \xrightarrow{\leq} & A + B \\
 & [g, id] & \\
 \end{array} & (73)
 \end{array}$$

which tell the two projections surjective and the two injections injective (as expected). At programming level, they ensure that adding entries to a `struct` or (disjoint) `union` is a valid representation strategy, provided functions  $f, g$  are supplied by default [17]. Alternatively, they can be replaced by the top relation  $\top$  (meaning a *don't care*

representation strategy). In the case of (73), even  $\perp$  will work instead of  $f, g$ , leading, for  $A = 1$ , to the standard representation of datatype  $A$  by a “pointer to  $A$ ”:



Exercise 13. Show that  $[id, \perp] = i_1^\circ$  and that  $[\perp, id] = i_2^\circ$ . □

Isomorphic data types. As instance of (69) consider  $f$  and  $r$  such that both



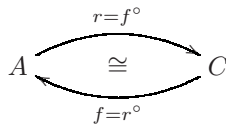
hold. This is equivalent to

$$\begin{aligned}
 & r \subseteq f^\circ \wedge f \subseteq r^\circ \\
 \equiv & \quad \{ \text{converses ; (14)} \} \\
 & r^\circ = f
 \end{aligned}
 \tag{74}$$

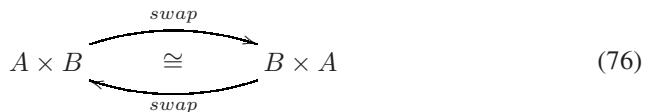
So  $r$  (a function) is the converse of another function  $f$ . This means that both are bijections (isomorphisms) — recall figure 3 — since

$$f \text{ is an isomorphism } \equiv f^\circ \text{ is a function}
 \tag{75}$$

In a diagram:



Isomorphism  $A \cong C$  corresponds to *minimal* impedance mismatch between types  $A$  and  $C$  in the sense that, although the format of data changes, data conversion in both ways is wholly recoverable. That is, two isomorphic types  $A$  and  $C$  are “abstractly” the same. Here is a trivial example



where *swap* is the name given to polymorphic function  $\langle \pi_2, \pi_1 \rangle$ . This isomorphism establishes the *commutativity* of  $\times$ , whose translation into practice is obvious: one can



change the order in which the entries in a `struct` (eg. in `C`) are listed; swap the order of two columns in a spreadsheet, etc.

The question arises: how can one be *certain* that *swap* is an isomorphism? A constructive, elegant way is to follow the advice of (75), which appeals to calculating the converse of *swap*,

$$\begin{aligned}
 & \text{swap}^\circ \\
 = & \{ (41) \} \\
 & (\pi_1^\circ \cdot \pi_2 \cap \pi_2^\circ \cdot \pi_1)^\circ \\
 = & \{ \text{converses} \} \\
 & \pi_2^\circ \cdot \pi_1 \cap \pi_1^\circ \cdot \pi_2 \\
 = & \{ (41) \text{ again} \} \\
 & \text{swap}
 \end{aligned}$$

which is *swap* again. So *swap* is its own converse and therefore an isomorphism.

*Exercise 14.* The calculation just above was too simple. To recognize the power of (75), prove the associative property of disjoint union,

$$\begin{array}{ccc}
 & r & \\
 & \curvearrowright & \\
 A + (B + C) & \cong & (A + B) + C \\
 & \curvearrowleft & \\
 & f = [id + i_1, i_2 \cdot i_2] & 
 \end{array} \tag{77}$$

by calculating the function *r* which is the converse of *f*.

Appreciate the elegance of this strategy when compared to what is conventional in discrete maths: to prove *f* bijective, one would have to either prove *f* injective and surjective, or *invent* its converse *f*<sup>◦</sup> and prove the two cancellations *f* · *f*<sup>◦</sup> = *id* and *f*<sup>◦</sup> · *f* = *id*. □

*Exercise 15.* The following are known isomorphisms involving sums and products:

$$A \times (B \times C) \cong (A \times B) \times C \tag{78}$$

$$A \cong A \times 1 \tag{79}$$

$$A \cong 1 \times A \tag{80}$$

$$A + B \cong B + A \tag{81}$$

$$C \times (A + B) \cong C \times A + C \times B \tag{82}$$

Guess the relevant isomorphism pairs. □

*Exercise 16.* Show that (75) holds, for *f* a function (of course). □

*Relation transposes.* Once again let us have a look at isomorphism pair (*r*, *f*) in (74), this time to introduce variables in the equality:

$$r^\circ = f$$

$$\begin{aligned} &\equiv \{ \text{introduce variables} \} \\ &\langle \forall a, c :: c (r^\circ) a \equiv c f a \rangle \\ &\equiv \{ \text{\textcircled{20}} \} \\ &\langle \forall a, c :: r c = a \equiv c = f a \rangle \end{aligned}$$

This is a pattern shared by many (pairs of) operators in the relational calculus, as is the case of eg. (omitting universal quantifiers)

$$k = \Lambda R \equiv R = \in \cdot k \tag{83}$$

where  $\Lambda$  converts a binary relation into *the corresponding* set-valued function [\[11\]](#), of

$$k = \text{tot } S \equiv S = \underbrace{i_1^\circ \cdot k}_{\text{untot } k} \tag{84}$$

where *tot* totalizes a simple relation  $S$  into *the corresponding* “Maybe-function” [\[16\]](#), and of

$$k = \text{curry } f \equiv f = \underbrace{\text{ap} \cdot (k \times \text{id})}_{\text{uncurry } k} \tag{85}$$

where *curry* converts a two-argument function  $f$  into *the corresponding* unary function, for  $\text{ap}(g, x) = g x$ .

These properties of  $\Lambda$ , *tot* and *curry* are normally referred to as *universal properties*, because of their particular pattern of universal quantification which ensures uniqueness [\[17\]](#). Novice readers will find them less cryptic once further (quantified) variables are introduced on their right hand sides:

$$\begin{aligned} k = \Lambda R &\equiv \langle \forall b, a :: b R a \equiv b \in (k a) \rangle \\ k = \text{tot } S &\equiv \langle \forall b, a :: b S a \equiv (i_1 b) = k a \rangle \\ k = \text{curry } f &\equiv \langle \forall b, a :: f(b, a) = (k b) a \rangle \end{aligned}$$

In summary,  $\Lambda$ , *tot* and *curry* are all isomorphisms. Here they are expressed by  $\cong$ -diagrams,

$$\begin{array}{ccc} (\mathcal{P}B)^A & \begin{array}{c} \xrightarrow{(\in \cdot)} \\ \cong \\ \xleftarrow{\Lambda} \end{array} & A \rightarrow B \\ & & \begin{array}{c} \xrightarrow{\text{untot}=(i_1^\circ \cdot)} \\ \cong \\ \xleftarrow{\text{tot}} \end{array} & (B + 1)^A & \xrightarrow{\cong} & A \rightarrow B \end{array} \tag{86}$$

$$\begin{array}{ccc} & & \begin{array}{c} \xrightarrow{\text{uncurry}} \\ \cong \\ \xleftarrow{\text{curry}} \end{array} \\ & & (B^A)^C & \xrightarrow{\cong} & B^{C \times A} \end{array}$$

where the exponential notation  $Y^X$  describes the datatype of all functions from  $X$  to  $Y$ .

<sup>16</sup> See [\[59\]](#). This corresponds to the view that simple relations are “possibly undefined” (ie. partial) functions. Also recall that  $A \xleftarrow{i_1^\circ} A + 1$  is the membership of *Maybe*.

<sup>17</sup> Consider, for instance, the right to left implication of [\(85\)](#): this tells that, given  $f$ , *curry*  $f$  is *the only* function satisfying  $f = \text{ap} \cdot (k \times \text{id})$ .

*Exercise 17.* (For Haskell programmers) Inspect the type of `flip lookup` and relate it to that of `tot`. (NB: `flip` is available from `GHC.Base` and `lookup` from `GHC.ListA`.)  $\square$

*Exercise 18.* The following is a well-known isomorphism involving exponentials:

$$(B \times C)^A \begin{array}{c} \xrightarrow{\langle (\pi_1 \cdot), (\pi_2 \cdot) \rangle} \\ \cong \\ \xleftarrow{\langle \_ \cdot, \_ \cdot \rangle} \end{array} B^A \times C^A \quad (87)$$

Write down the *universal property* captured by (87).  $\square$

*Exercise 19.* Relate function  $(p2p\ p)b \stackrel{\text{def}}{=} \text{if } b \text{ then } (\pi_1\ p) \text{ else } (\pi_2\ p)$  (read  $p2p$  as “pair to power”) with isomorphism

$$A \times A \cong A^2 \quad (88)$$

$\square$

Since exponentials are inhabited by functions and these are special cases of relations, there must be combinators which express functions in terms of relations and vice versa. Isomorphisms  $\lambda$  and  $\text{tot}$  (83, 84) already establish relationships of this kind. Let us see two more which will prove useful in calculations to follow.

“*Relational currying*”. Consider isomorphism

$$(C \rightarrow A)^B \begin{array}{c} \xrightarrow{(\_)\circ} \\ \cong \\ \xleftarrow{(\_)} \end{array} B \times C \rightarrow A \quad (89)$$

and associated universal property,

$$k = \overline{R} \equiv \langle \forall a, b, c :: a (k\ b)\ c \equiv a\ R\ (b, c) \rangle \quad (90)$$

where we suggest that  $\overline{R}$  be read “ $R$  transposed”.  $\overline{R}$  is thus a relation-valued function which expresses a kind of *selection/projection* mechanism: given some particular  $b_0$ ,  $\overline{R}\ b_0$  selects the “sub-relation” of  $R$  of all pairs  $(a, c)$  related to  $b_0$ .

This extension of *currying* to relations is a direct consequence of (83):

$$\begin{aligned} & B \times C \rightarrow A \\ \cong & \quad \{ \lambda / (\epsilon \cdot) \text{ (83, 86)} \} \\ & (\mathcal{P}A)^{B \times C} \\ \cong & \quad \{ \text{curry/uncurry} \} \\ & ((\mathcal{P}A)^C)^B \\ \cong & \quad \{ \text{exponentials preserve isomorphisms} \} \\ & (C \rightarrow A)^B \end{aligned}$$

The fact that, for simple relations, one could have resorted above to the *Maybe*-transpose (84) instead of the power transpose (83), leads to the conclusion that relational “currying” preserves simplicity:

$$(C \multimap A)^B \begin{array}{c} \xrightarrow{(\bar{\_})^\circ} \\ \cong \\ \xleftarrow{(\bar{\_})} \end{array} B \times C \multimap A \quad (91)$$

Since all relations are simple in (91), we can use notation convention (33) in the following pointwise definition of  $\overline{M}$  (for  $M$  simple):

$$\overline{M} b = \{c \mapsto M(b', c) \mid (b', c) \in \text{dom } M \wedge b' = b\} \quad (92)$$

This rule will play its role in multiple (foreign) key synthesis, see section 6

*Sets are fragments of “bang”.* We have already seen that sets can be modeled by coreflexive relations, which are simple. *Characteristic functions* are another way to represent sets:

$$2^A \begin{array}{c} \xrightarrow{\lambda p. \{a \in A \mid p a\}} \\ \cong \\ \xleftarrow{\lambda S. (\lambda a. a \in S)} \end{array} \mathcal{P}A \quad \text{cf.} \quad p = (\in S) \equiv S = \{a \mid p a\} \quad (93)$$

Here we see the correspondence between set comprehension and membership testing expressed by 2-valued functions, ie. predicates. By combining the *tot/untot* isomorphism (86) with (93) we obtain

$$\mathcal{P}A \begin{array}{c} \xrightarrow{s2m} \\ \cong \\ \xleftarrow{\text{dom}} \end{array} A \multimap 1 \quad (94)$$

where  $s2m S = ! \cdot \Phi_S$  and  $\text{dom}$  is defined by (32). This shows that every fragment of *bang* (!) models a set 18.

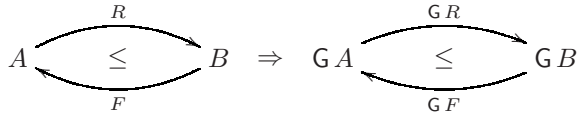
*Exercise 20.* Show that “obvious” facts such as  $S = \{a \mid a \in S\}$  and  $p x \equiv x \in \{a \mid p a\}$  stem from (93). Investigate other properties of set-comprehension which can be drawn from (93). □

*Relators and  $\leq$ -monotonicity.* A lesson learned from (69) is that right-invertible functions (surjections) have a  $\leq$ -rule of their own. For instance, predicate  $f n \stackrel{\text{def}}{=} n \neq 0$  over the integers is surjective (onto the Booleans). Thus Booleans can be represented by integers,  $2 \leq \mathbb{Z}$  — a fact C programmers know very well. Of course, one expects this “to scale up”: any data structure involving the Booleans (eg. trees of Booleans) can

<sup>18</sup> Relations at most *bang* (!) are referred to as *right-conditions* in [32].

be represented by a similar structure involving integers (eg. trees of integers). However, what does the word “similar” mean in this context? Typically, when building such a tree of integers, a C programmer looks at it and “sees” the tree with the same geometry where the integers have been replaced by their  $f$  images.

In general, let  $A$  and  $B$  be such that  $A \leq B$  and let  $G X$  denote a type parametric on  $X$ . We want to be able to *promote* the  $A$ -into- $B$  representation to structures of type  $G$  :



The questions arise: does this hold for *any* parametric type  $G$  we can think of? and what do relations  $G R$  and  $G F$  actually mean? Let us check. First of all, we investigate conditions for  $(G F, G R)$  to be connected to each other:

$$\begin{aligned}
 & G R \subseteq (G F)^\circ \\
 \Leftarrow & \quad \{ \text{assume } G(X^\circ) \subseteq (G X)^\circ, \text{ for all } X \} \\
 & G R \subseteq G(F^\circ) \\
 \Leftarrow & \quad \{ \text{assume monotonicity of } G \} \\
 & R \subseteq F^\circ \\
 \equiv & \quad \{ R \text{ is assumed connected to } F \} \\
 & \text{TRUE}
 \end{aligned}$$

Next,  $G R$  must be injective:

$$\begin{aligned}
 & (G R)^\circ \cdot G R \subseteq id \\
 \Leftarrow & \quad \{ \text{assume } (G X)^\circ \subseteq G(X^\circ) \} \\
 & (G R^\circ) \cdot G R \subseteq id \\
 \Leftarrow & \quad \{ \text{assume } (G R) \cdot (G T) \subseteq G(R \cdot T) \} \\
 & G(R^\circ \cdot R) \subseteq id \\
 \Leftarrow & \quad \{ \text{assume } G id \subseteq id \text{ and monotonicity of } G \} \\
 & R^\circ \cdot R \subseteq id \\
 \equiv & \quad \{ R \text{ is injective} \} \\
 & \text{TRUE}
 \end{aligned}$$

The reader eager to pursue checking the other requirements ( $R$  entire,  $F$  surjective, etc) will find out that the wish list concerning  $G$  will end up being as follows:

$$G id = id \tag{95}$$

$$G(R \cdot S) = (G R) \cdot (G S) \tag{96}$$

$$G(R^\circ) = (GR)^\circ \tag{97}$$

$$R \subseteq S \Rightarrow GR \subseteq GS \tag{98}$$

These turn up to be the properties of a *relator* [6], a concept which extends that of a *functor* to relations: a parametric datatype  $G$  is said to be a relator wherever, given a relation  $R$  from  $A$  to  $B$ ,  $GR$  extends  $R$  to  $G$ -structures. In other words, it is a relation from  $GA$  to  $GB$ , cf.

$$\begin{array}{ccc}
 A & \dots\dots\dots & GA \\
 R \downarrow & & \downarrow GR \\
 B & \dots\dots\dots & GB
 \end{array} \tag{99}$$

which obeys the properties above (it commutes with the identity, with composition and with converse, and it is monotonic). Once  $R, S$  above are restricted to functions, the behaviour of  $G$  in (95, 96) is that of a functor, and (97) and (98) become trivial — the former establishing that  $G$  preserves isomorphisms and the latter that  $G$  preserves equality (Leibniz).

It is easy to show that relators preserve all basic properties of relations as in figure 3. Two trivial relators are the *identity* relator  $ld$ , which is such that  $ld R = R$  and the *constant* relator  $K$  (for a given data type  $K$ ) which is such that  $K R = id_K$ . Relators can also be multi-parametric and we have already seen two of these: product  $R \times S$  (42) and sum  $R + S$  (48).

The prominence of parametric type  $G X = K \multimap X$ , for  $K$  a given datatype  $K$  of keys, leads us to the investigation of its properties as a relator,

$$\begin{array}{ccc}
 B & \dots\dots\dots & K \multimap B \\
 R \downarrow & & \downarrow K \multimap R \\
 C & \dots\dots\dots & K \multimap C
 \end{array}$$

where we define relation  $K \multimap R$  as follows:

$$N(K \multimap R)M \stackrel{\text{def}}{=} \delta M = \delta N \wedge N \cdot M^\circ \subseteq R \tag{100}$$

So, wherever simple  $N$  and  $M$  are  $(K \multimap R)$ -related, they are equally defined and their outputs are  $R$ -related. Wherever  $R$  is a function  $f$ ,  $K \multimap f$  is a function too defined by projection

$$(K \multimap f)M = f \cdot M \tag{101}$$

This can be extended to a bi-relator,

$$(g \multimap f)M = f \cdot M \cdot g^\circ \tag{102}$$

provided  $g$  is injective — recall (33).

*Exercise 21.* Show that instantiation  $R := f$  in (100) leads to  $N \subseteq f \cdot M$  and  $f \cdot M \subseteq N$  in the body of (100), and therefore to (101). □

*Exercise 22.* Show that  $(K \multimap \_)$  is a relator. □

*Indirection and dereferencing.* Indirection is a representation technique whereby data of interest stored in some data structure are replaced by references (pointers) to some global (dynamic) store — recall (57) — where the data are *actually* kept. The representation implicit in this technique involves allocating fresh cells in the global store; the abstraction consists in retrieving data by pointer dereferencing.

The motivation for this kind of representation is well-known: the referent is more expensive to move around than the reference. Despite being well understood and very widely used, dereferencing is a permanent source of errors in programming: it is impossible to retrieve data from a non-allocated reference.

To see how this strategy arises, consider  $B$  in (99) the datatype of interest (archived in some parametric container of type  $G$ , eg. binary trees of  $B$ s). Let  $A$  be the natural numbers and  $S$  be simple. Since relators preserve simplicity,  $G S$  will be simple too, as depicted aside. The meaning of this diagram is that of declaring a generic function (say  $rmap$ ) which, giving  $S$  simple, yields  $G S$  also simple. So  $rmap$  has type

$$\begin{array}{ccc} IN & \dots\dots\dots & G\ IN \\ S \downarrow & & \downarrow G\ S \\ B & \dots\dots\dots & G\ B \end{array}$$

$$(IN \rightarrow B) \rightarrow (G\ IN \rightarrow G\ B) \tag{103}$$

in the same way the  $fmap$  function of Haskell class `Functor` has type

$$fmap :: (a \rightarrow b) \rightarrow (g\ a \rightarrow g\ b)$$

(Recall that, once restricted to functions, relators coincide with functors.)

From (91) we infer that  $rmap$  can be “uncurried” into a simple relation of type  $((IN \rightarrow B) \times G\ IN) \rightarrow G\ B$  which is surjective, for finite structures. Of course we can replace  $IN$  above by any data domain, say  $K$  (suggestive of *key*), with the same cardinality, that is, such that  $K \cong IN$ . Then

$$\begin{array}{ccc} & \xrightarrow{R} & \\ G\ B & \leq & (K \rightarrow B) \times G\ K \\ & \xleftarrow{Dref} & \end{array} \tag{104}$$

holds for abstraction relation  $Dref$  such that  $\overline{Dref} = rmap$ , that is, such that (recalling (90))

$$y\ Dref\ (S, x) \equiv y(G\ S)x$$

for  $S$  a *store* and  $x$  a data structure of pointers (inhabitant of  $G\ K$ ).

Consider as example the indirect representation of finite lists of  $B$ s, in which fact  $l'\ Dref\ (S, l)$  instantiates to  $l'(S^*)l$ , itself meaning

$$\begin{aligned} l'(S^*)l &\equiv length\ l' = length\ l \wedge \\ &\quad (\forall i : 1 \leq i \leq length\ l : l\ i \in dom\ S \wedge (l'\ i) = S(l\ i)) \end{aligned}$$

So, wherever  $l'\ S^*l$  holds, no reference  $k$  in list  $l$  can live outside the domain of store  $S$ ,

$$k \in l \Rightarrow \langle \exists b :: b\ S\ k \rangle \tag{105}$$

where  $\in$  denotes finite list membership.

*Exercise 23.* Check that (105) PF-transforms to  $(\in \cdot \underline{l})^\circ \preceq S$ , an instance of NSRI (62) where  $\underline{l}$  denotes the “everywhere  $l$ ” constant function.  $\square$

*Exercise 24.* Define a representation function  $r \subseteq Dref^\circ$  (104) for  $G X = X^*$ .  $\square$

*Summary.* This section presented the essence of this paper’s approach to data calculation: a preorder ( $\leq$ ) on data types which formalizes data impedance mismatch in terms of representation/abstraction pairs. This preorder is compatible with the data type constructors introduced in section 4 and leads to a data structuring calculus whose laws enable systematic calculation of data implementations from abstract models. This is shown in the sections which follow.

## 6 Calculating Database Schemes from Abstract Models

Relational schema modeling is central to the “open-ended list of mapping issues” identified in [42]. In this section we develop a number of  $\leq$ -rules intended for cross-cutting impedance mismatch with respect to relational modeling. In other words, we intend to provide a practical method for inferring the schema of a database which (correctly) implements a given abstract model, including the stepwise synthesis of the associated abstraction and representation data mappings and concrete invariants. This method will be shown to extend to recursive structures in section 9.

*Relational schemes “relationally”.* Broadly speaking, a relational database is a  $n$ -tuple of tables, where each table is a relation involving value-level tuples. The latter are vectors of values which inhabit “atomic” data types, that is, which hold data with no further structure. Since many such relations (tables) exhibit *keys*, they can be thought of as *simple relations*. In this context, let

$$RDBT \stackrel{\text{def}}{=} \prod_{i=1}^n \left( \prod_{j=1}^{n_i} K_j \rightarrow \prod_{k=1}^{m_i} D_k \right) \tag{106}$$

denote the *generic type* of a relational database [2]. Every *RDBT*-compliant tuple  $db$  is a collection of  $n$  relational tables (index  $i = 1, n$ ) each of which is a mapping from a tuple of *keys* (index  $j$ ) to a tuple of *data of interest* (index  $k$ ). Wherever  $m_i = 0$  we have  $\prod_{k=1}^0 D_k \cong 1$ , meaning — via (94) — a *finite set* of tuples of type  $\prod_{j=1}^{n_i} K_j$ . (These are called *relationships* in the standard terminology.) Wherever  $n_i = 1$  we are in presence of a singleton relational table. Last but not least, all  $K_j$  and  $D_k$  are “atomic” types, otherwise  $db$  would fail first normal form (1NF) compliance [45].

Compared to what we have seen so far, type *RDBT* (106) is “flat”: there are no sums, no exponentials, no room for a single recursive datatype. Thus the mismatch identified in [42]: how does one map structured data (eg. encoded in XML) or a text generated according to some grammar, or even a collection of object types, into *RDBT*?

We devote the remainder of this section to a number of  $\leq$ -rules which can be used to transform arbitrary data models into instances of “flat” *RDBT*. Such rules share the generic pattern  $A \leq B$  (of which  $A \cong B$  is a special case) where  $B$  only contains products and simple relations. So, by successive application of such rules, one is lead



— eventually — to an instance of *RDBT*. Note that (89) and (94) are already rules of this kind (from left to right), the latter enabling one to get rid of powersets and the other of (some forms of) exponentials. Below we present a few more rules of this kind.

*Getting rid of sums.* It can be shown (see eg. [11]) that the *either* combinator  $[R, S]$  as defined by (47) is an isomorphism. This happens because one can always (uniquely) project a relation  $(B + C) \xrightarrow{T} A$  into two components  $B \xrightarrow{R} A$  and  $C \xrightarrow{S} A$ , such that  $T = [R, S]$ . Thus we have

$$(B + C) \rightarrow A \begin{array}{c} \xrightarrow{[-, \cdot]^\circ} \\ \cong \\ \xleftarrow{[-, \cdot]} \end{array} (B \rightarrow A) \times (C \rightarrow A) \quad (107)$$

which establishes universal property

$$T = [R, S] \equiv T \cdot i_1 = R \wedge T \cdot i_2 = S \quad (108)$$

When applied from left to right, rule (107) can be of help in removing sums from data models: relations whose input types involve sums can always be decomposed into pairs of relations whose types don't involve (such) sums.

Sums are a main ingredient in describing the *abstract syntax* of data. For instance, in the grammar approach to data modeling, alternative branches of a production in extended BNF notation map to polynomial sums, recall (49). The application of rule (107) removes such sums with no loss of information (it is an isomorphism), thus reducing the mismatch between abstract syntax and relational database models.

The calculation of (107), which is easily performed via the power-transpose [11], can alternatively be performed via the *Maybe*-transpose [59] — in the case of simple relations — meaning that relational *either* preserves simplicity:

$$(B + C) \rightarrow A \begin{array}{c} \xrightarrow{[-, \cdot]^\circ} \\ \cong \\ \xleftarrow{[-, \cdot]} \end{array} (B \rightarrow A) \times (C \rightarrow A) \quad (109)$$

What about the other (very common) circumstance in which sums occur at the output rather than at the input type of a relation? Another sum-elimination rule is applicable to such situations,

$$A \rightarrow (B + C) \begin{array}{c} \xrightarrow{\Delta_+} \\ \cong \\ \xleftarrow{\Downarrow} \end{array} (A \rightarrow B) \times (A \rightarrow C) \quad (110)$$

where

$$M \Downarrow N \stackrel{\text{def}}{=} i_1 \cdot M \cup i_2 \cdot N \quad (111)$$

$$\Delta_+ M \stackrel{\text{def}}{=} (i_1^\circ \cdot M, i_2^\circ \cdot M) \quad (112)$$

However, (110) does not hold as it stands for simple relations, because  $\overset{\dagger}{\bowtie}$  does not preserve simplicity: the union of two simple relations is not always simple. The weakest pre-condition for simplicity to be maintained is calculated as follows:

$$\begin{aligned}
 & M \overset{\dagger}{\bowtie} N \text{ is simple} \\
 \equiv & \quad \{ \text{definition (111)} \} \\
 & (i_1 \cdot M \cup i_2 \cdot N) \text{ is simple} \\
 \equiv & \quad \{ \text{simplicity of union of simple relations (34)} \} \\
 & (i_1 \cdot M) \cdot (i_2 \cdot N)^\circ \subseteq id \\
 \equiv & \quad \{ \text{converses ; shunting (26 27)} \} \\
 & M \cdot N^\circ \subseteq i_1^\circ \cdot i_2 \\
 \equiv & \quad \{ i_1^\circ \cdot i_2 = \perp ; (29 30) \} \\
 & \delta M \cdot \delta N \subseteq \perp \\
 \equiv & \quad \{ \text{coreflexives (15)} \} \\
 & \delta M \cap \delta N = \perp
 \end{aligned} \tag{113}$$

Thus,  $M \overset{\dagger}{\bowtie} N$  is simple iff  $M$  and  $N$  are domain-disjoint.

*Exercise 25.* Show that  $\overset{\dagger}{\bowtie} \cdot \Delta_+ = id$  holds. (NB: property  $id + id = id$  can be of help in the calculation.)  $\square$

*Exercise 26.* Do better than in exercise 25 and show that  $\overset{\dagger}{\bowtie}$  is the converse of  $\Delta_+$ , of course finding inspiration in (75). Universal property (108) will soften calculations if meanwhile you show that  $(M \overset{\dagger}{\bowtie} N)^\circ = [M^\circ, N^\circ]$  holds.  $\square$

*Getting rid of multivalued types.* Recall the *Books* type (58) defined earlier on. It deviates from *RDBT* in the second factor of its range type,  $5 \multimap Author$ , whereby book entries are bound to record up to 5 authors. How do we cope with this situation? *Books* is an instance of the generic relational type  $A \multimap (D \times (B \multimap C))$  for arbitrary  $A, B, C$  and  $D$ , where entry  $B \multimap C$  generalizes the notion of a multivalued attribute. Our aim in the calculations which follow is to split this relation type in two, so as to combine the two keys of types  $A$  and  $B$ :

$$\begin{aligned}
 & A \multimap (D \times (B \multimap C)) \\
 \cong & \quad \{ \text{Maybe transpose (86)} \} \\
 & (D \times (B \multimap C) + 1)^A \\
 \leq & \quad \{ (68) \} \\
 & ((D + 1) \times (B \multimap C))^A \\
 \cong & \quad \{ \text{splitting (87)} \}
 \end{aligned}$$

$$\begin{aligned}
 & (D + 1)^A \times (B \rightarrow C)^A \\
 \cong & \quad \{ \text{Maybe transpose } \text{\textcircled{86}} \text{\textcircled{89}} \} \\
 & (A \rightarrow D) \times (A \times B \rightarrow C)
 \end{aligned}$$

Altogether, we can rely on  $\leq$ -rule

$$A \rightarrow (D \times (B \rightarrow C)) \begin{array}{c} \xrightarrow{\Delta_n} \\ \leq \\ \xleftarrow{\text{\textcircled{86}}} \end{array} (A \rightarrow D) \times (A \times B \rightarrow C) \quad (114)$$

where the “nested join” operator  $\text{\textcircled{86}}$  is defined by

$$M \text{\textcircled{86}} N = \langle M, \overline{N} \rangle \quad (115)$$

— recall [\(91\)](#) — and  $\Delta_n$  is

$$\Delta_n M = (\pi_1 \cdot M, \text{usc}(\pi_2 \cdot M)) \quad (116)$$

where *usc* (=“undo simple currying”) is defined in comprehension notation as follows,

$$\text{usc } M \stackrel{\text{def}}{=} \{ (a, b) \mapsto (M a) b \mid a \in \text{dom } M, b \in \text{dom}(M a) \} \quad (117)$$

since  $M$  is simple. (Details about the calculation of this abstraction / representation pair can be found in [\[63\]](#).)

*Example.* Let us see the application of  $\leq$ -rule [\(114\)](#) to the *Books* data model [\(58\)](#). We document each step by pointing out the involved abstraction/representation pair:

$$\begin{aligned}
 \text{Books} &= \text{ISBN} \rightarrow (\text{Title} \times (5 \rightarrow \text{Author}) \times \text{Publisher}) \\
 &\cong_1 \quad \{ r_1 = \text{id} \rightarrow \langle \langle \pi_1, \pi_3 \rangle, \pi_2 \rangle, f_1 = \text{id} \rightarrow \langle \pi_1 \cdot \pi_1, \pi_2, \pi_2 \cdot \pi_1 \rangle \} \\
 &\quad \text{ISBN} \rightarrow (\text{Title} \times \text{Publisher}) \times (5 \rightarrow \text{Author}) \\
 &\leq_2 \quad \{ r_2 = \Delta_n, f_2 = \text{\textcircled{86}}, \text{cf. } \text{\textcircled{114}} \} \\
 &\quad (\text{ISBN} \rightarrow \text{Title} \times \text{Publisher}) \times (\text{ISBN} \times 5 \rightarrow \text{Author}) \\
 &= \text{Books}_2
 \end{aligned}$$

Since  $\text{Books}_2$  belongs to the *RDBT* class of types (assuming *ISBN*, *Title*, *Publisher* and *Author* atomic) it is directly implementable as a relational database schema.

Altogether, we have been able to calculate a *type-level* mapping between a source data model (*Books*) and a target data model ( $\text{Books}_2$ ). To carry on with the *mapping scenario* set up in [\[42\]](#), we need to be able to synthesize the two data maps (“map forward” and “map backward”) between *Books* and  $\text{Books}_2$ . We do this below as an exercise of PF-reasoning followed by pointwise translation.

Following rule [\(65\)](#), which enables composition of representations and abstractions, we synthesize  $r = \Delta_n \cdot (\text{id} \rightarrow \langle \langle \pi_1, \pi_3 \rangle, \pi_2 \rangle)$  as overall “map forward” representation,

and  $f = (id \rightarrow \langle \pi_1 \cdot \pi_1, \pi_2, \pi_2 \cdot \pi_1 \rangle) \bowtie_n$  as overall “map backward” abstraction. Let us transcribe  $r$  to pointwise notation:

$$\begin{aligned}
 r M &= \Delta_n((id \rightarrow \langle \pi_1, \pi_3, \pi_2 \rangle)M) \\
 &= \{ \text{(I02)} \} \\
 &\quad \Delta_n(\langle \pi_1, \pi_3, \pi_2 \rangle \cdot M) \\
 &= \{ \text{(I16)} \} \\
 &\quad (\pi_1 \cdot \langle \pi_1, \pi_3, \pi_2 \rangle \cdot M, usc(\pi_2 \cdot \langle \pi_1, \pi_3, \pi_2 \rangle \cdot M)) \\
 &= \{ \text{exercise 8; projections} \} \\
 &\quad (\langle \pi_1, \pi_3 \rangle \cdot M, usc(\pi_2 \cdot M))
 \end{aligned}$$

Thanks to (33), the first component in this pair transforms to pointwise

$$\{ isbn \mapsto (\pi_1(M isbn), \pi_3(M isbn)) \mid isbn \in dom M \}$$

and the second to

$$\{(isbn, a) \mapsto ((\pi_2 \cdot M) isbn)a \mid isbn \in dom M, a \in dom((\pi_2 \cdot M) isbn)\}$$

using definition (I17).

The same kind of reasoning will lead us to overall abstraction (“map backward”)  $f$ :

$$\begin{aligned}
 f(M, N) &= (id \rightarrow \langle \pi_1 \cdot \pi_1, \pi_2, \pi_2 \cdot \pi_1 \rangle)(M \bowtie_n N) \\
 &= \{ \text{(I02) and (I15)} \} \\
 &\quad \langle \pi_1 \cdot \pi_1, \pi_2, \pi_2 \cdot \pi_1 \rangle \cdot \langle M, \overline{N} \rangle \\
 &= \{ \text{exercise 8; projections} \} \\
 &\quad \langle \pi_1 \cdot \pi_1 \cdot \langle M, \overline{N} \rangle, \pi_2 \cdot \langle M, \overline{N} \rangle, \pi_2 \cdot \pi_1 \cdot \langle M, \overline{N} \rangle \rangle \\
 &= \{ \text{exercise 9; } \overline{N} \text{ is a function} \} \\
 &\quad \langle \pi_1 \cdot M, \overline{N} \cdot \delta M, \pi_2 \cdot M \rangle \\
 &= \{ \text{(92)} \} \\
 &\quad \{ isbn \mapsto (\pi_1(M isbn), N', \pi_2(M isbn)) \mid isbn \in dom M \}
 \end{aligned}$$

where  $N'$  abbreviates  $\{n \mapsto N(i, n) \mid (i, n) \in dom N \wedge i = isbn\}$ .

The fact that  $\overline{N}$  is preconditioned by  $\delta M$  in the abstraction is a clear indication that any addition to  $N$  of authors of books whose ISBN don't participate in  $M$  is doomed to be ignored when “backward mapping” the data. This explains why a foreign key constraint must be added to any SQL encoding of  $Books_2$ , eg.:

```

CREATE TABLE BOOKS (
  ISBN          VARCHAR (...) NOT NULL,
  Publisher     VARCHAR (...) NOT NULL,

```

```

Title      VARCHAR (...) NOT NULL,
CONSTRAINT BOOKS PRIMARY KEY (ISBN)
);

CREATE TABLE AUTHORS (
  ISBN     VARCHAR (...) NOT NULL,
  Count    NUMBER (...) NOT NULL,
  Author   VARCHAR (...) NOT NULL,
  CONSTRAINT AUTHORS_pk PRIMARY KEY (ISBN, Count)
);

ALTER TABLE AUTHORS ADD CONSTRAINT AUTHORS_FK
  FOREIGN KEY (ISBN) REFERENCES BOOKS (ISBN);

```

It can be observed that this constraint is ensured by representation  $r$  (otherwise right-invertibility wouldn't take place). Constraints of this kind are known as *concrete invariants*. We discuss this important notion in the section which follows.

*Summary.* This section described the application of the calculus introduced in section 5 to the transformation of abstract data models targeted at relational database implementations. It also showed how more elaborate laws can be derived from simpler ones and how to synthesize composite “forward” and “backward” data mappings using the underlying relational calculus. We proceed to showing how to take further advantage of relational reasoning in synthesizing data type invariants entailed by the representation process.

## 7 Concrete Invariants

The fact that  $R$  and  $F$  are connected (63) in every  $\leq$ -rule (7) forces the range of  $R$  to be at most the domain of  $F$ ,  $\rho R \subseteq \delta F$ . This means that the representation space ( $B$ ) can be divided in three parts:

- *inside*  $\rho R$  — data inside  $\rho R$  are referred to as *canonical representatives*; the predicate associated to  $\rho R$ , which is the strongest property ensured by the representation, is referred to as the induced *concrete invariant*, or *representation invariant*.
- *outside*  $\delta F$  — data outside  $\delta F$  are *illegal* data: there is no way in which they can be retrieved; we say that the target model is *corrupted* (using the database terminology) once its CRUD drives data into this zone.
- *inside*  $\delta F$  and *outside*  $\rho R$  — this part contains data values which  $R$  never generates but which are retrievable and therefore regarded as *legal* representatives; however, if the CRUD of the target model lets data go into this zone, the range of the representation cannot be assumed as concrete invariant.

The following properties of domain and range

$$\delta R = \ker R \cap id \quad (118)$$

$$\rho R = \text{img } R \cap id \quad (119)$$

$$\rho(R \cdot S) = \rho(R \cdot \rho S) \quad (120)$$

$$\delta(R \cdot S) = \delta(\delta R \cdot S) \quad (121)$$

help in inferring *concrete invariants*, in particular those induced by  $\leq$ -chaining [65].

Concrete invariant calculation, which is in general nontrivial, is softened wherever  $\leq$ -rules are expressed by GCs [19]. In this case, the range of the representation (concrete invariant) can be computed as coreflexive  $r \cdot f \cap id$ , that is, predicate [20]

$$\phi x \stackrel{\text{def}}{=} r(f x) = x \quad (122)$$

As illustration of this process, consider law

$$A \rightarrow B \times C \begin{array}{c} \xrightarrow{\langle (\pi_1 \cdot), (\pi_2 \cdot) \rangle} \\ \leq \\ \xrightarrow{\langle -, - \rangle} \end{array} (A \rightarrow B) \times (A \rightarrow C) \quad (123)$$

which expresses the universal property of the *split* operator, a perfect GC:

$$X \subseteq \langle R, S \rangle \equiv \pi_1 \cdot X \subseteq R \wedge \pi_2 \cdot X \subseteq S \quad (124)$$

Calculation of the concrete invariant induced by [123] follows:

$$\begin{aligned} & \phi(R, S) \\ \equiv & \quad \{ [122] [123] \} \\ & (R, S) = (\pi_1 \cdot \langle R, S \rangle, \pi_2 \cdot \langle R, S \rangle) \\ \equiv & \quad \{ [46] \} \\ & R = R \cdot \delta S \wedge S = S \cdot \delta R \\ \equiv & \quad \{ \delta X \subseteq \Phi \equiv X \subseteq X \cdot \Phi \} \\ & \delta R \subseteq \delta S \wedge \delta S \subseteq \delta R \\ \equiv & \quad \{ [14] \} \\ & \delta R = \delta S \end{aligned}$$

In other words: if equally defined  $R$  and  $S$  are joined and then decomposed again, this will be a lossless decomposition [58].

Similarly, the following concrete invariant can be shown to hold for rule [114] [21]:

$$\phi(M, N) \stackrel{\text{def}}{=} N \cdot \in^\circ \preceq M \quad (125)$$

Finally note the very important fact that, in the case of  $\leq$ -rules supported by perfect GCs, the source datatype is actually *isomorphic* to the subset of the target datatype determined by the *concrete invariant* (as range of the representation function [22]).

<sup>19</sup> Of course, these have to be *perfect* [64] on the source (abstract) side.

<sup>20</sup> See Theorem 5.20 in [11].

<sup>21</sup> See [63] for details.

<sup>22</sup> See the *Unity of opposites* theorem of [5].

Exercise 27. Infer (124) from (41) and universal property

$$X \subseteq (R \cap S) \equiv (X \subseteq R) \wedge (X \subseteq S) \tag{126}$$

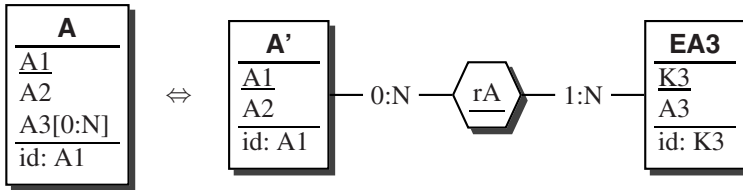
Moreover, show that (40) instantiates (124). □

Exercise 28. Show that (113) is the concrete invariant induced by rule (110), from left-to-right, in case all relations are simple. □

Concrete invariants play an important role in data refinement. For instance, Morgan [49] takes them into account in building *functional abstractions* of the form  $af \cdot \Phi_{dti}$  where (entire) abstraction function  $af$  is explicitly constrained by concrete invariant  $dti$ . In the section which follows we show how such invariants help in calculating model transformations. The reader is also referred to [8] for a PF-theory of invariants in general.

## 8 Calculating Model Transformations

References [30] and [43] postulate a number of model transformation rules (concerning GERs in the first case and UML class diagrams in the second) which we are in position to calculate. We illustrate this process with rule 12.2 of [30], the rule which converts a (multivalued) attribute into an entity type:



The PF-semantics of entity **A** are captured by simple relations from identity  $A_1$  to attributes  $A_2$  and  $A_3$ , this one represented by a powerset due to being [0:N]:

$$A_1 \rightarrow A_2 \times \mathcal{P}A_3$$

The main step in the calculation is the creation of the new entity **EA3** by indirection — recall (104) — whereafter we proceed as before:

$$\begin{aligned} & A_1 \rightarrow A_2 \times \mathcal{P}A_3 \\ \leq_1 & \{ (104) \} \\ & (K_3 \rightarrow A_3) \times (A_1 \rightarrow A_2 \times \mathcal{P}K_3) \\ \cong_2 & \{ (94) \} \\ & (K_3 \rightarrow A_3) \times (A_1 \rightarrow A_2 \times (K_3 \rightarrow 1)) \\ \leq_3 & \{ (114) \} \\ & (K_3 \rightarrow A_3) \times ((A_1 \rightarrow A_2) \times (A_1 \times K_3 \rightarrow 1)) \\ \cong_4 & \{ \text{introduce ternary product} \} \\ & \underbrace{(A_1 \rightarrow A_2)}_{A'} \times \underbrace{(A_1 \times K_3 \rightarrow 1)}_{rA} \times \underbrace{(K_3 \rightarrow A_3)}_{EA3} \end{aligned}$$

The overall concrete invariant is

$$\phi(M, R, N) = R \cdot \epsilon^\circ \preceq M \wedge R \cdot \epsilon^\circ \preceq N$$

— recall eg. (125) — which can be further transformed into:

$$\begin{aligned} \phi(M, R, N) &= R \cdot \epsilon^\circ \preceq M \wedge R \cdot \epsilon^\circ \preceq N \\ &\equiv \{ (54), (53) \} \\ &\quad R \cdot \pi_1^\circ \preceq M \wedge R \cdot \pi_2^\circ \preceq N \\ &\equiv \{ (36) \} \\ &\quad R \preceq M \cdot \pi_1 \wedge R \preceq N \cdot \pi_2 \end{aligned}$$

In words, this means that relationship  $R$  (rA in the diagram) must integrate referentially with  $M$  (**A'** in the diagram) on the first attribute of its compound key and with  $N$  (**EA3** in the diagram) wrt. the second attribute.

The reader eager to calculate the overall representation and abstraction relations will realize that the former is a relation, due to the fact that there are many ways in which the keys of the newly created entity can be associated to values of the  $A3$  attribute. This association cannot be recovered once such keys are abstracted from. So, even restricted by the concrete invariant, the calculated model is surely a valid implementation of the original, but not isomorphic to it. Therefore, the rule should not be regarded as bidirectional.

## 9 On the Impedance of Recursive Data Models

Recursive data structuring is a source of data impedance mismatch because it is not *directly* supported in every programming environment. While functional programmers regard recursion as *the natural way* to programming, for instance, database programmers don't think in that way: somehow trees have to give room to flat data. Somewhere in between is (pointer-based) imperative programming and object oriented programming: direct support for recursive data structures doesn't exist, but dynamic memory management makes it possible to implement them as heap structures involving pointers or object identities.

In this section we address recursive data structure representation in terms of non-recursive ones. In a sense, we want to show how to “get away with recursion” (56) in data modeling. It is a standard result (and every a programmer's experience) that *recursive types using products and sums can be implemented using pointers* (69). Our challenge is to generalize this result and present it in a calculational style.

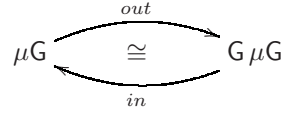
As we have seen already, recursive (finite) data structures are least solutions to equations of the form  $X \cong G X$ , where  $G$  is a relator. The standard notation for such a solution is  $\mu G$ . (This always exists when  $G$  is *regular* (11), a class which embodies all polynomial  $G$ .)

Programming languages which implement datatype  $\mu G$  always do so by *wrapping* it inside some syntax. For instance, the Haskell declaration of datatype `PTree` (38)



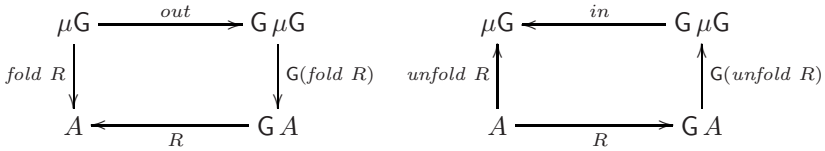
involves constructor `Node` and selectors `name`, `birth`, `mother` and `father`, which cannot be found in equation (51). But this is precisely why the equation expresses isomorphism and not equality: constructor and selectors participate in two bijections which witness the isomorphism and enable one to construct or inspect inhabitants of the datatype being declared.

The general case is depicted in the diagram aside, where *in* embodies the chosen syntax for constructing inhabitants of  $\mu G$  and  $out = in^\circ$  embodies the syntax for destructing (inspecting) such inhabitants. For instance, the *in* bijection associated with `PTree` (38) interpreted as solution to equation (51) is



$$in((n, b), m, f) \stackrel{\text{def}}{=} Node\ n\ b\ m\ f \tag{127}$$

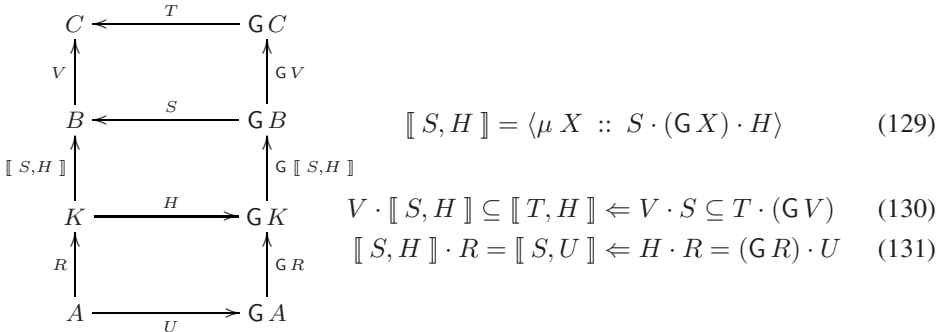
Programs handling  $\mu G$  can be of essentially two kinds: either they read (parse, inspect)  $\mu G$ -structures (trees) or they actually build such structures. The former kind is known as *folding* and the latter as *unfolding*, and both can be pictured as diagrams exhibiting their recursive (inductive) nature:



Both *fold* and *unfold* are instances of a more general, binary combinator known as *hylomorphism* [11], which is normally expressed using the bracketed notation  $\llbracket \_, \_ \rrbracket$  of (129) below to save parentheses:

$$\begin{aligned} unfold\ R &= \llbracket in, R \rrbracket \\ fold\ S &= \llbracket R, out \rrbracket \end{aligned} \tag{128}$$

As fixed points (129), hylomorphisms enjoy a number of so-called *fusion* properties, two of which are listed below for their relevance in calculations to follow<sup>23</sup>:



<sup>23</sup> These and other properties of hylomorphisms arise from the powerful  $\mu$ -fusion theorem [5] once the relational operators involved are identified as lower adjoints in GCs, recall table [1]

In (liberal) Haskell syntax we might write the type of the *unfold* combinator as something like

$$\text{unfold} :: (a \rightarrow g a) \rightarrow a \rightarrow \mu g$$

assuming only functions involved. If we generalize these to simple relations, we obtain the following type for function *unfold*

$$(A \rightarrow \mu G)^{(A \rightarrow G A)}$$

which, thanks to (89), “uncurries” into  $((A \rightarrow G A) \times A) \rightarrow \mu G$ .

Let us temporarily assume that there exists a datatype  $K$  such that simple relation  $Unf$ , of type  $((K \rightarrow G K) \times K) \rightarrow \mu G$  and such that  $\overline{Unf} = \text{unfold}$ , is surjective. Then we are in condition to establish the  $\leq$ -equation which follows,

$$\begin{array}{ccc} & R & \\ \mu G & \begin{array}{c} \xrightarrow{\quad} \\ \leq \\ \xleftarrow{\quad} \end{array} & \underbrace{(K \rightarrow G K) \times K}_{\text{“heap”}} \\ & Unf & \end{array} \tag{132}$$

where  $K$  can be regarded as a data type of “heap addresses”, or “pointers”, and  $K \rightarrow G K$  a datatype of  $G$ -structured heaps<sup>24</sup>. So, assertion  $t \text{Unf } (H, k)$  means that, if pair  $(H, k)$  is in the domain of  $Unf$ , then the abstract value  $t = (\text{unfold } H)k$  will be retrieved — recall (90). This corresponds to dereferencing  $k$  in  $H$  and carrying on doing so (structurally) while building (via *in*) the tree which corresponds to such a walk through the heap.

Termination of this process requires  $H$  to be free of dangling references — ie. satisfy the NSRI property (62) — and to be referentially acyclic. This second requirement can also be expressed via the membership relation associated with  $G$ : relation  $K \xleftarrow{\in_G \cdot H} K$  on references must be well-founded [23].

Jourdan [39] developed a pointwise proof of the surjectiveness of  $Unf$  (132) for  $K$  isomorphic to the natural numbers and  $G$  polynomial (see more about this in section 13). The representation relation  $R$ , which should be chosen among the entire sub-relations of  $Unf^\circ$ , is an injective *fold* (since converses of unfolds are folds [11]). Appendix A illustrates a strategy for encoding such folds, in the case of  $G$  polynomial and  $K$  the natural numbers.

“De-recursion” law (132) generalizes, in the generic PF-style, the main result of [69] and bears some resemblance (at least in spirit) with “defunctionalization” [35], a technique which is used in program transformation and compilation. The genericity of this result and the ubiquity of its translation into practice — cf. name spaces, dynamic memory management, pointers and heaps, database files, object run-time systems, etc — turns it into a useful device for cross-paradigm transformations. For instance, [56] shows how to use it in calculating a universal SQL representation for XML data.

The sections which follow will illustrate this potential, while stressing on genericity [37]. Operations of the *algebra of heaps* such as eg. *defragment* (cf. *garbage-collection*) will be stated generically and be shown to be correct with respect to the abstraction relation.

<sup>24</sup> Technically, this view corresponds to regarding heaps as (finite) relational  $G$ -coalgebras.

## 10 Cross-Paradigm Impedance Handled by Calculation

Let us resume work on the case study started in section 2 and finally show how to map the recursive datatype  $\text{PTree}$  (38) down to a relational model (SQL) via an intermediate heap/pointer representation.

Note that we shall be crossing over three paradigms — functional, imperative and database relational — in a single calculation, using the same notation:

$$\begin{aligned}
 & \text{PTree} \\
 \cong_1 & \quad \{ r_1 = \text{out}, f_1 = \text{in}, \text{for } \mathbf{G} K \stackrel{\text{def}}{=} \text{Ind} \times (K+1) \times (K+1) \text{ — cf. (51) (127)} \} \\
 & \quad \mu\mathbf{G} \\
 \leq_2 & \quad \{ R_2 = \text{Unf}^\circ, F_2 = \text{Unf} \text{ — cf. (132)} \} \\
 & \quad (K \rightarrow \text{Ind} \times (K+1) \times (K+1)) \times K \\
 \cong_3 & \quad \{ r_3 = (\text{id} \rightarrow \text{flatr}^\circ) \times \text{id}, f_3 = (\text{id} \rightarrow \text{flatr}) \times \text{id} \text{ — cf. (43)} \} \\
 & \quad (K \rightarrow \text{Ind} \times ((K+1) \times (K+1))) \times K \\
 \cong_4 & \quad \{ r_4 = (\text{id} \rightarrow \text{id} \times p2p) \times \text{id}, f_4 = (\text{id} \rightarrow \text{id} \times p2p^\circ) \times \text{id} \text{ — cf. (88)} \} \\
 & \quad (K \rightarrow \text{Ind} \times (K+1)^2) \times K \\
 \cong_5 & \quad \{ r_5 = (\text{id} \rightarrow \text{id} \times \text{tot}^\circ) \times \text{id}, f_5 = (\text{id} \rightarrow \text{id} \times \text{tot}) \times \text{id} \text{ — cf. (84)} \} \\
 & \quad (K \rightarrow \text{Ind} \times (2 \rightarrow K)) \times K \\
 \leq_6 & \quad \{ r_6 = \Delta_n, f_6 = \bowtie_n \text{ — cf. (114)} \} \\
 & \quad ((K \rightarrow \text{Ind}) \times (K \times 2 \rightarrow K)) \times K \\
 \cong_7 & \quad \{ r_7 = \text{flatl}, f_7 = \text{flatl}^\circ \text{ — cf. (44)} \} \\
 & \quad (K \rightarrow \text{Ind}) \times (K \times 2 \rightarrow K) \times K \\
 =_8 & \quad \{ \text{since } \text{Ind} = \text{Name} \times \text{Birth} \text{ (51)} \} \\
 & \quad (K \rightarrow \text{Name} \times \text{Birth}) \times (K \times 2 \rightarrow K) \times K
 \end{aligned} \tag{133}$$

In summary:

- Step 2 moves from the functional (inductive) to the pointer-based representation. In our example, this corresponds to mapping inductive tree (9) to the heap of figure 2a.
- Step 5 starts the move from pointer-based to relational-based representation. Isomorphism (84) between *Maybe*-functions and simple relations (which is the main theme of [59]) provides the relevant data-link between the two paradigms: pointers “become” primary/foreign keys.
- Steps 7 and 8 deliver an RDBT structure (illustrated in figure 2b) made up of two tables, one telling the details of each individual, and the other recording its immediate ancestors. The 2-valued attribute in the second table indicates whether the mother or the father of each individual is to be reached. The third factor in (133) is the key which gives access to the root of the original tree.

In practice, a final step is required, translating the relational data into the syntax of the target relational engine (eg. a script of SQL `INSERT` commands for each relation), bringing symmetry to the exercise: in either way (forwards or backwards), data mappings start by *removing* syntax and close by *introducing* syntax.

*Exercise 29.* Let  $f_{4;7}$  denote the composition of abstraction functions  $f_4 \cdot (\dots) \cdot f_7$ . Show that  $(id \multimap \pi_1) \cdot \pi_1 \cdot f_{4;7}$  is the same as  $\pi_1$ .  $\square$

## 11 On the Transcription Level

Our final calculations have to do with what the authors of [42] identify as the *transcription level*, the third ingredient of a *mapping scenario*. This has to do with diagram (10): once two pairs of data maps (“map forward” and “map backward”)  $F, R$  and  $F', R'$  have been calculated so as to represent two source datatypes  $A$  and  $B$ , they can be used to transcribe a given source operation  $B \xleftarrow{O} A$  into some target operation  $D \xleftarrow{P} C$ .

How do we establish that  $P$  *correctly* implements  $O$ ? Intuitively,  $P$  must be such that the performance of  $O$  and that of  $P$  (the latter *wrapped* within the relevant abstraction and representation relations) cannot be distinguished:

$$O = F' \cdot P \cdot R \tag{134}$$

Equality is, however, much too strong a requirement. In fact, there is no disadvantage in letting the target side of (134) be more defined than the source operation  $O$ , provided both are simple<sup>25</sup>:

$$O \subseteq F' \cdot P \cdot R \tag{135}$$

Judicious use of (29, 30) will render (135) equivalent to

$$O \cdot F \subseteq F' \cdot P \tag{136}$$

provided  $R$  is chosen maximal ( $R = F^\circ$ ) and  $F \preceq P$ . This last requirement is obvious:  $P$  must be prepared to cope with all possible representations delivered by  $R = F^\circ$ .

In particular, wherever the source operation  $O$  is a *query*, ie.  $F' = id$  in (136), this shrinks to  $O \cdot F \subseteq P$ . In words: wherever the source query  $O$  delivers a result  $b$  for some input  $a$ , then the target query  $P$  must deliver the same  $b$  for any target value which represents  $a$ .

Suppose that, in the context of our running example (pedigree trees), one wishes to transcribe into SQL the query which fetches the name of the person whose pedigree tree is given. In the Haskell data model `PTree`, this is simply the (selector) function `name`. We want to investigate how this function gets mapped to lower levels of abstraction.

The interesting step is  $\leq_2$ , whereby trees are represented by pointers to heaps. The abstraction relation  $Unf$  associated to this step is inductive. Does this entail inductive

<sup>25</sup> Staying within this class of operations is still quite general: it encompasses all deterministic, possibly partial computations. Within this class, inclusion coincides with the standard definition of *operation refinement* [60].

reasoning? Let us see. Focusing on this step alone, we want to solve equation  $name \cdot Unf \subseteq Hname$  for unknown  $Hname$  — a query of type  $((K \rightarrow GK) \times K) \rightarrow Name$ .

Simple relation currying (91) makes this equivalent to finding  $Hname$  such that, for every heap  $H$ ,  $name \cdot (\overline{Unf} H) \subseteq \overline{Hname} H$  holds, that is,  $name \cdot (unfold H) \subseteq \overline{Hname} H$ . Since both  $unfold H$  and  $\overline{Hname} H$  are hylomorphisms, we write them as such,  $name \cdot \llbracket in, H \rrbracket \subseteq \llbracket T, H \rrbracket$ , so that  $T$  becomes the unknown. Then we calculate:

$$\begin{aligned}
 & name \cdot \llbracket in, H \rrbracket \subseteq \llbracket T, H \rrbracket \\
 \Leftarrow & \quad \{ \text{fusion (130)} \} \\
 & name \cdot in \subseteq T \cdot G(name) \\
 \equiv & \quad \{ name \cdot Node = \pi_1 \cdot \pi_1 \text{ (127)}; \text{expansion of } G(name) \} \\
 & \pi_1 \cdot \pi_1 \subseteq T \cdot (id \times (name + id) \times (name + id)) \\
 \Leftarrow & \quad \{ \pi_1 \cdot (f \times g) = f \cdot \pi_1 \} \\
 & T = \pi_1 \cdot \pi_1
 \end{aligned}$$

Thus

$$\begin{aligned}
 \overline{Hname} H &= \llbracket \pi_1 \cdot \pi_1, H \rrbracket \\
 &= \{ \text{(129)} \} \\
 &\quad \langle \mu X :: \pi_1 \cdot \pi_1 \cdot (id \times (X + id) \times (X + id)) \cdot H \rangle \\
 &= \{ \pi_1 \cdot (f \times g) = f \cdot \pi_1 \} \\
 &\quad \langle \mu X :: \pi_1 \cdot \pi_1 \cdot H \rangle \\
 &= \{ \text{trivia} \} \\
 &\quad \pi_1 \cdot \pi_1 \cdot H
 \end{aligned}$$

Back to *uncurried* format and introducing variables, we get (the post-condition of  $Hname$ )

$$n Hname(H, k) \equiv k \in dom H \wedge n = \pi_1(\pi_1(H k))$$

which means what one would expect: should pointer  $k$  be successfully dereferenced in  $H$ , selection of the *Ind* field will take place, wherefrom the name field is finally selected (recall that  $Ind = Name \times Birth$ ).

The exercise of mapping  $Hname$  down to the SQL level (133) is similar but less interesting. It will lead us to

$$n Rname(M, N, k) = k \in dom M \wedge n = \pi_1(M k)$$

where  $M$  and  $N$  are the two relational tables which originated from  $H$  after step 2.  $Rname$  can be encoded into SQL as something like

SELECT Name FROM M WHERE PID = k

under some obvious assumptions concerning the case in which  $k$  cannot be found in  $M$ . So we are done as far as transcribing *name* is concerned.

The main ingredient of the exercise just completed is the use of fusion property (I130). But perhaps it all was *much ado for little*: queries aren't very difficult to transcribe in general. The example we give below is far more eloquent and has to do with heap housekeeping. Suppose one wants to defragment the heap at level 2 via some reallocation of heap cells. Let  $K \xleftarrow{f} K$  be the function chosen to *rename* cell addresses. Recalling (33), defragmentation is easy to model as a projection:

$$\begin{aligned} \text{defragment} & : (K \longrightarrow K) \longrightarrow (K \rightarrow \mathsf{G} K) \longrightarrow (K \rightarrow \mathsf{G} K) \\ \text{defragment } f H & \stackrel{\text{def}}{=} (\mathsf{G} f) \cdot H \cdot f^\circ \end{aligned} \quad (137)$$

The correctness of *defragment* has two facets. First,  $H \cdot f^\circ$  should remain simple; second, the information stored in  $H$  should be preserved: *the pedigree tree recorded in the heap (and pointer) shouldn't change in consequence of a defragment operation*. In symbols:

$$t \text{ Unf } (\text{defragment } f H, f k) \equiv t \text{ Unf } (H, k) \quad (138)$$

Let us check (I138):

$$\begin{aligned} & t \text{ Unf } (\text{defragment } f H, f k) \equiv t \text{ Unf } (H, k) \\ \equiv & \quad \{ \text{(I132)} ; \text{(I28)} \} \\ & t \llbracket \text{in}, \text{defragment } f H \rrbracket (f k) \equiv t \llbracket \text{in}, H \rrbracket k \\ \equiv & \quad \{ \text{go pointfree (20)} ; \text{definition (I137)} \} \\ & \llbracket \text{in}, (\mathsf{G} f) \cdot H \cdot f^\circ \rrbracket \cdot f = \llbracket \text{in}, H \rrbracket \\ \Leftarrow & \quad \{ \text{fusion property (I131)} \} \\ & (\mathsf{G} f) \cdot H \cdot f^\circ \cdot f = (\mathsf{G} f) \cdot H \\ \Leftarrow & \quad \{ \text{Leibniz} \} \\ & H \cdot f^\circ \cdot f = H \\ \equiv & \quad \{ \text{since } H \subseteq H \cdot f^\circ \cdot f \text{ always holds} \} \\ & H \cdot f^\circ \cdot f \subseteq H \end{aligned}$$

So, condition  $H \cdot f^\circ \cdot f \subseteq H$  (with points:

$$k \in \text{dom } H \wedge f k = f k' \Rightarrow k' \in \text{dom } H \wedge H k = H k'$$

for all heap addresses  $k, k'$ ) is sufficient for *defragment* to preserve the information stored in the heap *and its simplicity*<sup>26</sup>. Of course, any injective  $f$  will qualify for safe defragmentation, for *every* heap.

<sup>26</sup> In fact,  $H \cdot f^\circ \cdot f \subseteq H$  ensures  $H \cdot f^\circ$  simple, via (30) and monotonicity.

Some comments are in order. First of all, and unlike what is common in data refinement involving recursive data structures (see eg. [24] for a comprehensive case study), our calculations above have dispensed with any kind of inductive or coinductive argument. (This fact alone should convince the reader of the advantages of the PF-transform in program reasoning.)

Secondly, the *defragment* operation we’ve just reasoned about is a so-called *representation changer* [34]. These operations (which include garbage collection, etc) are important because they add to efficiency without disturbing the service delivered to the client. In the *mapping scenario* terminology of [42], these correspond to operations which transcribe backwards to the identity function, at source level.

Finally, a comment on CRUD operation transcription. Although CRUD operations in general can be arbitrarily complex, in the process of transcription they split into simpler and simpler middleware and dataware operations which, at the target (eg. database) level end up involving standard protocols for data access [42].

The ubiquity of *simplicity* in data modeling, as shown throughout this paper, invites one to pay special attention to the CRUD of this kind of relation. Reference [57] identifies some “design patterns” for simple relations. The one dealt with in this paper is the *identity pattern*. For this pattern, a succinct specification of the four CRUD operations on simple  $M$  is as follows:

- $Create(N): M \mapsto N \dagger M$ , where (simple) argument  $N$  embodies the new entries to add to  $M$ . The use of the override operator  $\dagger$  [38, 59] instead of union ( $\cup$ ) ensures simplicity and prevents from writing over existing entries.
- $Read(a)$ : deliver  $b$  such that  $b M a$ , if any.
- $Update(f, \Phi): M \mapsto M \dagger f \cdot M \cdot \Phi$ . This is a selective update: the contents of every entry whose key is selected by  $\Phi$  get updated by  $f$ ; all the other remain unchanged.
- $Delete(\Phi): M \mapsto M \cdot (id - \Phi)$ , where  $R - S$  means relational difference (cf. table [1]). All entries whose keys are selected by  $\Phi$  are removed.

Space constraints preclude going further on this topic in this paper. The interested reader will find in reference [57] the application of the PF-transform in speeding-up reasoning about CRUD preservation of datatype invariants on simple relations, as a particular case of the general theory [8]. Similar gains are expected from the same approach applied to CRUD transcription.

*Exercise 30.* Investigate the transcription of selector function *mother* [38] to the heap-and-pointer level, that is, solve  $mother \cdot Unf \subseteq P$  for  $P$ . You should obtain a simple relation which, should it succeed in dereferencing the input pointer, it will follow on to the second position in the heap-cell so as to unfold (if this is the case) and show the tree accessible from that point. The so-called *hylo-computation rule* —  $\llbracket R, S \rrbracket = R \cdot (F \llbracket R, S \rrbracket) \cdot S$  — is what matters this time. □

*Summary.* The transcription level is the third component of a mapping scenario whereby abstract operations are “mapped forward” to the target level and give room to concrete implementations (running code). In the approach put forward in this paper, this is performed by solving an equation [134] where the unknown is the concrete implementation  $P$  one is aiming at. This section gave an example of how to carry out this task in

presence of recursive data structures represented by heaps and pointers. The topic of CRUD operation transcription was also (briefly) addressed.

## 12 Related Work

This section addresses two areas of research which are intimately related to the data transformation discipline put forward in the current paper. One is *bidirectional programming* used to synchronize heterogeneous data formats [13]. The other is the design of term rewriting systems for type-safe data transformation [17].

*Lenses.* The proximity is obvious between abstraction/representation pairs implicit in  $\leq$ -rules and bidirectional transformations known as *lenses* and developed in the context of the classical *view-update problem* [13, 14, 27, 33]. Each lens connects a concrete data type  $C$  with an abstract view  $A$  on it by means of two functions  $A \times C \xrightarrow{put} C$  and  $A \xleftarrow{get} C$ . (Note the similarity with  $(R, F)$  pairs, except for *put*'s additional argument of type  $C$ .)

A lens is said to be *well-behaved* if two conditions hold,

$$get(put(v, s)) = v \quad \text{and} \quad put(get\ s, s) = s$$

known as *acceptability* and *stability*, respectively. For total lenses, these are easily PF-transformed into

$$put \cdot \pi_1^\circ \subseteq get^\circ \tag{139}$$

$$\langle get, id \rangle \subseteq put^\circ \tag{140}$$

which can be immediately recognized as stating the connectivity requirements of  $\leq$ -diagrams

$$\begin{array}{ccc}
 \begin{array}{ccc}
 \pi_1^\circ & \xrightarrow{\quad} & A \times C \\
 \swarrow & & \searrow \text{put} \\
 A & \leq & C \\
 \swarrow & & \searrow \text{get}
 \end{array}
 & \text{and} &
 \begin{array}{ccc}
 & \langle get, id \rangle & \\
 \swarrow & & \searrow \\
 C & \leq & A \times C \\
 \swarrow & & \searrow \text{put}
 \end{array}
 \end{array}
 \tag{141}$$

respectively.

Proving that these diagrams hold in fact is easy to check in the PF-calculus: stability [140] enforces *put* surjective (of course  $\langle get, id \rangle$  is injective even in case *get* is not). Acceptability [139] enforces *get* surjective since it is larger than the converse of entire  $put \cdot \pi_1^\circ$  (recall rules of thumb of exercise 2). Conversely, being at most the converse of a function,  $put \cdot \pi_1^\circ$  is injective, meaning that

$$\begin{aligned}
 & \pi_1 \cdot put^\circ \cdot put \cdot \pi_1^\circ \subseteq id \\
 \equiv & \quad \{ \text{shunting [26, 27] and adding variables} \} \\
 & put(a, c) = put(a', c') \Rightarrow a = a'
 \end{aligned}$$

holds. This fact is known in the literature as the *semi-injectivity* of *put* [27].



*Exercise 31.* A (total, well-behaved) lens is said to be *oblivious* [27] if *put* is of the form  $f \cdot \pi_1$ , for some  $f$ . Use the PF-calculus to show that in this case *get* and  $f$  are bijections, that is,  $A$  and  $C$  in (141) are isomorphic [27]. Suggestion: show that  $get = f^\circ$  and recall (75).  $\square$

Put side by side, the two  $\leq$ -diagrams displayed in (141) express the bidirectional nature of lenses in a neat way [28]. They also suggest that lenses could somehow be “programmed by calculation” in the same manner as the structural transformations investigated in the main body of this paper. See section 13 for future research directions in this respect.

*2LT — a library for two-level data transformation.* The 2LT package of the U.Minho Haskell libraries [10, 17, 18] applies the theory presented in the current paper to data refinement via (typed) strategic term re-writing using GADTs. The refinement process is modeled by a type-changing rewrite system, each rewrite step of which animates a  $\leq$ -rule of the calculus: it takes the form  $A \mapsto (C, to, from)$  where  $C$ , the target type, is packaged with the conversion functions (*to* and *from*) between the old ( $A$ ) and new type ( $C$ ). By repeatedly applying such rewrite steps, complex conversion functions (data mappings) are calculated incrementally while a new type is being derived. (So, 2LT representation mappings are restricted to functions.)

Data mappings obtained after type-rewriting can be subject to subsequent simplification using laws of PF program calculation. Such simplifications include migration of queries on the source data type to queries on a target data type by fusion with the relevant data mappings (a particular case of transcription, as we have seen). Further to PF functional simplification, 2LT implements rewrite techniques for transformation of structure-shy functions (XPath expressions and strategic functions), see eg. [18].

In practice, 2LT can be used to scale-up the data transformation/mapping techniques presented in this paper to real-size case-studies, mainly by mechanizing repetitive tasks and discharging housekeeping duties. More information can be gathered from the project’s website: <http://code.google.com/p/2lt>.

## 13 Conclusions and Future Work

This paper presented a mathematical approach to data transformation. As main advantages of the approach we point out: (a) a unified and powerful notation to describe data-structures across various programming paradigms, and its (b) associated calculus based on elegant rules which are reminiscent of school algebra; (c) the fact that data impedance mismatch is easily expressed by rules of the calculus which, by construction, offer type-level transformations *together with* well-typed data mappings; (d) the properties enjoyed by such rules, which enable their application in a stepwise, structured way.

The novelty of this approach when compared to previous attempts to lay down the same theory is the use of binary relation pointfree notation to express *both* algorithms

<sup>27</sup> This is Lemma 3.9 in [27], restricted to functions.

<sup>28</sup> Note however that, in general, lenses are not entire [27].

and data, in a way which dispenses with inductive proofs and cumbersome reasoning. In fact, most work on the pointfree relation calculus has so far been focused on reasoning about programs (ie. algorithms). Advantages of our proposal to *uniformly* PF-transform both programs *and data* are already apparent at practical level, see eg. the work reported in [50].

Thanks to the PF-transform, opportunities for creativity steps are easier to spot and carry out with less symbol trading. This style of calculation has been offered to Minho students for several years (in the context of the local tradition on formal modeling) as alternative to standard database design techniques<sup>29</sup>. It is the foundation of the “2LT bundle” of tools available from the UMinho Haskell libraries. However, there is still much work to be done. The items listed below are proposed as prompt topics for research.

*Lenses.* The pointwise treatment of lenses as partial functions in [27] is *cpo*-based, entailing the need for continuity arguments. In this paper we have seen that partial functions are *simple* relations easily accommodated in the binary relation calculus. At first sight, generalizing *put* and *get* of section [12] from functions to simple relations doesn’t seem to be particularly hard, even in the presence of recursion, thanks to the PF hylomorphism calculus (recall section [9]).

How much the data mapping formalism presented in the current paper can offer to the theory of bidirectional programming is the subject of on-going research.

*Heaps and pointers at target.* We believe that Jourdan’s long, inductive pointwise argument [39] for  $\leq$ -law [132] can be supplanted by succinct pointfree calculation if results developed meanwhile by Gibbons [29] are taken into account. Moreover, the same law should be put in parallel with other related work on calculating with pointers (read eg. [12] and follow the references).

*Separation logic.* Law [132] has a clear connection to shared-mutable data representation and thus with *separation logic* [62]. There is work on a PF-relational model for this logic [64] which is believed to be useful in better studying and further generalizing law [132] and to extend the overall approach to in-place data-structure updating.

*Concrete invariants.* Taking concrete invariants into account is useful because these ensure (for free) properties at target-data level which can be advantageous in the transcription of source operations. The techniques presented in section [7] and detailed in [63] are the subject of current research taking into account the PF-calculus of invariants of [8]. Moreover,  $\leq$ -rules should be able to take invariants into account (a topic suggested but little developed in [55]).

*Mapping scenarios for the UML.* Following the exercise of section [8], a calculational theory of UML mapping scenarios could be developed starting from eg. K. Lano’s catalogue [43]. This should also take the *Calculating with Concepts* [22] semantics for UML class diagrams into account. For preliminary work on this subject see eg. [9].

<sup>29</sup> The  $\leq$ -rules of the calculus are used in practical classes and lab assignments in the derivation of database schemas from abstract models, including the synthesis of data mappings. The proofs of such rules (as given in the current paper) are addressed in the theory classes.

*PF-transform.* Last but not least, we think that further research on the PF-transform should go along with applying it in practice. In particular, going further and formalizing the analogy with the Laplace transform (which so far has only been hinted at) would be a fascinating piece of research in mathematics and computer science in itself, and one which would *put the vast storehouse in order*, to use the words of Lawvere and Schanuel [44]. In these times of widespread pre-scientific software technology, putting the PF-transform under the same umbrella as other mathematical transforms would contribute to better framing the software sciences within engineering mathematics as a whole.

## Acknowledgments

The author wishes to thank his colleagues at Minho University and his (current and former) students for the warm reception to his (ever evolving) ideas on data calculation. Special thanks go to L.S. Barbosa, to C.J. Rodrigues, to J.C. Ramalho and to the 2LT team *core*: Alcino Cunha, Joost Visser, Tiago Alves and Hugo Pacheco. Jeremy Gibbons comments on the proceedings version of this paper are gratefully acknowledged.

The author is also indebted to the anonymous referees for detailed and helpful comments which improved the paper's presentation and technical contents.

Part of this research was carried out in the context of the PURE Project (*Program Understanding and Re-engineering: Calculi and Applications*) funded by FCT contract POSI/ICHS/44304/2002.

## References

1. Aarts, C., Backhouse, R.C., Hoogendijk, P., Voermans, E., van der Woude, J.: A relational theory of datatypes (December 1992), <http://www.cs.nott.ac.uk/~rcb>
2. Alves, T.L., Silva, P.F., Visser, J., Oliveira, J.N.: Strategic term rewriting and its application to a VDM-SL to SQL conversion. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 399–414. Springer, Heidelberg (2005)
3. Ambler, S.W.: The object-relational impedance mismatch (February 15, 2006), <http://www.agiledata.org/essays/impedanceMismatch.html>
4. Backhouse, K., Backhouse, R.C.: Safety of abstract interpretations for free, via logical relations and Galois connections. SCP 15(1–2), 153–196 (2004)
5. Backhouse, R.C.: Mathematics of Program Construction, pages 608. Univ. of Nottingham (2004); Draft of book in preparation
6. Backhouse, R.C., de Bruin, P., Hoogendijk, P., Malcolm, G., Voermans, T.S., van der Woude, J.: Polynomial relators. In: AMAST 1991, pp. 303–362. Springer, Heidelberg (1992)
7. Backus, J.: Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. CACM 21(8), 613–639 (1978)
8. Barbosa, L.S., Oliveira, J.N., Silva, A.M.: Calculating invariants as coreflexive bisimulations. In: Meseguer, J., Roşu, G. (eds.) AMAST 2008. LNCS, vol. 5140, pp. 83–99. Springer, Heidelberg (2008)
9. Berdager, P.: Algebraic representation of UML class-diagrams, May, Dept. Informatics, U.Minho. Technical note (2007)
10. Berdager, P., Cunha, A., Pacheco, H., Visser, J.: Coupled Schema Transformation and Data Conversion For XML and SQL. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 290–304. Springer, Heidelberg (2006)

11. Bird, R., de Moor, O.: Algebra of Programming. C.A.R. Hoare, series editor, Series in Computer Science. Prentice-Hall International, Englewood Cliffs (1997)
12. Bird, R.S.: Unfolding pointer algorithms. *J. Funct. Program.* 11(3), 347–358 (2001)
13. Bohannon, A., Foster, J.N., Pierce, B.C., Pilkiewicz, A., Schmitt, A.: Boomerang: Resourceful lenses for string data. In: ACM SIGPLAN–SIGACT POPL Symposium, pp. 407–419 (January 2008)
14. Bohannon, A., Vaughan, J.A., Pierce, B.C.: Relational lenses: A language for updateable views. In: Principles of Database Systems (PODS) (2006)
15. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley Longman, Amsterdam (1999)
16. Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. *JACM* 24(1), 44–67 (1977)
17. Cunha, A., Oliveira, J.N., Visser, J.: Type-safe two-level data transformation. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 284–289. Springer, Heidelberg (2006)
18. Cunha, A., Visser, J.: Transformation of structure-shy programs: applied to XPath queries and strategic functions. In: PEPM 2007, pp. 11–20. ACM, New York (2007)
19. Darlington, J.: A synthesis of several sorting algorithms. *Acta Informatica* 11, 1–30 (1978)
20. de Roeper, W.-P., Engelhardt, K., Coenen, J., Buth, K.-H., Gardiner, P., Lakhnech, Y., Stomp, F.: Data Refinement Model-Oriented Proof methods and their Comparison. Cambridge University Press, Cambridge (1999)
21. Deutsch, M., Henson, M., Reeves, S.: Modular reasoning in Z: scrutinising monotonicity and refinement (to appear, 2006)
22. Dijkman, R.M., Pires, L.F., Joosten, S.: Calculating with concepts: a technique for the development of business process support. In: pUML. LNI, vol. 7, pp. 87–98. GI (2001)
23. Doornbos, H., Backhouse, R., van der Woude, J.: A calculational approach to mathematical induction. *Theoretical Computer Science* 179(1–2), 103–135 (1997)
24. Fielding, E.: The specification of abstract mappings and their implementation as  $B^+$ -trees. Technical Report PRG-18, Oxford University (September 1980)
25. Fitzgerald, J., Larsen, P.G.: Modelling Systems: Practical Tools and Techniques for Software Development, 1st edn. Cambridge University Press, Cambridge (1998)
26. Floyd, R.W.: Assigning meanings to programs. In: Schwartz, J.T. (ed.) Proc. Symposia in Applied Mathematics Mathematical Aspects of Computer Science, vol. 19, pp. 19–32. American Mathematical Society (1967)
27. 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), 17 (2007)
28. Frias, M.F.: Fork algebras in algebra, logic and computer science. Logic and Computer Science. World Scientific Publishing Co, Singapore (2002)
29. Gibbons, J.: When is a function a fold or an unfold?, Working document 833 FAV-12 available from the website of IFIP WG 2.1, 57th meeting, New York City, USA (2003)
30. Hainaut, J.-L.: The transformational approach to database engineering. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2005. LNCS, vol. 4143, pp. 95–143. Springer, Heidelberg (2006)
31. He, J., Hoare, C.A.R., Sanders, J.W.: Data refinement refined. In: Robinet, B., Wilhelm, R. (eds.) ESOP 1986. LNCS, vol. 213, pp. 187–196 (1986)
32. Hoogendijk, P.: A Generic Theory of Data Types. PhD thesis, University of Eindhoven, The Netherlands (1997)
33. Hu, Z., Mu, S.-C., Takeichi, M.: A programmable editor for developing structured documents based on bidirectional transformations. In: Proc. ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, pp. 178–189. ACM Press, New York (2004)

34. Hutton, G., Meijer, E.: Back to basics: Deriving representation changers functionally. *Journal of Functional Programming* (1993) (Functional Pearl)
35. Hutton, G., Wright, J.: Compiling exceptions correctly. In: Kozen, D. (ed.) *MPC 2004*. LNCS, vol. 3125, pp. 211–227. Springer, Heidelberg (2004)
36. Jackson, D.: *Software abstractions: logic, language, and analysis*. The MIT Press, Cambridge Mass (2006)
37. Jeuring, J., Jansson, P.: Polytypic programming. In: *Advanced Functional Programming*. Springer, Heidelberg (1996)
38. Jones, C.B.: *Systematic Software Development Using VDM*, 1st edn. Series in Computer Science. Prentice-Hall Int., Englewood Cliffs (1986)
39. Jourdan, I.S.: Reificação de tipos abstractos de dados: Uma abordagem matemática. Master's thesis, University of Coimbra (1992) (in Portuguese)
40. Kahl, W.: Refinement and development of programs from relational specifications. *ENTCS* 4, 1–4 (2003)
41. Kreyszig, E.: *Advanced Engineering Mathematics*, 6th edn. J. Wiley & Sons, Chichester (1988)
42. Lämmel, R., Meijer, E.: Mappings make data processing go round. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) *GTTSE 2005*. LNCS, vol. 4143, pp. 169–218. Springer, Heidelberg (2006)
43. Lano, K.: *Catalogue of model transformations*, <http://www.dcs.kcl.ac.uk/staff/kcl/>
44. Lawvere, B., Schanuel, S.: *Conceptual Mathematics: a First Introduction to Categories*. Cambridge University Press, Cambridge (1997)
45. Maier, D.: *The Theory of Relational Databases*. Computer Science Press (1983)
46. McCarthy, J.: Towards a mathematical science of computation. In: Poplewell, C.M. (ed.) *Proc. IFIP 62*, pp. 21–28. North-Holland Pub.Company, Amsterdam (1963)
47. McLarty, C.: *Elementary Categories, Elementary Toposes*, 1st edn. Oxford Logic Guides, vol. 21. Calendron Press, Oxford (1995)
48. Meng, S., Barbosa, L.S.: On refinement of generic state-based software components. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) *AMAST 2004*. LNCS, vol. 3116, pp. 506–520. Springer, Heidelberg (2004) (Best student co-authored paper award)
49. Morgan, C.: *Programming from Specification*. C.A.R. Hoare, series (ed.), Series in Computer Science. Prentice-Hall International, Englewood Cliffs (1990)
50. Necco, C., Oliveira, J.N., Visser, J.: Extended static checking by strategic rewriting of point-free relational expressions. Technical Report FAST:07.01, CCTC Research Centre, University of Minho (2007)
51. Oliveira, J.N.: Refinamento transformacional de especificações (terminais). In: *Proc. of XII Jornadas Luso-Espanholas de Matemática*, vol. II, pp. 412–417 (May 1987)
52. Oliveira, J.N.: A Reification Calculus for Model-Oriented Software Specification. *Formal Aspects of Computing* 2(1), 1–23 (1990)
53. Oliveira, J.N.: Invited paper: Software Reification using the SETS Calculus. In: Denvir, T., Jones, C.B., Shaw, R.C. (eds.) *Proc. of the BCS FACS 5th Refinement Workshop, Theory and Practice of Formal Software Development*, London, UK, pp. 140–171. Springer, Heidelberg (1992)
54. Oliveira, J.N.: Data processing by calculation. In: *6th Estonian Winter School in Computer Science*, Palmse, Estonia, March 4-9, 2001. Lecture notes, pages 108 (2001)
55. Oliveira, J.N.: Constrained datatypes, invariants and business rules: a relational approach, PUReCafé, DI-UM, 2004.5.20 [talk], PUR<sub>E</sub> Project (POSI/CHS/44304/2002) (2004)
56. Oliveira, J.N.: Calculate databases with simplicity, Presentation at the IFIP WG 2.1 #59 Meeting, Nottingham, UK (September 2004) (Slides available from the author's website)
57. Oliveira, J.N.: Reinvigorating pen-and-paper proofs in VDM: the pointfree approach. In: *The Third OVERTURE Workshop*, Newcastle, UK, 27-28 November (2006)

58. Oliveira, J.N.: Pointfree foundations for (generic) lossless decomposition (submitted, 2007)
59. Oliveira, J.N., Rodrigues, C.J.: Transposing relations: from Maybe functions to hash tables. In: Kozen, D. (ed.) MPC 2004. LNCS, vol. 3125, pp. 334–356. Springer, Heidelberg (2004)
60. Oliveira, J.N., Rodrigues, C.J.: Pointfree factorization of operation refinement. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085. pp. 236–251. Springer, Heidelberg (2006)
61. Pratt, V.: Origins of the calculus of binary relations. In: Proc. of the 7th Annual IEEE Symp. on Logic in Computer Science, pp. 248–254. IEEE Computer Society Press, Los Alamitos (1992)
62. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS, pp. 55–74 (2002)
63. Rodrigues, C.J.: Software Refinement by Calculation. PhD thesis, Departamento de Informática, Universidade do Minho (submitted, 2007)
64. Wang, S., Barbosa, L.S., Oliveira, J.N.: A relational model for confined separation logic. In: TASE 2008, The 2nd IEEE International Symposium on Theoretical Aspects of Software Engineering, June 17 - 19. LNCS. Springer, Heidelberg (2008)
65. Sestoft, P.: Deriving a lazy abstract machine. *J. Funct. Program* 7(3), 231–264 (1997)
66. Sheard, T., Pasalic, E.: Two-level types and parameterized modules. *Journal of Functional Programming* 14(5), 547–587 (2004)
67. Thomas, D.: The impedance imperative tuples + objects + infosets =too much stuff! *Journal of Object Technology* 2(5) (September/ October 5, 2003)
68. Visser, J.: Generic Traversal over Typed Source Code Representations. Ph. D. dissertation, University of Amsterdam, Amsterdam, The Netherlands (2003)
69. Wagner, E.G.: All recursive types defined using products and sums can be implemented using pointers. In: Bergman, C., Maddux, R.D., Pigozzi, D. (eds.) Algebraic Logic and Universal Algebra in Computer Science. LNCS, vol. 425. Springer, Heidelberg (1990)
70. Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof. Prentice-Hall, Inc., Upper Saddle River (1996)

## A PTree Example in Haskell

This annex presents the exercise, in Haskell, of representing inductive type `PTree` (38) by pointers and heaps. For simplicity, the datatype of `PTree`-shaped heaps is modeled by finite lists of pairs, together with a pointer telling where to start from:

```
data Heap a k = Heap [(k, (a, Maybe k, Maybe k))] k
```

It is convenient to regard this datatype as a bifunctor (30):

```
instance BiFunctor Heap
  where bmap g f
        (Heap h k') =
          Heap [ (f k) |-> (g a, fmap f p, fmap f p')
                | (k, (a, p, p')) <- h ]
          (f k')
```

<sup>30</sup> Note the sugaring of pairing in terms of the infix combinator `x |-> y = (x, y)`, as suggested by (33). Class `BiFunctor` is the binary extension to standard class `Functor` offering `bmap :: (a -> b) -> (c -> d) -> (f a c -> f b d)`, the binary counterpart of `fmap`.

The chosen (functional) representation is a *fold* over PTree,

```
r (Node n b m f) = let x = fmap r m
                    y = fmap r f
                    in merge (n,b) x y
```

where merge is the interesting function:

```
merge a (Just x) (Just y) =
    Heap ([ 1 |-> (a, Just k1, Just k2) ] ++ h1 ++ h2) 1
    where (Heap h1 k1) = bmap id even_ x
          (Heap h2 k2) = bmap id odd_ y
merge a Nothing Nothing =
    Heap ([ 1 |-> (a, Nothing, Nothing) ]) 1
merge a Nothing (Just x) =
    Heap ([ 1 |-> (a, Nothing, Just k2) ] ++ h2) 1
    where (Heap h2 k2) = bmap id odd_ x
merge a (Just x) Nothing =
    Heap ([ 1 |-> (a, Just k1, Nothing) ] ++ h1) 1
    where (Heap h1 k1) = bmap id even_ x
```

Note the use of two functions

```
even_ k = 2*k
odd_ k = 2*k+1
```

which generate the  $k$ th even and odd numbers. Functorial renaming of heap addresses via these functions (whose ranges are disjoint) ensure that the heaps one is joining (via list concatenation) are *separate* [62, 64]. This representation technique is reminiscent of that of storing “binary heaps” (which are not quite the same as in this paper) as arrays without pointers<sup>31</sup>. It can be generalized to any polynomial type of degree  $n$  by building  $n$ -functions  $f_i k \stackrel{\text{def}}{=} nk + i$ , for  $0 \leq i < n$ .

Finally, the abstraction relation is encoded as a partial function in Haskell as follows:

```
f (Heap h k) = let Just (a,x,y) = lookup k h
                in Node (fst a) (snd a)
                   (fmap (f . Heap h) x)
                   (fmap (f . Heap h) y)
```

<sup>31</sup> See eg. entry `Binary_heap` in the Wikipedia.



# How to Write Fast Numerical Code: A Small Introduction

Srinivas Chellappa, Franz Franchetti, and Markus Püschel

Electrical and Computer Engineering  
Carnegie Mellon University  
{schellap, franzf, pueschel}@ece.cmu.edu

**Abstract.** The complexity of modern computing platforms has made it extremely difficult to write numerical code that achieves the best possible performance. Straightforward implementations based on algorithms that minimize the operations count often fall short in performance by at least one order of magnitude. This tutorial introduces the reader to a set of general techniques to improve the performance of numerical code, focusing on optimizations for the computer's memory hierarchy. Further, program generators are discussed as a way to reduce the implementation and optimization effort. Two running examples are used to demonstrate these techniques: matrix-matrix multiplication and the discrete Fourier transform.

## 1 Introduction

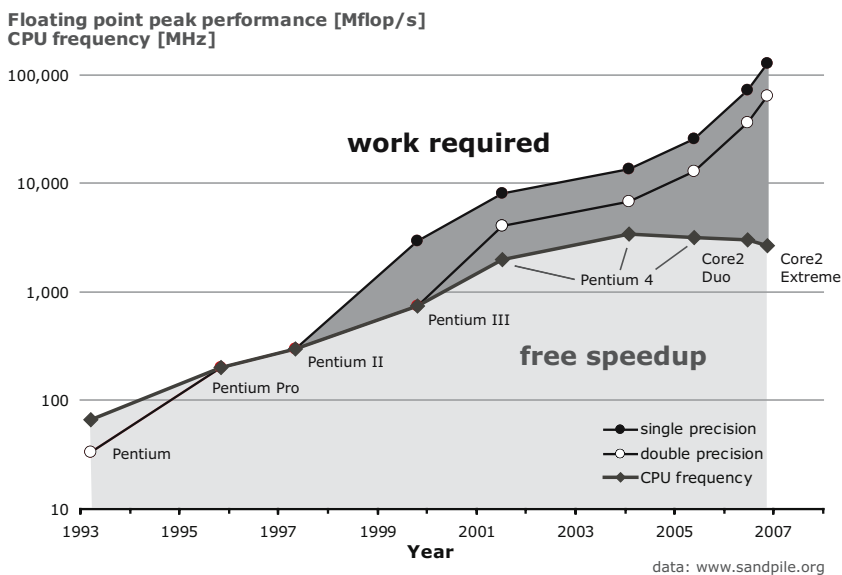
The growth in the performance of computing platforms in the past few decades has followed a reliable pattern usually referred to as Moore's Law. Moore observed in 1965 [1] that the number of transistors per chip roughly doubles every 18 months and predicted—correctly—that this trend would continue. In parallel, due to the shrinking size of transistors, CPU frequencies could be increased at roughly the same exponential rate. This trend has been the big supporter for many performance demanding applications in scientific computing (such as climate modeling and other physics simulations), consumer computing (such as audio, image, and video processing), and embedded computing (such as control, communication, and signal processing). In fact, these domains have a practically unlimited need for performance (for example, the ever growing need for higher resolution videos), and it seems that the evolution of computers is well on track to support these needs.

However, everything comes at a price, and in this case it is the increasing difficulty of writing the fastest possible software. In this tutorial, we focus on *numerical* software. By that we mean code that mainly consists of floating point computations.

**The problem.** To understand the problem we investigate Fig. 1, which considers various Intel architectures from the first Pentium to the (at the time of this writing) latest Core2 Extreme. The  $x$ -axis shows the year of release. The  $y$ -axis, in log-scale, shows both the CPU frequency (in MHz) and the single/double precision theoretical peak performance (in Mflop/s = Mega Floating point Operations per Second) of the respective machines. First we note, as explained above, the exponential increase in CPU frequency. This



## Evolution of Intel Platforms



**Fig. 1.** The evolution of computing platform’s peak performance versus their CPU frequency explains why high performance software development becomes increasingly harder

results in a “free” speedup for numerical software. In other words, legacy code written for an obsolete predecessor will run faster without any extra programming effort. However, the theoretical performance of computers has evolved at a faster pace due to increases in the processors’ parallelism. This parallelism comes in several forms, including pipelining, superscalar processing, vector processing and multi-threading. Single-instruction multiple-data (SIMD) vector instructions enable the execution of an operation on 2, 4, or more data elements in parallel. The latest generations are also “multicore,” which means 2, 4, or more processing cores<sup>1</sup> exist on a single chip. Exploiting parallelism in numerical software is not trivial, it requires implementation effort. Legacy code typically neither includes vector instructions, nor is it multi-threaded to take advantage of multiple processor cores or multiple processors. Ideally, compilers would take care of this problem by automatically vectorizing and parallelizing existing source code. However, while much outstanding compiler research has attacked these problems (e.g., [2, 3, 4]), they are in general still unsolved. Experience shows that this is particularly true for numerical problems. The reason is, for numerical problems, taking advantage of the platform’s available parallelism often requires an algorithm structured differently than the one that would be used in the corresponding sequential code. Compilers cannot be made to change or restructure algorithms since doing so requires knowledge of the algorithm domain.

<sup>1</sup> At the time of this writing 8 cores per chip is the best commonly available multicore CPU configuration.

Similar problems are caused by the computer’s memory hierarchy, independently of the available parallelism. The fast processor speeds have made it increasingly difficult to “feed all floating point execution units” at the necessary rate to keep them busy. Moving data from and to memory has become the bottleneck. The memory hierarchy, consisting of registers and multiple levels of cache, aims to address this problem, but can only work if data is accessed in a suitable order. One cache miss may incur a penalty of 20–100s CPU cycles, a time in which 100 or more floating point operations could have been performed. Again, compilers are inherently limited in optimizing for the memory hierarchy since optimization may require algorithm restructuring or an entirely different choice of algorithm to begin with.

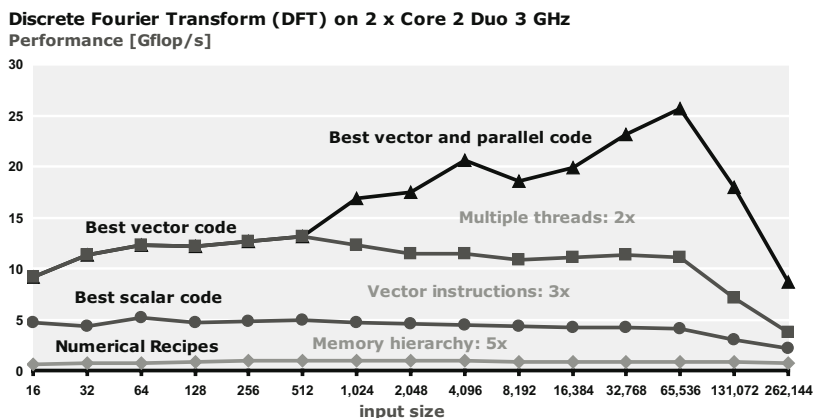
Adding to these problems is the fact that CPU frequency scaling is approaching its end due to limits to the chip’s possible power density (see Fig. 1): since 2004 it has hovered around 3 GHz. This implies *the end of automatic speedup*; future performance gains will be exclusively due to increasing parallelism.

In summary, two main problems can be identified from Fig. 1

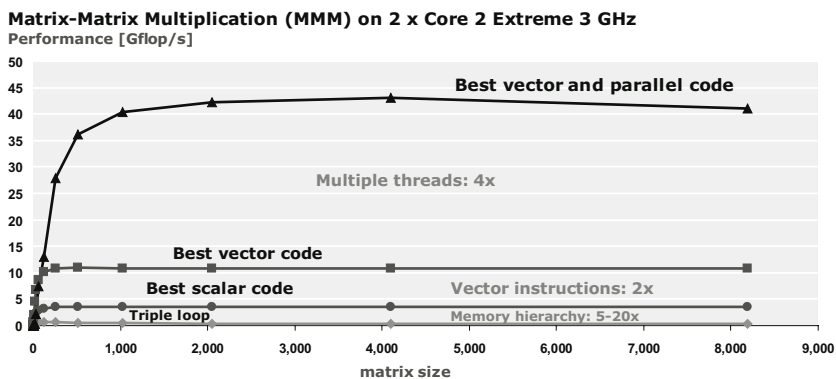
- Years of exponential increase in CPU frequency meant free speed-up for existing software but also have caused and worsened the processor-memory bottleneck. This means to achieve the highest possible performance, code has to be restructured and tuned to the memory hierarchy.
- The times of free speed-up are over; future performance gains are due to parallelism in various forms. This means, code has to be rewritten using vector instructions and multiple threads and in addition has to be optimized for the memory hierarchy.

To quantify the problem we look at two representative examples, which are among the most important numerical kernels used: the discrete Fourier transform (DFT) and the matrix-matrix multiplication (MMM). The DFT is used across disciplines and is the most important tool used in signal processing; MMM is the crucial kernel in most dense linear algebra algorithms.

It is well-known that the complexity of the DFT for input size  $n$  is  $O(n \log(n))$  due to the availability of fast Fourier transform algorithms (FFTs) [5]. Fig. 2 shows the performance of four different FFT implementations on an Intel Core platform with four cores. The  $x$ -axis is the input size  $n = 2^4, \dots, 2^{18}$ . The  $y$ -axis is the performance in Gflop/s. For all implementations, the operations count is estimated as  $5n \log_2(n)$ , so the numbers are proportional to inverse runtime. The bottom line shows the performance of the implementation by Numerical Recipes [6] compiled with the best available compiler (the Intel vendor compiler `icc 10.1` in this case) and all optimizations enabled. The next line (best scalar) shows the performance of the fastest standard C implementation for the DFT and is roughly 5 times faster due to optimizations for the memory hierarchy. The next line (best vector) shows the performance when vector instructions are used in addition, for a further gain of a factor of 3. Finally, for large sizes, another factor of 2 can be gained by writing multi-threaded code to use all processor cores. Note that all four implementations *have roughly the same operations count* for a given size but the performance difference is a factor of 12 for small sizes, and a factor of up to 30 for large sizes. The uppermost three lines correspond to code generated by Spiral [7, 8]; a roughly similar performance is achieved by FFTW [9, 10, 11].



**Fig. 2.** Performance of four single precision implementations of the discrete Fourier transform. The operations count is roughly the same.



**Fig. 3.** Performance of four double precision implementations of matrix-matrix multiplication. The operations count is exactly the same.

Fig. 3 shows a similar plot for MMM (assuming square matrices), where the bottom line corresponds to a standard, triple loop implementation. Here the performance difference with respect to the best code can be as much as 160 times, including a factor of 5-20 solely due to optimizations for the memory hierarchy. All the implementations have exactly the same floating point operations count of  $2n^3$ . The top two lines are from Goto BLAS [12]; the best scalar code is generated using ATLAS [13].

To summarize the above discussion, the task of achieving the highest performance with an implementation usually lies to a great extent with the programmer. For a given problem, he or she has to carefully consider different algorithms and possibly restructure them to adapt to the given platform's memory hierarchy and available parallelism. This is very difficult, time-consuming, and requires interdisciplinary knowledge about algorithms, software optimizations, and the hardware architecture. Further, the tuning

process is platform-dependent: an implementation optimized for one computer will not necessarily be the fastest one on another, since performance depends on many microarchitectural features including but not restricted to the details of the memory hierarchy. Consequently, to achieve highest performance, tuning has to be repeated with the release of each new platform. Since the times of a free speedup (due to frequency scaling) are over, this retuning has become mandatory if any performance gains are desired. Needless to say, the problem is not merely an academic one, but one that affects the software industry as a whole.

**Automatic performance tuning.** A number of research efforts have started to address this problem in a new area called “automatic performance tuning” [14]. The general idea is to at least partially automate the implementation and optimization procedure. Two basic approaches have emerged so far in this area: adaptive libraries and source code generators.

Examples of adaptive libraries include FFTW [10] for the discrete Fourier transform and adaptive sorting libraries [15, 16]. In both cases, the libraries are highly optimized, and beyond that, have degrees of freedom with regard to the chosen divide-and-conquer strategy (both DFT and sorting are done recursively in these libraries). This strategy is determined at runtime, on the given platform, using a search mechanism. This way, the library can dynamically adapt to the computer’s memory hierarchy. Sparsity and OSKI from the BeBOP group [17, 18, 19, 20] are other examples of such a libraries, used for sparse linear algebra problems.

On the other hand, source code generators produce algorithm implementations from scratch. They are used to generate either crucial components, or libraries in their entirety. For instance, ATLAS (Automatically Tuned Linear Algebra Software) and its predecessor PHiPAC [18, 21, 22] generate the kernel code for MMM and other basic matrix routines. They do so by generating many different variants arising from different choices of blocking, loop unrolling, and instruction ordering. These are all measured and the fastest one is selected using search methods.

FFTW also uses a generator to produce small size DFT kernels [23]. Here, no search is used, but many optimizations are performed before the actual code is output. Spiral [7, 24] is a library generator for arbitrary sized linear transforms including the DFT, filters, and others. Besides enumerating alternatives, and in contrast to other work, Spiral uses an internal domain-specific mathematical language to optimize algorithms at a high level of abstraction before source code is generated. This includes algorithm restructuring for the memory hierarchy, vector instructions, and multi-threaded code [24, 25, 26]. FLAME considers dense linear algebra algorithm and is in spirit similar to Spiral. It represents algorithms in a structural form and shows how to systematically derive alternatives and parallelize them [27, 28, 29].

Other automatic performance tuning efforts include [17] for sparse linear algebra and [30] for tensor computations.

This new research is promising but much more work is needed to automate the implementation and optimization of a large set of library functionality. We believe that program generation techniques will prove crucial for this area of research.

**Summary.** We summarize the main points of this section:

- *End of free-speedup for legacy code.* CPU frequencies have hit the power wall and stalled. Future performance gains in computers will be obtained by increasing parallelism. This means that code has to be rewritten to take advantage of the available parallelism and performance.
- *Minimizing operations count does not mean maximizing performance.* Floating-point operations are much cheaper than cache misses. Fastest performance requires code that is adapted to the memory hierarchy, uses vector instructions and multiple cores (if available). As a consequence, we have the following problem.
- *The performance difference between a straightforward implementation and the best possible can be a factor of 10, 20, or more.* This is true even if the former is based on an algorithm that is optimal in its (floating-point) operations count.
- *It is very difficult to write the fastest possible code.* The reason is that performance-optimal code has to be carefully optimized for the platform’s memory hierarchy and available parallelism. For numerical problems, compilers cannot perform these optimizations, or can only perform them to a very limited extent.
- *Performance is in general non-portable.* The fastest code for one computer may perform poorly on another.
- *Overcoming these problems* by automation is a challenge at the core of computer science. To date this research area is still in its infancy. One crucial technique that emerges in this research area is generative programming.

**Goal of this tutorial.** The goal of this tutorial is twofold. First, it provides the reader with a small introduction to the performance optimization of numerical problems, focusing on optimizations for the computer’s memory hierarchy, i.e., the dark area in Fig. 1 is not discussed. The computers considered in this tutorial are COTS (commercial off-the-shelf) desktop computers with the latest microarchitectures such as Core2 Duo or the Pentium from Intel, the Opteron from AMD, and the PowerPC from Apple and Motorola. We assume that the reader has the level of knowledge of a junior (third year) student in computer science or engineering. This includes basic knowledge of computer architecture, algorithms, matrix algebra, and solid C programming skills.

Second, we want to raise awareness and bring this topic closer to the program generation community. Generative programming is an active field of research (e.g., [31, 32]), but has to date mostly focused on reducing the implementation effort in producing correct code. Numerical code and performance optimization have not been considered. In contrast, in the area of automatic performance tuning, program generation has started to emerge as one promising tool as briefly explained in this tutorial. However, much more research is needed and any advances have high impact potential.

The tutorial is in part based on the course [33].

**Organization.** Section 2 provides some basic background information on algorithm analysis, the MMM and the DFT, features of modern computer systems relevant to this tutorial, and compilers and their correct usage. It also identifies data access patterns that are necessary for obtaining high performance on modern computer systems. Section 3 first introduces the basics of benchmarking numerical code and then provides a general high-level procedure for attacking the problem of performance optimization given

an existing program that has to be tuned for performance. This procedure reduces the problem to the optimization of performance-critical kernels, which is first studied in general in Section 4 and then in Sections 5 and 6 using MMM and the DFT as examples. The latter two sections also explain how program generators can be applied in this domain using ATLAS (for MMM) and Spiral (for the DFT) as examples. We conclude with Section 7

Along with the explanations, we provide programming exercises to provide the reader with hands-on experience.

## 2 Background

In this section we provide the necessary background for this tutorial. We briefly review algorithm analysis, introduce MMM and the DFT, discuss the memory hierarchy of off-the-shelf microarchitectures, and explain the use of compilers. The following standard books provide more information on algorithms [34], MMM and linear algebra [35], the DFT [5, 36], and computer architecture and systems [37, 38].

### 2.1 Cost Analysis of Algorithms

The starting point for any implementation of a numerical problem is the choice of algorithm. Before an actual implementation, algorithm analysis, based on the number of operations performed, can give a rough estimate of the performance to be expected. We discuss the floating point operations count and the degree of reuse.

**Cost: asymptotic, exact, and measured.** It is common in algorithm analysis to represent the asymptotic runtime of an algorithm in  $O$ -notation as  $O(f(n))$ , where  $n$  is the input size and  $f$ , a function [34]. For numerical algorithms,  $f(n)$  is typically determined from the number of floating point operations performed. The  $O$ -notation neglects constants and lower order terms; for example,  $O(n^3 + 100n^2) = O(5n^3)$ . Hence it is only suited to describe the performance *trend* but not the *actual* performance itself. Further, it makes a statement only about the asymptotic behavior, i.e., the behavior as  $n$  goes to infinity. Thus it is in principle possible that an  $O(n^3)$  algorithm performs better than an  $O(n^2)$  algorithm for all practically relevant input sizes  $n$ .

A better form of analysis for numerical algorithms is to compute the *exact* number of floating point operations, or at least the exact highest order term. However, this may be difficult in practice. In this case, profiling tools can be used on an actual implementation to determine the number of operations actually performed. The latter can also be used to determine the computational bottleneck in a given implementation.

However, even if the exact number of operations of an algorithm and its implementation is known, it is very difficult to determine the actual runtime. As an example consider Fig. 3: all four implementations require exactly  $2n^3$  operations, but the runtime differs by up to two orders of magnitude.

**Reuse: CPU bound vs. memory bound.** Another useful measure of an algorithm is the degree of reuse. The asymptotic reuse for an  $O(f(n))$  algorithm is given by  $O(f(n)/n)$  if  $n$  is the input size. Intuitively, the degree of reuse measures how often a given input

value is used in a computation during the algorithm. A high degree of reuse implies that an algorithm may perform better (in terms of operations per second) on a computer with memory hierarchy, since the number of computations dominates the number of data transfers from memory to CPU. In this case we say that the algorithm is *CPU bound*. A low degree of reuse implies that the number of data transfers from memory to CPU is high compared to the number of operations and the performance (in operations per second) may deteriorate: in this case we say that the algorithm is *memory bound*.

A CPU bound algorithm will run faster on a machines with a faster CPU. A memory bound algorithm will run faster on a machine with a faster memory bus.

## 2.2 Matrix-Matrix Multiplication

Matrix-matrix multiplication (MMM) is arguably the most important numerical kernel functionality. It is used in many linear algebra algorithms such as solving systems of linear equations, matrix inversion, eigenvalue computations, and many others. We will use MMM, and the DFT (Section 2.3) as examples to demonstrate optimizations for performance.

**Definition.** Given a  $k \times m$  matrix  $A = [a_{i,j}]$  and an  $m \times n$  matrix  $B = [b_{i,j}]$ , the product  $C = AB$  is a  $k \times n$  matrix with entries

$$c_{i,j} = \sum_{k=1}^m a_{i,k} b_{k,j}.$$

For actual applications, usually  $C = C + AB$  is implemented instead of  $C = AB$ .

**Complexity and analysis.** Given two  $n \times n$  matrices  $A, B$ , MMM computed as  $C = C + AB$  by definition requires  $n^3$  multiplications and  $n^3$  additions for a total of  $2n^3 = O(n^3)$  floating point operations. Since the input data (the matrices) have size  $O(n^2)$ , the reuse is given by  $O(n^3/n^2) = O(n)$ .

Asymptotically better MMM algorithms do exist. Strassen's algorithm [39] requires only  $O(n^{\log_2 7}) \approx O(n^{2.808})$  operations. The actual crossover point (i.e., when it requires less operations than the computation by definition) is at  $n = 655$ . However, the more complicated structure of Strassen's algorithm and its weaker numerical stability reduce its applicability. The best-known algorithm for MMM is due to Coppersmith-Winograd and requires  $O(n^{2.376})$  [40]. The large hidden constant and a complicated structure have so far made this algorithm impractical for real applications.

**Direct implementation.** A direct implementation of MMM is the triple loop shown below.

```
// MMM - direct implementation
for(i=0; i<m; i++)
  for(j=0; j<p; j++)
    for(k=0; k<n; k++)
      c[i][j] += a[i][k] * b[k][j];
```

**BLAS and LAPACK.** BLAS (Basic Linear Algebra Subprogram) is a set of standardized basic linear algebra operations, including MMM [41]. Implementations of BLAS



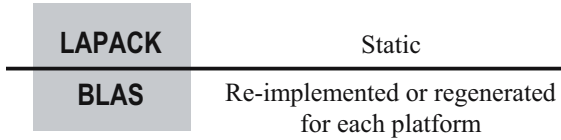


Fig. 4. LAPACK is implemented on top of BLAS

are provided by packages such as ATLAS and Goto BLAS. BLAS routines are used as kernels in fundamental linear algebra algorithms such as linear equation solving, eigenvalue computations, singular value decompositions, LU/Cholesky/QR decompositions, and others. Such higher level functions are implemented by the LAPACK (Linear Algebra PACKage) library, [42] using MMM and other BLAS routines as kernels (see Fig. 4). The idea behind this two-level design is to redesign and/or re-optimize the BLAS implementations for new hardware architectures, while reusing LAPACK without a need for modification. The performance improvements from the BLAS implementation then translate into performance gains for the LAPACK library. This design has proven very successful until the release of multicore systems, which appears to require a redesign of LAPACK.

### Further reading

- *Linear algebra*. General information about numerical linear algebra can be found in [35, 38].
- *BLAS*. ATLAS provides an implementation of BLAS, as does Goto BLAS. Further information on ATLAS is available in [13, 21, 43]. Details on Goto BLAS can be found at [12, 44].
- *Linear algebra libraries*. LAPACK is described in [42, 45]. The distributed memory extension ScaLAPACK is described in [46, 47]. An alternative approach is pursued by PLAPACK [48, 49] and FLAME [27, 28, 50].

## 2.3 Discrete Fourier Transform

The discrete Fourier transform (DFT) is another numerical kernel of importance in a wide range of disciplines. In particular, in the field of signal processing, the DFT is arguably the most important tool used. Even though the DFT seems at first glance based on linear algebra, it is in its nature fundamentally different from MMM. In particular, it is never computed by definition—fast algorithms are always used, instead. The techniques used by these fast algorithms are different from the techniques used to speed up MMM.

**Definition.** The discrete Fourier transform (DFT) of an input vector  $x$  of length  $n$  is defined as the matrix-vector product

$$y = \text{DFT}_n x, \quad \text{DFT}_n = [\omega_n^{k\ell}]_{0 \leq k, \ell < n}, \quad \omega_n = e^{-2\pi i/n}, \quad i = \sqrt{-1}.$$

In words,  $\omega_n$  is a primitive  $n$ th root of unity. In this tutorial we assume that  $n$  is a two-power.



**Complexity and analysis.** Computing the DFT by definition requires  $O(n^2)$  many operations, and is never done in practice. There exists a number of fast algorithms, called fast Fourier transforms (FFTs), that reduce the runtime to  $O(n \log(n))$  for all sizes  $n$  [5]. For  $n = 2^k$ , the FFTs used in practice require between  $4n \log_2(n) + O(n)$  and  $5n \log_2(n) + O(n)$  many operations. The best known FFT has a cost of  $\frac{34}{9}n \log_2 n + O(n)$  [51]. The degree of reuse is hence  $O(\log(n))$ , less than for MMM, which explains the performance drop in Fig. 2 for large sizes when the working set is too large for the L2 cache.

We defer a detailed introduction of FFTs to Section 6.

**Direct implementation.** In contrast to MMM, a straightforward implementation of the DFT is not done by definition, but performed by a direct implementation of an FFT. One example is the so-called iterative radix-2 FFT algorithm as implemented by Numerical Recipes [6], whose performance was shown in Fig. 2. The corresponding code is shown below.

```
#include <math.h>

#define SWAP(a,b) tempr=a;a=b;b=tempr
void four1(float *data, int *nn, int *isign)
{ /* altered for consistency with original FORTRAN.
   * Press, Flannery, Teukolsky, Vettering "Numerical
   * Recipes in C" tuned up ; Code works only when *nn is
   * a power of 2 */
  int n, mmax, m, j, i;
  double wtemp, wr, wpr, wpi, wi, theta, wpin;
  double tempr, tempi, datar, datai,
         datalr, datali;
  n = *nn * 2;
  j = 0;
  for(i = 0; i < n; i += 2)
  { if (j > i) { /* could use j>i+1 to help
                * compiler analysis */
      SWAP(data[j], data[i]);
      SWAP(data[j + 1], data[i + 1]);
    }
    m = *nn;
    while (m >= 2 && j >= m) {
      j -= m;
      m >>= 1;
    }
    j += m;
  }
  theta = 3.141592653589795 * .5;
  if (*isign < 0)
    theta = -theta;
  wpin = 0; /* sin(++PI) */
  for(mmax = 2; n > mmax; mmax *= 2)
  { wpi = wpin;
```

```

wpin = sin(theta);
wpr = 1 - wpin * wpin - wpin * wpin;
/* cos(theta*2) */
theta *= .5;
wr = 1;
wi = 0;
for(m = 0; m < mmax; m += 2)
{ j = m + mmax;
  tempr = (double) wr *(data1r = data[j]);
  tempi = (double) wi *(data1i = data[j + 1]);
  for(i = m; i < n - mmax * 2; i += mmax * 2)
  { /* mixed precision not significantly more
    * accurate here; if removing double casts,
    * tempr and tempi should be double */
    tempr -= tempi;
    tempi = (double) wr *data1i + (double) wi *data1r;
    /* don't expect compiler to analyze j > i+1 */
    data1r = data[j + mmax * 2];
    data1i = data[j + mmax * 2 + 1];
    data[i] = (datar = data[i]) + tempr;
    data[i + 1] = (datai = data[i + 1]) + tempi;
    data[j] = datar - tempr;
    data[j + 1] = datai - tempi;
    tempr = (double) wr *data1r;
    tempi = (double) wi *data1i;
    j += mmax * 2;
  }
  tempr -= tempi;
  tempi = (double) wr *data1i + (double) wi *data1r;
  data[i] = (datar = data[i]) + tempr;
  data[i + 1] = (datai = data[i + 1]) + tempi;
  data[j] = datar - tempr;
  data[j + 1] = datai - tempi;
  wr = (wtemp = wr) * wpr - wi * wpi;
  wi = wtemp * wpi + wi * wpr;
}
}
}

```

### Further reading

- *FFT algorithms*. [36, 52] give an overview of FFT algorithms. [5] uses the Kronecker product formalism to describe many different FFT algorithms, including parallel and vector algorithms. [53] uses the Kronecker formalism to parallelize and vectorize FFT algorithms.
- *FFTW*. FFTW can be downloaded at [11]. The latest version, FFTW3, is described in [10]. The previous version FFTW2 is described in [9] and the codelet generator *genfft* in [23].
- *SPIRAL*. Spiral is a program generation system for transforms. The core system is described in [7] and on the web at [8]. Using Kronecker product manipulations,

SIMD vectorization is described in [24, 54], shared memory (SMP and multicore) parallelization in [25], and message passing (MPI) in [55].

- *Open source FFT libraries.* FFTPACK [56] is a mixed-radix Fortran FFT library. The GNU Scientific library (GSL) [57] contains a C port of FFTPACK. UHFFT [58, 59] is an adaptive FFT library. Numerical Recipes [6] contains the radix-2 FFT implementation shown above. FFTE [60] provides a parallel FFT library for distributed memory machines.
- *Proprietary FFT libraries.* The AMD Core Math Library (ACML) [61] is the vendor library for AMD processors. Intel provides fast FFT implementations as a part of their Math Kernel Library (MKL) [62] and Integrated Performance Primitives (IPP) [63]. IBM's IBM Engineering and Scientific Software Library (ESSL) [64] and the parallel version (PESSL) contain FFT functions optimized for IBM machines. The vDSP library contains FFT functions optimized for AltiVec. The libraries of the Numerical Algorithms Group (NAG) [65] and the International Mathematical and Statistical Library (IMSL) [66] also contain FFT functionality.

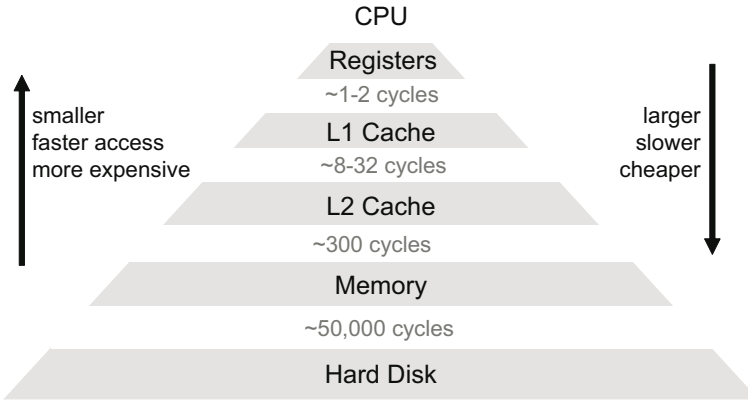
## 2.4 State-of-the-Art Desktop and Laptop Computer Systems

Modern computers include several performance enhancing microarchitectural features like cache systems, a memory hierarchy, virtual memory, and CPU features like vector and parallel processing. While these features usually increase the achievable performance, they also make the optimization process more complex. This section introduces several microarchitectural features relevant to writing fast code. For further reading, refer to [37, 38].

**Memory hierarchy.** Most computer systems use a *memory hierarchy* to bridge the speed gap between the processor(s) and its connection to main memory. As shown in Fig. 5 the highest levels of the memory hierarchy contain the fastest and the smallest memory systems, and vice versa.

A hierarchical memory enables the processor to take advantage of the memory locality of computer programs. Optimizing numerical programs for the memory hierarchy is one of the most fundamental approaches to producing fast code, and the subject of this tutorial. Programs typically exhibit temporal and spatial memory locality. Temporal locality means that a memory location that is referenced by a program will likely be referenced again in the near future. Spatial locality means that the likelihood of referencing a memory location by a program is higher if a nearby location was recently referenced. High performance computer software must be designed so that the hardware can easily take advantage of locality. Thus, this tutorial focuses on writing fast code by designing programs to exhibit maximal temporal and spatial localities.

**Registers.** Registers inside the processor are the highest level of the memory hierarchy. Any value (address or data) that is involved in computation has to eventually be placed into a register. Registers may be designed to hold only a specific type of value (special purpose registers), or only floating point values (e.g., double FP registers), vector values (vector registers), or any value (general purpose registers). The number of registers in a processor varies by architecture. A few examples are provided in Table 1. When an active computation requires more values to be held than the register space will allow,



**Fig. 5.** Memory hierarchy. Typical latencies for data transfers from the CPU to each of the levels are shown. The numbers shown here are only an indication, and the actual numbers will depend on the exact architecture under consideration.

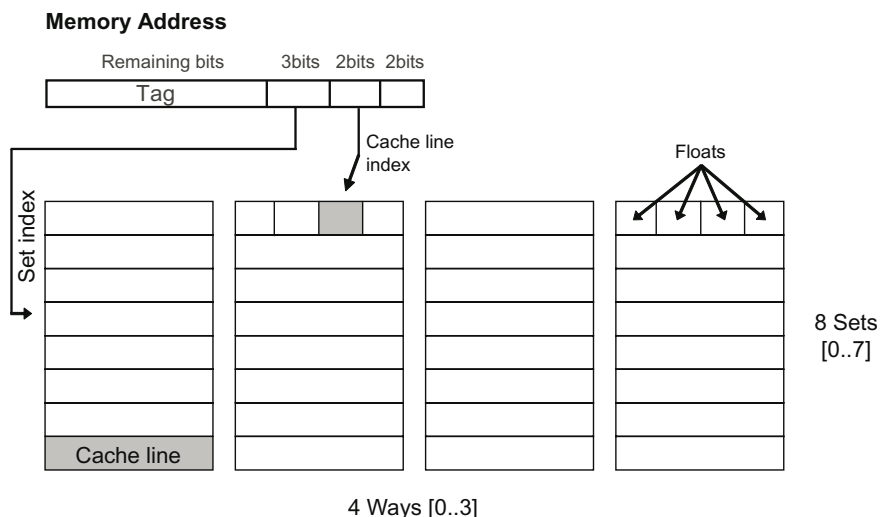
**Table 1.** Sample scalar register space (per core) in various architectures. In addition to integer and FP registers, the Core2 Extreme also has 16 multimedia registers.

Processor	Integer Registers	Double FP Registers
Core2 Extreme	16	16
Itanium 2	128	128
UltraSPARC T2	32	32
POWER6	32	32

a *register spill* occurs, and the register contents are written to lower levels of memory from which they will be reloaded again. Register spills are expensive. To avoid them and speed up computation, a processor might make use of internal registers that are not visible to the programmer. Many optimizations that work on other levels of the memory hierarchy can typically also be extended to the register level.

**Cache memory.** Cache memory is a small, fast memory that resides between the main memory and the processor. It reduces average memory access times by taking advantage of spatial and temporal locality. When the processor initially requests data from a memory location (called a cache miss), the cache fetches and stores the requested data and data spatially close. Subsequent accesses, called *hits*, can be serviced by the cache without needing to access main memory. A well designed cache system has a low miss to hit ratio (also known as just the miss ratio or miss rate).

Caches, as shown in Fig. 6 are divided into cache lines (also known as blocks) and sets. Data is moved in and out of cache memory in chunks equal to the line size. Cache lines exist to take advantage of spatial locality. Multiple levels of caches and separate data and instruction caches may exist, as shown in Table 2. Caches may be direct mapped (every main memory location is mapped to a specific cache location) or *k*-way set associative (every main memory location can be mapped to precisely *k* possible



**Fig. 6.** 4-way set associative cache with cache line size of 4 single precision floats (4 bytes per float) per line, and cache size of 128 floats (total cache size is 512 bytes). The figure also illustrates the parts of a memory address used to index into the cache. Since each data element under consideration is 4 bytes long, the two least significant bits are irrelevant in this case. The number of bits used to address into the cache line would be different for double precision floats.

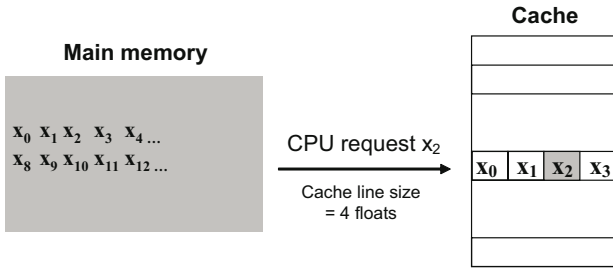
**Table 2.** Cache system example: Intel Core2 Duo, Merom Notebook processor

Level/Type	Size	Associativity
L1 Data (per core)	32 KB	8-way set associative
L1 Instruction (per core)	32 KB	8-way set associative
L2 Unified (common)	4 MB	8-way set associative

cache locations). In addition to misses caused due to data being brought in for the first time (compulsory misses) and those due to cache capacity constraints (capacity misses), caches that are not fully associative can incur conflict misses [67].

Since cache misses are typically expensive, writing fast code involves designing programs to have low miss rates. This is accomplished using two important guiding principles, illustrated in Fig. 7 and described below:

- **Reuse: Temporal locality.** Once data is brought into the cache, the program should reuse it as much as possible before it gets evicted. In other words, programs must try to avoid scattering computations made on a particular data location throughout the execution of the program. Otherwise, the same data (or data location) has to go through several cycles of being brought into the cache and subsequently evicted, which increases runtime.
- **Neighbor use (using all data brought in): Spatial locality.** Data is always brought into the cache in chunks the size of a cache line. This is seen in Fig. 7 where



**Fig. 7.** Neighbor use and reuse: When the CPU requests  $x_2$ ,  $x_0$ ,  $x_1$ , and  $x_3$  are also brought into the cache since the cache line size holds 4 floats

one data element  $x_2$  was requested, and three others are also brought in since they belong to the same cache line. To take advantage of this, programs must be designed to perform computations on neighboring data (physically close in memory) before the line is evicted. This might involve reordering loops, for instance, to work on data in small chunks.

These two principles work at multiple levels. For instance, code can be designed to use and reuse all data within a single cache block, as well as within an entire cache level. In fact, these principles hold throughout the memory hierarchy, and thus can be used at various cache and memory levels. Depending on the computation being performed, techniques that use these principles may not be trivial to design or implement.

In scientific or numerical computing, data typically consists of floating point numbers. Therefore, it helps to view the cache organization, lines, and sets in terms of the number of floating point numbers that can be held. For instance, the cache shown in Fig. 6 is a 512 byte, 4-way set associative cache with a line size of 16 bytes. There are a total of 32 lines (512 bytes / 16 bytes per line), and 8 sets (32 lines / 4 lines per set). If we note that each cache line can hold 4 floats (16 bytes / 4 bytes per float), we can immediately see that the cache can hold a total of 128 floats. This means that datasets larger than 128 floats will not fit in the cache. Also, if we make an initial access to 128 consecutive floats, there will be a total of 32 cache misses and 96 cache hits (since 4 floats in a line are loaded on each cache miss). This gives us a rough estimate of the runtime of such a set of accesses, which is useful both in designing programs and in performing sanity checks.

**Cache analysis.** We now consider three examples of accessing an array in various sequences, and analyze their effects on the cache.

Consider a simple direct mapped 16 byte data cache with two cache lines, each of size 8 bytes (two floats per line). Consider the following code sequence, in which the array  $X$  is cache-aligned (that is,  $X[0]$  is always loaded into the beginning of the first cache line) and accessed twice in consecutive order:

```
float X[8];
for(int j=0; j<2; j++)
    for(int i=0; i<8; i++)
        access(X[i]);
```

Example 1: Sequential access

line0	X0 m	X1 h	X0	X1	X4 m	X5 h	X4	X5	X0 m	X1 h	X0	X1	X4 m	X5 h	X4	X5
line1			X2 m	X3 h	X2	X3	X6 m	X7 h	X6	X7	X2 m	X3 h	X2	X3	X6 m	X7 h

Example 2: Strided access

line0	X0 m	X1	X4 m	X5	X0	X1 m	X4	X5 m	X0 m	X1	X4 m	X5	X0	X1 m	X4	X5 m
line1	X2 m	X3	X6 m	X7	X2	X3 m	X6	X7 m	X2 m	X3	X6 m	X7	X2	X3 m	X6	X7 m

Example 3: Blocked access

line0	X0 m	X1 h	X0	X1	X0 h	X1 h	X0	X1	X4 m	X5 h	X4	X5	X4 h	X5 h	X4	X5
line1			X2 m	X3 h	X2	X3	X2 h	X3 h	X2	X3	X6 m	X7 h	X6	X7	X6 h	X7 h

**Fig. 8.** Cache access analysis: The state of the complete cache for each example is shown after every two accesses, along with whether the two accesses resulted in hits or misses (shown by h or m). The two requests just made are shown in black, while the remaining parts of the cache are shown in gray. To save space, square brackets are not shown: X0 refers to X[0].

The top row on Fig. 8 shows the states of the cache after every two (out of the total of sixteen) accesses for this example. To analyze the cache footprint and pattern of this code sequence, we first observe that the size of the array (8 floats) exceeds the size of the cache (4 floats). We then observe that a total of 16 accesses are made to the array. To calculate how many result in hits, and how many in misses, we observe the cache access pattern of the code. The pattern is “0123456701234567” (only the indices of X accessed are shown). We note that an access to any even index of X results in that element and the subsequent element being loaded since they are in the same cache line. Thus, accessing X[0] loads X[0] and X[1] into the cache. We can then compute the hit/miss pattern to be: “MHMHMHMHMHMHMHMH”. So in all, there are 8 hits and 8 misses.

We now look at another code sequence that again accesses the same array twice (similar to the last example), albeit with a stride of 2:

```
float X[8];
for(int j=0; j<2; j++)
{ for(int i=0; i<7; i+=2)
  access(X[i]);
  for(int i=1; i<8; i+=2)
    access(X[i]);
}
```

The middle row on Fig. 8 shows the corresponding cache states for this example. The access pattern here is “0246135702461357”. A similar analysis shows us that the miss ratio is even worse: every single access in this pattern results in a miss (with a total of 16 misses and 0 hits). This example illustrates an important point: strided accesses generally result in poor cache efficiency, since they effectively “make the cache smaller.”

Finally, let us consider a third code sequence that again accesses the same array twice:

```
float X[8];
for(i=0; i<2; i++)
  for(k=0; k<2; k++)
    for(j=0; j<4; j++)
      access(X[j+(i*4)]);
```

The bottom row on Fig. 8 shows the corresponding cache states for this example. The access pattern here is “0123012345674567”. Counting the hits and misses, (“MHMH-HHHMHMHHHHH”), we observe that there are 12 hits and 4 misses. We also note that if this rearrangement is legal, it is a cache optimized version of the original code sequence. In fact, this rearrangement is an example of both of the previously mentioned principles behind optimizing for the memory hierarchy: reuse and neighbor use. Unlike the first example, the “0123” block is reused here before being evicted. Unlike the second example, every time an even-indexed element is accessed, the succeeding odd-indexed element which is a part of the same cache line is also immediately accessed. Thus, analyzing the cache can help us estimate and improve the cache performance of a program.

**CPU features.** Modern microprocessors also contain other performance enhancing features. Most processors contain pipelined superscalar out-of-order cores with multiple execution units. Pipelining is a form of parallelism where different parts of the processor work simultaneously on different components of different instructions. Superscalar cores can retire more than one instruction per processor clock cycle. Out-of-order processing cores can detect instruction dependencies and reschedule the instruction sequence for performance. The programmer has to be cognizant of these features in order to be able to optimize for a particular architecture.

Most such aggressive cores also contain multiple execution units (for instance, floating point units) for increased performance. This means that a processor might be able to, for instance, simultaneously retire one floating point add instruction every cycle, and one floating point multiplication instruction every other cycle. It is up to the programmer and the compiler to keep the processor’s execution units adequately busy (primarily via instruction scheduling and memory locality) in order to achieve maximum performance.

The theoretical rate at which a processor can perform floating point operations is known as the processor’s *theoretical peak performance*. This is measured in flop/s (Floating point Operations per Second). For instance, a processor running at 1 GHz that can retire one addition every cycle, and one multiplication every other cycle has a theoretical peak of 1.5 Gflop/s. The theoretical peak of a Core2 Extreme processor operating under various modes is shown in Table 3.

In practice, cache misses, pipeline stalls due to dependencies, branches, branch mispredictions, and the fact that meaningful programs contain instructions other than floating point instructions, do not allow a processor to perform at its theoretical peak performance. Further, the achievable performance also depends on the inherent limitations of the algorithm, such as reuse. For example, MMM, with a reuse degree of  $O(n)$



**Table 3.** Core2 Extreme: Peak performance (in Gflop/s) for a 3 GHz Core2 Extreme processor in various operation modes

	1 core	2 cores	4 cores
x87 double	6	12	24
SSE2 double	12	24	48
x87 float	6	12	24
SSE float	24	48	96

can achieve close to the peak performance of 48 Gflop/s (as seen in Fig. 3), whereas the DFT with a reuse degree of  $O(\log(n))$  reaches only about 50% (as seen in Fig. 2).

In summary, knowing a processor’s theoretical peak and an algorithm’s degree of reuse gives us a rough estimate of the extent to which a program could potentially be improved.

Modern processors also contain two major explicit forms of parallelism: vector processing and multicore processing, which are important for writing fast code, but beyond the scope of this tutorial.

### Further reading

- *General computer architecture.* [37, 38].
- *CPU/architecture specific.* [68, 69].

## 2.5 Using Compilers

To produce fast code it is not sufficient to write and optimize source code—the programmer must also ensure that the code that is written gets compiled into an efficient binary executable. This involves the careful selection and use of compiler flags, use of language extensions, and monitoring and analyzing the compiler’s output. Furthermore, in some situations, it is best to let the compiler know of all the degrees of freedom it has, so it can optimize well. In other situations, it is best to direct the compiler to do exactly what is required. This section goes over the compile process, what to keep in mind before, while, and after compiling, and some of the common pitfalls related to the compiling process.

**Variable declaration: Memory allocation.** Understanding how C handles the allocation of space for variables is beneficial. C assigns variables to different *storage class specifiers* by default, based on where in the source code they appear. The default storage class for a variable can be overridden by preceding a variable declaration with the desired storage class specifier.

Variables that are shared among source files use the `extern` storage class. Global variables belong to the `static` storage class, and typically exist in static memory (as do `extern` variables), which means that they exist as long as the program executes. Local variables belong to the `auto` (automatic) storage class, which means that they are allocated on the stack automatically upon entering the local block within which they

are defined, and destroyed upon exit. The `register` storage class requests that the compiler allocates space for the variable directly in the CPU registers. These are useful to eliminate load/store latencies on heavily used variables. Keep in mind that depending on the compiler being used, care should be taken to initialize variables before usage.

**Variable declaration: Qualifiers.** Most compilers provide further means to specify variable attributes through *qualifiers*. A `const` qualifier specifies that a variable's value will never change. A `volatile` qualifier is used to refer to variables whose values might be influenced by sources external to the compiler's knowledge. Operations involving volatile variables will not be optimized by the compiler, in order to preserve correctness. A `restrict` qualifier is especially useful to writing fast code, since it tells the compiler that a certain memory address will be restricted to access via the specified pointer. This allows for effective compiler optimization.

Finally, memory alignment can also be specified by qualifiers. Such qualifiers are specific to the compiler being used. For instance, `__attribute__((aligned(128)))` requests a variable to be aligned at the specified 128-byte memory boundary. Such requests allow variables to be aligned to cache line boundaries or virtual memory pages as desired. Similar qualifiers can be used to tell the compiler that the address pointed to by a pointer is memory aligned.

**Dynamic memory allocation.** Dynamic memory allocation, using `malloc` for example, involves allocating memory in the *heap*, and returning a pointer to the allocated memory. If alignment is of importance, many libraries provide a `memalign` function (the Intel equivalent is `mm_malloc`) to allocate memory aligned to a specified boundary. The alternative is to allocate more memory than required, and to then check and shift the returned pointer adequately to achieve the required alignment.

**Inline assembly and intrinsics.** Sometimes, it is best to write assembly code to access powerful features of the machine which may not be available via C. Assembly can be included as a part of any program in C using inline assembly. However, inline assembly use must be minimized as it might interfere with compiler optimizations. Architecture vendors typically provide C language extensions to allow programmers to access special machine instructions. These extensions, called *intrinsics*, are similar to function calls that allow the programmer to avoid writing inline assembly. Importantly, intrinsics allow the compiler to understand what data and/or control the programmer is manipulating, thus allowing for better optimization. As an example, Intel's MMX and SSE extensions to the x86 ISA can be accessed via C intrinsics provided by Intel.

**Compiler flags.** Most compilers are highly configurable via a plethora of command line options and flags. In fact, finding the right set of compiler options that yield optimal performance is non-trivial. However, there are some basic ideas to keep in mind while using a compiler, as listed below. Note that these ideas apply to most compilers.

- *C standards.* A compiler can be set to follow a certain C standard such as C99. Certain qualifiers and libraries might need specific C standards to work. By switching to a newer standard, the programmer can typically communicate more to the compiler, thus enabling it to work better.

- *Architecture specifications.* Most compilers will compile and optimize by default for a basic ISA standard to maximize compatibility. Machine and architecture specific optimizations may not be performed as a result. For instance, a compiler on an AMD Athlon processor may compile to the x86 standard by default, and not perform Athlon-specific optimizations. Instructing the compiler to compile for the correct target architecture may result in considerable performance gains. Additional flags may be required for these optimizations. For example, gcc requires the “-sse” flag to include vector instructions.
- *Optimization levels.* Most compilers usually define several optimization levels that can be selected. Determining the optimization level that yields maximum performance is a black art usually done by trial and error. A more aggressive optimization level doesn’t necessarily yield better performance. Optimization levels are usually a shortcut to turn on or off a large set of compiler flags (discussed next).
- *Specialized compiler options.* Compilers typically perform numerous optimizations, many which can be selectively turned on or off and configured through command line flags. Loop unrolling, function inlining, instruction scheduling, and other loop optimizations are only some of the available configurable optimizations. Usually, finding the right optimization level is sufficient, but sometimes, inspection of assembly code provides insights that can be used to fine-tune compiler optimizations.

**Compiler output.** The output of the compiler is usually an executable binary. As mentioned earlier, the compiler can also be used to produce various intermediate stages, including the preprocessed source, assembly code, and the object code. Sometimes, it is important and useful to visually inspect the assembly code to better understand both the performance of an executable and the behavior of the compiler.

Compilers also output warnings, which can be controlled through compiler flags. Sometimes, a seemingly innocuous warning might provide excellent insights into the source of a bug, which makes these warnings a significant debugging tool.

Optimization reports are an important part of the compiler output that must be inspected. For instance, a vectorizing compiler will inform the programmer of whether it was able to successfully vectorize or not. A failure to vectorize a program that was expected to be vectorized is a reason for examining the program carefully, and modifying or annotating the code as appropriate.

In conclusion, it is important for programmers to be knowledgeable about the compiler that they use in order to be able to use the compiler efficiently, and to ensure that poor compiler usage does not diminish the results of code designed for high performance.

### Further reading

- *Gnu compiler collection (gcc).* [70].
- *Intel compiler.* [71].

## 2.6 Exercises

1. **Direct implementations.** Implement, execute, and verify:

- A direct implementation of MMM (code snippet given in Section 2.2),
- The Numerical Recipes code for the DFT as given in [6],

This code will also be used in the exercises of later sections.

2. **Determining hardware information.** In this exercise, you will determine the relevant hardware configuration of your computer. You will use this information in later exercises.

Determine the following information about your computer:

- CPU type and clock speed
- For each cache: size, associativity, and cache line size
- Size of main memory
- System bus speed

Here are a few tips on how to determine this information:

- Look in the computer’s manual.
- Look in the CPU manufacturer’s manual.
- To obtain CPU information in Linux, execute `cat /proc/cpuinfo`.
- To obtain cache information in Linux, search for lines with the word “cache” in the kernel ring buffer. You can do so by typing: `dmesg | grep '^CPU.*cache'` on most systems.

3. **Loop optimization for the cache.** Consider a 2-way set associative cache with a cache size of 32KB, a cache line size of 32B, and a FIFO (First In, First Out) replacement policy (this means if one of the two candidate cache lines has to be replaced, it will be the one that was first brought into the cache). Consider two single-precision floating point arrays (single precision float = 4B),  $A$  and  $B$  with  $n$  elements, where  $n$  is much larger than the cache and is a multiple of the cache size. Further, assume that  $A$  and  $B$  are both fully cache-aligned, i.e.,  $A[0]$  and  $B[0]$  map to the first position in the first cache line.

Now consider the following pseudo code snippet:

```
for(i from 0 to n-1)
  A[i] = A[i] + B[f(i)]
```

where  $f(i)$  is an index mapping function that reads  $B$  at a stride of 8. (If for example,  $B$  was 16 elements long, then reading it at stride 8 would result in this access pattern:  $f(i) = 0, 8, 1, 9, 2, 10, 3, 11, 4, 12, 5, 13, 6, 14, 7, 15$ ).

Assume an empty cache for each part of this exercise.

- (a) (Disregard the code snippet for this part) What is the expected number of cache misses incurred by streaming once completely through array  $A$  alone in sequential order, given the cache parameters above?
- (b) (Disregard the code snippet for this part) What is the expected number of cache misses incurred by streaming once completely through array  $B$  alone at stride of 8 given the cache parameters above?
- (c) How many cache misses is the given pseudo code snippet expected to incur? (Assume, for simplicity, that index variables are not cached).
- (d) Rewrite the code (without changing the semantics, i.e., overall computation) to reduce the number of cache misses as much as possible. (Assume, for simplicity, that index variables are not cached).

### 3 Performance Optimization: The Basics

In this section we will review the basic steps required to assess the performance of a given implementation, also known as “benchmarking.” We focus on runtime benchmarking as the most important case. (Other examples of benchmarking includes assessing the usage of memory or other resources.)

For a given program, the basic procedure consists of three steps:

1. Finding the hotspots (hotspots are the most frequently executed code regions),
2. Timing the hotspots, and
3. Analyzing the measured runtimes.

It is essential to find the parts of the program that perform the bulk of the computation and restrict further investigation to these *hotspots*. Optimizing other parts of the program will have little to no effect on the overall runtime. In order to obtain a meaningful runtime measurement, one has to build a test environment for each hotspot that exercises and measures it in the correct way. Finally, one has to assess the measured data and relate it to the cost analysis of the respective hotspot. This way one can make efficiency statements and target the correct (inefficient) hotspot for further optimization.

#### 3.1 Finding the Hotspots

The first step in benchmarking is to find the parts of the program where most time is spent. Most development platforms contain a *profiling* tool. For instance, the development environment available on the GNU/Linux platform contains the GNU gprof profiler. On Windows platforms, the Intel VTune tool [72] that plugs into Microsoft’s Visual Studio [73] can be used to profile applications.

If no profiling tool is available, obtain first-order profiling information can be obtained by inserting statements throughout the program that print out the current system time. In this case, less is more, as inserting too many time points may have side effects on the measured program.

**Example: GNU tool chain.** We provide a small example of using the GNU tool chain to profile a sample program.

Consider the following program:

```
#include <stdio.h>

float function1()
{ int i; float retval=0;
  for(i=1; i<1000000; i++)
    retval += (1/i);
  return(retval);
}

float function2()
{ int i; float retval=0;
  for(i=1; i<10000000; i++)
    retval += (1/(i+1));
```

```

    return(retval);
}

void function3() { return; }

int main()
{ int i;
  printf("Result: %.2f\n", function1());
  printf("Result: %.2f\n", function2());
  if (1==2) function3();
  return(0);
}

```

Our final objective is to optimize this program. In order to do so, we first need to find where the program spends most of its execution time, using `gprof`.

As specified in the `gprof` manual [74], three steps are involved in profiling using `gprof`:

1. Compile and link with profiling enabled:

```
gcc -O0 -lm -g -pg -o ourProgram ourProgram.c
```

The resulting executable is instrumented. This means that in addition to executing your program, it will also write out profiling data when executed. (Note: We use the `-O0` flag to prevent the compiler from inlining our functions and performing other optimizing transforms that might make it difficult for us to make sense of the profile output. For profiling to provide us with meaningful information, we would need to compile at the level of optimization that we intend to finally use, with the understanding that mapping the profiler's output back to the source code in this case might involve some effort.)

2. Execute the program to generate the profile data file

```
./ourProgram
```

The program executes and writes the profile data to `gmon.out`.

3. Run `gprof` on the profile data file to analyze the profile data

```
gprof ourProgram gmon.out > profile.txt
```

The analysis is now contained in `profile.txt`. This file shows you how many times each function was executed, and how much time was spent in each function, and plenty of other detail. For our example program, we obtain:

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
92.68	0.38	0.38	1	380.00	380.00	function2
7.32	0.41	0.03	1	30.00	30.00	function1

We can see that most of the program runtime was spent in executing `function2`, with relatively little spent on `function1`. This tells us that it is most important to optimize the runtime of `function2`.

Further down in `profile.txt`, we see that `gprof` also tells us if the time taken by a function was spent inside the function or inside other function calls made by the function. Note that `gprof` can take several other arguments to produce different kinds of profiling analyses for the executable, including the number of times a certain line in the source code was executed.

### 3.2 Timing a Hotspot

Once the hotspots have been found, we need to measure their runtime for further analysis. Each hotspot must be timed separately with an appropriate timing routine. The general idea is the following:

1. Read the current time (start time) from the appropriate time source.
2. Execute the kernel/hotspot. Iterate an adequate number of times to obtain a meaningful value off the time source.
3. Read the current time (end time) from the appropriate time source.
4. Execution time of the kernel/hotspot =  $\frac{\text{End time} - \text{Start time}}{\text{Number of iterations}}$ .

We first discuss time sources, and reading the time from them; then we explain how to write a timing routine to get meaningful results.

**Time functions.** Depending on the system one is using, a variety of time sources to “get the current time” may be available:

- Most Unix systems define `gettimeofday()` to portably query the current time (as defined in IEEE Std 1003.1).
- ANSI C defines `ctime()` and `clock()` as portable ways of obtaining the current time.
- On Intel processors, the `rdtsc` instruction reads the time stamp counter which allows near-cycle accurate timing. On PowerPC processors, the `mfspr` instruction reads the time-base register.

Generally, portable time functions have much less precision than cycle-counter-based methods. The pros and cons of various timing methods are listed below:

Timer type	Advantages	Disadvantages
Wall clock; Unix: <code>gettimeofday()</code>	Simple to use, highly portable	Low resolution, does not account for background tasks
System timer; Unix: <code>time</code> command	Gives wall clock, user-cpu, and system-cpu times	Relatively low resolution
Hardware timestamp counter (discussed below)	High resolution, most precise and accurate	Does not account for background system load (effectively, wall clock time), best for kernels with short runtimes; non-portable

We give a simplified example of a timing macro based on `rdtsc` (a hardware times-tamp counter) for a 32-bit Intel processor to be used with Microsoft VisualStudio:

```
typedef union
{ __int64 int64;
  struct {__int32 lo, hi;} int32;
} tsc_counter;

#define RDTSC(cpu_c)          \
{ __asm rdtsc                  \
  __asm mov (cpu_c).int32.lo, eax \
  __asm mov (cpu_c).int32.hi, edx \
}
```

The corresponding code sequence in GNU C looks slightly different:

```
typedef union
{ unsigned long long int64;
  struct {unsigned int lo, hi;} int32;
} tsc_counter;

#define RDTSC(cpu_c)          \
  __asm__ __volatile__ ("rdtsc" : \
  "=a" ((cpu_c).int32.lo), \
  "=d" ((cpu_c).int32.hi))
```

**Timing routine.** A timing routine calls the function that is to be timed without executing the original program. The objective is to isolate the kernel and measure the runtime of the kernel with the least disturbance and highest accuracy possible. A timing routine consists of the following steps:

- Initialize kernel-related data structures.
- Initialize kernel input data.
- Call kernel a few times to put microarchitectural components into steady state.
- Read current time.
- Call kernel multiple times to obtain an adequately precise value from the timing source used.
- Read current time.
- Divide the time difference by the number of kernel calls.

To obtain more stable timing results, one often has to run multiple timings and take the average or minimum value.

We give an example timing routine for an MMM function computing  $C = C + AB$ , assuming all matrices are square  $N \times N$ . The RDTSC macro is defined above.

```
double time_MMM(int N, double *A, double *B, double *C)
{ // init C
  for(i=0; i<N; i++)
    C[i] = 0.0;

  // put microarchitecture in steady state
  MMM(A,B,C);
```



```
// time
RDTSC(t0);
for(int i=0; i<TIMING_REPETITIONS; i++)
    MMM(A,B,C);
RDTSC(t1);

// compute runtime in cycles
return (double)((t1.int64-t0.int64)/TIMING_REPETITIONS);
}
```

**Known problems.** The following problems may occur when timing numerical kernels:

- Too few iterations of the function to be timed are executed between the two time stamp readings, and the resulting timing is inaccurate due to poor timer resolution.
- Too many iterations are executed between the two time stamp readings, and the resulting timing is affected by system events.
- The machine is under load and the load has side effects on the measured program.
- Multiple timing jobs are executed concurrently, and they interfere with one another.
- Data alignment of input and output triggers cache problems.
- Virtual-to-physical memory translation makes timing irreproducible.
- The time stamp counter overflows and either triggers an interrupt or produces a meaningless value.
- Reading the timestamp counters requires hundred(s) of cycles, which itself affects the timing.
- The linking order of object files changes locality of static constants and this produces cache interference.
- The machine was not rebooted in a long time and the operating system state causes problems.
- The control flow in the numerical kernel being timed is data-dependent and the test data is not representative.
- The kernel is in-place (e.g., the input is a vector  $x$  and the output is written back to  $x$ ), and the norm of the output is larger than the norm of the input. Repetitive application of the kernel leads to an exponential growth of the norm and finally triggers floating-point exceptions which interfere with the timing.
- The transform is timed with a zero vector, and the operating system is “smart,” and responds to a request for a large zero-vector dynamic memory allocation by returning a special zero-valued copy-on-write virtual memory page. Read accesses to this “page” would be much faster than accesses to a page that is actually allocated, since this page is a special one maintained by the operating system for efficiency.

One needs to be very careful when timing numerical kernels to rule out these problems. Getting highly accurate, reproducible, stable timing results for the full range of problem sizes is often nontrivial. Note that small problem sizes may suffer from timer resolution issues, while large problem sizes with longer runtimes may suffer from the effects of intervening processes.

### 3.3 Analyzing the Measured Runtime

We now know how to calculate the theoretical peak performance and the memory bandwidth for our target platform, and how to obtain the operations count and the runtime for our numerical kernel. The next step is to use these to conduct a performance analysis that answers two questions:

- What is the limiting resource, i.e., is the kernel CPU-bound or memory-bound? This provides an idea of the various optimization methods that can be used.
- How efficient is the implementation with respect to the limiting resource? This shows the potential performance increase we can expect through optimization.

**Normalization.** To assess the runtime behavior of a kernel as function of the problem size, the runtime (or inverse runtime) has to be normalized with the asymptotic or exact operations count. For instance, FFT performance is usually reported in pseudo Mflop/s. This value is computed as  $5n \log_2(n)/\text{runtime}$  for  $\text{DFT}_n$ ;  $5n \log_2(n)$  is the operations count of the radix-2 FFT. For MMM, the situation is easier, since all currently relevant implementations have the exact operations count  $2n^3$ .

Let us now take a look at at Fig. 2. The Numerical Recipes FFT program achieves almost the same pseudo Mflop/s value, independently of the problem size. This means that all problem sizes run approximately at the same level of (in)efficiency. In contrast, the best code shows a wide variation of performance, generally at a much higher pseudo Mflop/s level. In particular, the performance ramps up to 25 Gflop/s and then drops dramatically. This means, the DFT becomes more and more efficient with larger problems, but only up to a certain size. Analysis shows that the breakdown occurs once the whole working set of the computation does not fit into the L2 cache any more and the problem switches from being CPU-bound to memory-bound, since the DFT's reuse is only  $O(\log(n))$ .

In contrast, Fig. 3 shows that MMM maintains the performance even for out-of-cache sizes. This is possible since MMM has a reuse of  $O(n)$ , higher than the DFT.

Fig. 2 shows that performance plots for high-performance implementations can feature unanticipated characteristics. That is especially true if the kernel changes behavior, for instance, if it slowly changes from being CPU-bound to memory-bound as the kernel size is varied.

**Relative performance.** Absolute performance only tells a part of the story. Comparing the measured performance to the theoretical peak performance shows how efficient the implementation is. A low efficiency for an algorithm with sufficiently high reuse means there is room for optimization.

We continue examining our examples from Fig. 2 and Fig. 3 with the target machine being a Core2 Extreme at 3 GHz.

In Fig. 2 Numerical Recipes is a single-core single-precision x87 implementation and thus the corresponding peak performance is 6 Gflop/s (see Table 3). As Numerical Recipes reaches around 1 pseudo Gflop/s it runs at about 16% of the peak. Note that if SSE (4-way vector) instructions and all four cores are used, the peak performance goes up by a factor of 16. (see Table 3). The best scalar code achieves around 4 Gflop/s or about 60% of the x87 peak. The fastest overall code uses SSE and 4 cores and reaches up to 25 Gflop/s or 25% of the quad-core SSE peak.

In Fig. 2 the overall fastest code reaches and sustains about 42 Gflop/s or about 85% of the quad-core SSE2 peak. This is much higher than the DFT and also due to the higher degree of reuse in MMM compared to the DFT.

### 3.4 Exercises

1. **Performance analysis.** In this exercise, you will measure and analyze the performance of the naive implementations of MMM and the DFT from Exercise 1 in Section 2. The steps you will need to follow to complete this exercise are given below. For this exercise, use the hardware configuration of your computer as you determined in Exercise 2 on page 216.

- (a) *Determine your computer's theoretical peak performance.* The theoretical peak performance is the number of floating point operations that can be done in a second. This is found by determining the CPU clock speed, and examining the microarchitecture to look at the throughput of floating point operations. For instance, a CPU running at 900 MHz that can retire 2 floating point operations per cycle, has a theoretical peak performance of 1800 Mflop/s. If the type of instructions that the CPU can retire at the same rate includes FMA (fused multiply add) instructions, the theoretical peak would be 3600 Mflop/s (2 multiplies and 2 adds per cycle = 4 operations per cycle). For this exercise, do not consider vector operations.
- (b) *Measure runtimes.* Use your implementations of the MMM and DFT as completed in Exercise 1 on page 215. Use the techniques described in Section 3.2 to measure the runtimes of your implementations using at least two different timers.
- (c) *Determine performance and interpret results.*
  - Performance: The performance of your implementation is its number of floating point operations per unit time, measured in flop/s. For the DFT, the number of operations should be assumed  $5n \log(n)$ .
  - Percentage peak performance: This is simply the percentage of theoretical peak performance. For instance, if your measured code runs at 1.2 Gflop/s on a machine with a peak performance of 3.6 Gflop/s, this implies that your implementation achieves 33.3% of peak performance.

2. **Micro-benchmarks: mathematical functions.** We assume a Pentium compatible machine. Determine the runtime (in cycles) of the following computations ( $x, y$  are doubles) as accurately as possible:

- $y = x$
- $y = 7.12x$
- $y = x + 7.12$
- $y = \sin(x), x \in \{0.0, 0.2, 4.1, 170.32\}$
- $y = \log(x), x \in \{0.001, 1.00001, 10.65, 2762.32\}$
- $y = \exp(x), x \in \{-1.234e-17, 0.101, 3.72, 1.234e25\}$

There are a total of 15 runtimes. Explain the results. The benchmark setup should be as follows:

- (a) Allocate two vector doubles  $x[N]$  and  $y[N]$  and initialize all  $x[i]$  to be one of the values from above.
- (b) Use

```
for (i=0; i<N; i++)
    y[i] = f(x[i]);
```

to compute  $y[i] = f(x[i])$ , with  $f()$  being one of the functions above and time this `for` loop.

- (c) Choose  $N$  such that all data easily fits into L1 cache but there are enough iterations to obtain a reasonable amount of work.
- (d) Use the x86 time stamp counter via the interface provided by `rdtsc.h`, as listed in Section [3.2](#)

To accurately measure these very short computations, use the following guidelines:

- Only time the actual work, leave everything else (initializations, timing related computations, etc.) outside the timing loop.
- Use the C preprocessor to produce a parameterized implementation to easily check different parameters.
- You may have to run your `for(N)` loop multiple times to obtain reasonable timing accuracy.
- You may have to take the minimum across multiple such measurements to obtain stable results. Thus, you might end up with three nested loops.
- You must put microarchitectural components into steady state before the experiment: variables where you store the timing results, the timed routine and the data vectors should all be loaded into the L1 cache, since cache misses might result in inaccurate timing results.
- Alignment of your data vectors on cache line sizes or page sizes can influence the runtime significantly.
- The use of `CPUID` to serialize the CPU before reading the RDTSC as explained in the Intel manual produces a considerable amount of overhead and may be omitted for this exercise.

## 4 Optimization for the Memory Hierarchy

In this section we describe methods for optimizations targeted at the memory hierarchy of a state-of-the-art computer system. We divide the discussion into four sections:

- Performance-conscious programming.
- Optimizations for cache.
- Optimizations for the registers and CPU.
- Parameter-based performance tuning.

We first overview the general concepts, and then apply them to MMM and the DFT later.

## 4.1 Performance-Conscious Programming

Before we discuss specific optimizations, we need to ensure that our code does not yield poor performance because it violates certain procedures fundamental to writing fast code. Such procedures are discussed in this section. It is important to note that programming for high performance may go to some extent against standard software engineering principles. This is justified if performance is critical.

**Language: C.** For high performance implementations, C is a good choice, as long as one is careful with the language features used (see below). The next typical choice for high-performance numerical code is Fortran, which tends to be more cumbersome to use than C when dynamic memory and dynamic data structures are used.

Object-oriented programming (C++) must be avoided for performance-critical parts since using object oriented features such as operator overloading and late binding incurs significant performance overhead. Languages that are not compiled to native machine code (like Java) should also be avoided.

**Arrays.** Whenever possible, one-dimensional arrays of scalar variables should be used. Assume a two-dimensional array `A` is needed whose size is not known at compile time. It is tempting to declare it as `**A` or `A[] []` but as a consequence, every access `A[i][j]` results in a sequence of two dependent pointer dereferencings or two loads. If linearized, only one dereferencing or load per access is needed (at the expensive of a simple index computation). If the size of `A` is known at compile time the compiler should perform the linearization but it is again safer to do it yourself.

**Records.** Using an abstract data type implemented as `struct` often prevents compiler optimization. Further, it may introduce implicit index computations and alignment issues that may not be handled well by the compiler. Hence, complicated `struct` and `union` data types should be avoided. For example, to represent vectors of complex numbers, vectors of real numbers of twice the size should be used, with the real and imaginary parts appearing as pairs along the vector.

**Dynamic data structures.** Dynamically generated data structures like linked lists and trees must be avoided if the algorithm using them can be implemented on array structures instead. Heap storage must be allocated in large chunks, as opposed to separate allocations for each object.

**Control flow.** Unpredictable conditional branches are computationally expensive on machines with long pipelines. Hence, `while` loops and loops with complicated termination conditions must be avoided. `for` loops with loop counters and loop bounds known at compile-time must be used whenever possible. `switch`, `?:`, and `if` statements must be avoided in hot spots and inner loops, as they may be translated into conditional branches. For small, repetitive tasks, macros are a better choice than functions. Macros are expanded before compilation while the compiler must perform analysis on inline functions.

## 4.2 Cache Optimization

For lower levels in the memory hierarchy (L1, L2, L3 data cache, TLB = translation lookaside buffer) the overarching optimization goal is to reuse data as much as possible

once brought in. The architecture of a set-associative cache (Fig. 6) suggests three major optimization methods that target different hardware restrictions.

- **Blocking:** working on data in chunks that fit into the respective cache level, to overcome restrictions due to cache capacity,
- **Loop merging:** merging consecutive loops that sweep through data into one loop to reuse data in the cache and hence make the best use of the restricted memory bandwidth, and,
- **Buffering:** copying data into contiguous temporary buffers to overcome conflict cache misses due to cache associativity.

The actual optimization process applies one or more of these ideas to some of the levels of the memory hierarchy. It is not always a good idea to apply all methods to all levels, as code complexity may increase dramatically.

Finally, the correct parameters for blocking and/or buffering on the targeted computer system have to be found. A good approach is to write the program parameterized, i.e., collect all parameters as named constants. Then it is easy to try different parameter settings by hand or using a script to find the variant that achieves the highest performance.

**Blocking.** The basic idea of blocking is to perform the computation in “blocks” that operate on a subset of the input data to achieve memory locality. This can be achieved in different ways. For example, loops in loop nests, like the triple loop MMM in Section 2.2 may be split and swapped (a transformation called tiling) so that the working set of the inner loops fits into the targeted memory hierarchy level, whereas the outer loop jumps from block to block. Another way to achieve blocking is to choose a recursive algorithm to start with. Recursive algorithms naturally divide a large problem into smaller problems that typically operate on subsets of the data. If designed and parameterized well, at some level all sub-problems fit into the targeted memory level and blocking is achieved implicitly. An example of such an algorithm is the recursive Cooley-Tukey FFT introduced later in in (3).

**Loop merging.** Numerical algorithms often have multiple stages. Each stage accesses the whole data set before the next stage can start, which produces multiple sweeps through the working set. If the working set does not fit into the cache this can dramatically reduce performance.

In some algorithms the dependencies do not require that *all* operations of a previous stage are completed before *any* operation in a later stage can be started. If this is the case, loops can be merged and the number of passes through the working set can be reduced. This optimization is essential for implementing high-performance DFT functions.

**Buffering.** When working on multi-dimensional data like matrices, logically close elements can be far from each other in linearized memory. For instance, matrix elements in one column are stored at a distance equal to the number of columns of that matrix. Cache associativity and cache line size get into conflict if one wants to hold, for instance, a small rectangular section of such a matrix in cache, leading to cache thrashing. This means the elements accessed by the kernel are mapped to the same cache locations and hence are moved in and out during computation.

One simple solution is to copy the desired block into a contiguous temporary buffer. That incurs a one-time cost but alleviates cache thrashing. This optimization is often called buffering.

### 4.3 CPU and Register Level Optimization

Optimization for the highest level in the memory hierarchy, the registers, is to some extent similar to optimizations for the cache. However it also needs to take into account microarchitectural properties of the target CPU. Current high-end CPUs are superscalar, out-of-order, deeply pipelined, feature complicated branch prediction units, and many other performance enhancing technologies. From a high-level point of view, one can summarize the optimization goals for a modern CPU as follows. An efficient C program should:

- have inner loops with adequately large loop bodies,
- have many independent operations inside an inner loop body,
- use automatic variables whenever possible,
- reuse loaded data elements to the extent possible,
- avoid math library function calls inside an inner loop if possible.

Some of these goals might conflict with others, or are constrained by machine parameters. The following methods help us achieve the stated goals:

- Blocking
- Unrolling and scheduling
- Scalar replacement
- Precomputation of constants

We now discuss these methods in detail.

**Blocking.** Register-level blocking partitions the data into chunks on which the computation can be performed within the register set. Only initial loads and final stores but no register spilling is required. Sometimes a small amount of spilling can be tolerated. We show the blocking of a single loop as example. Consider the example code below.

```
for(i=0; i<8; i++)
{ y[2*i]   = x[2*i] + x[2*i+1];
  y[2*i+1] = x[2*i] - x[2*i+1];
}
```

We block the `i` loop, obtaining the following code.

```
for(i1=0; i1<4; i1++)
  for(i2=0; i2<2; i2++)
  { y[4*i1+2*i2]   = x[4*i1+2*i2] + x[4*i1+2*i2+1];
    y[4*i1+2*i2+1] = x[4*i1+2*i2] - x[4*i1+2*i2+1];
  }
```

On many machines registers are only addressable by name but not indirectly via other registers (holding loop counters). In this case, once the data fits into registers, either

loop unrolling or software pipelining with register rotation (as supported by Itanium) is required to actually take advantage of register-blocked computation.

**Unrolling and scheduling.** Unrolling produces larger basic blocks. That allows the compiler to apply strength reduction to simplify expressions. It decreases the number of conditional branches thus decreasing potential branch mispredictions and condition evaluations. Further it increases the number of operations in the basic block and allows the compiler to better utilize the register file. However, too much unrolling may increase the code size too much and overflow the instruction cache. The following code is the code above with unrolled inner loop `i2`.

```

for (i1=0; i1<4; i1++)
{
  y[4*i1]   = x[4*i1] + x[4*i1+1];
  y[4*i1+1] = x[4*i1] - x[4*i1+1];
  y[4*i1+2] = x[4*i1+2] + x[4*i1+3];
  y[4*i1+3] = x[4*i1+2] - x[4*i1+3];
}

```

Unrolling exposes an opportunity to perform instruction scheduling. With unrolled code, it becomes easy to determine data dependencies between instructions. Issuing an instruction right after a preceding instruction that it is dependent upon will lead to the CPU pipeline being stalled until the former instruction completes. Instruction scheduling is the process of rearranging code to include independent instructions in between two dependent instructions to minimize pipeline stalls.

Scheduling large basic blocks with complicated dependencies may be too challenging for the compiler. In this case source scheduling may help. Source scheduling is the (legal) reordering of statements in the unrolled basic block. Different scheduling algorithms apply different rules, aiming at, e.g., minimizing distance between producer and consumer (which may potentially not be too short), and/or minimizing the number of live variables for each statement in the basic block. It is sometimes better to source schedule basic blocks and turn off aggressive scheduling by the compiler.

The number of registers, quality of the C compiler, and size of the instruction cache limit the amount of unrolling, that increases performance. Experiments show that on current machines, roughly 1,000 operations are the limit. Note, that unrolling always increases the size of the loop body, but not necessarily the instruction-level parallelism. Depending on the algorithm, more complicated loop transformations may be required. One example is the MMM, discussed later.

**Scalar replacement.** In C compilers, pointer analysis is complicated, and using even the simplest pointer constructs can prevent “obvious” optimizations. This observation extends to arrays with known sizes. It is very important to replace arrays that are fully inside the scope of an innermost loop by one automatic, scalar variable per array element. This can be done as the array access pattern does not depend on any loop variable and will help compiler optimization tremendously. As an example, consider the following code:

```

double t[2];
for (i=0; i<8; i++)
{
  t[0] = x[2*i] + x[2*i+1];
}

```



```

    t[1] = x[2*i] - x[2*i+1];
    y[2*i] = t[0] * D[2*i];
    y[2*i+1] = t[0] * D[2*i];
}

```

Scalarizing `t` will result in code that the compiler can better optimize:

```

double t0, t1;
for(i=0; i<8; i++)
{ t0 = x[2*i] + x[2*i+1];
  t1 = x[2*i] - x[2*i+1];
  y[2*i] = t0 * D[2*i];
  y[2*i+1] = t1 * D[2*i];
}

```

The difference is that `t0` and `t1` are automatic variables and can be held in registers whereas the array `t` will most likely be allocated in memory, and loaded and stored from memory for each operation.

If an input value `x[i]` or precomputed data `D[i]` is reused it makes sense to first copy the value into an automatic variable (`xi` or `Di`, respectively), and then reuse the automatic variable.

```

double t0, t1, x0, x1, D0;
for(i=0; i<8; i++)
{ x0 = x[2*i];
  x1 = x[2*i+1];
  D0 = D[2*i];
  t0 = x0 + x1;
  t1 = x0 - x1;
  y[2*i] = t0 * D0;
  y[2*i+1] = t1 * D0;
}

```

If the value of `y[i]` is used as source in operations like `y[i] += t0`, one should use scalar replacement for `y[i]`.

**Precomputation of constants.** In a CPU-bound kernel, all constants that are known ahead of time should be precomputed at compile time or initialization time and stored in a data array. At execution time, the kernel simply loads the precomputed data instead of needing to invoke math library functions. Consider the following example.

```

for(i=0; i<8; i++)
    y[i] = x[i] * sin(M_PI * i / 8);

```

The program contains an function call to the math library in the inner loop. Calling `sin()` can cost multiple thousands of cycles on modern CPUs. However, all the constants are known before entering the kernel and thus can be precomputed.

```

static double D[8];
void init()
{ for(int i=0; i<8; i++)
    D[i] = sin(M_PI * i / 8);
}

```

```

...
// in the kernel
for(i=0; i<8; i++)
    y[i] = x[i] * D[i];

```

The initialization function needs to be called only once. If the kernel is used over and over again, precomputation results in enormous savings. If the kernel is used only once, chances are that performance does not matter.

#### 4.4 Parameter-Based Performance Tuning and Program Generation

Many of the optimizations for the memory hierarchy discussed above have inherent degrees of freedom such as the block size for blocking or the degree of unrolling the code. While it may be possible to derive a reasonable estimation of these parameters, the complexity of modern microarchitecture makes an exact prediction impossible. In fact, often the best value may come as a surprise to the programmer. As a consequence, it makes sense to perform an empirical search to find those parameters. This means creating the variants, ideally through a set of scripts, through parameterized coding (for instance, defining all parameters as C preprocessor constants in a separate header file), or through other program generation techniques, and measuring their performance to find the best choice. Since the result may depend on the target platform, the search should be repeated for each new platform.

This parameter-based performance optimization is one of the techniques used in recent research on automatic performance tuning [14].

However, parameter based tuning is inherently not extensible in the sense that new forms of code or algorithm restructuring cannot be incorporated easily. Examples could be transformations for various forms of parallelism. A better solution than parameter-based tuning may be properly designed domain-specific languages used in tandem with rewriting systems. We will see the difference between these two approaches later in Section 5.4 and 6.6 where we discuss program generation for MMM and the DFT.

## 5 MMM

In this section, we optimize matrix-matrix multiplication (MMM) for the memory hierarchy. We explain the optimizations implemented by the ATLAS [13], and organize the steps as in Section 4. ATLAS is a program generator for MMM and other BLAS routines and also performs other optimizations not discussed here. It is introduced in Section 5.4.

Our presentation closely follows the one in Yotov et al. [75], which presents a model-based version of ATLAS.

For the rest of this section, we will assume the dimensions of the input matrices  $A$  and  $B$  to be  $N \times K$  and  $K \times M$  respectively, which implies an  $N \times M$  output matrix  $C$ . For simplicity, we will further assume that various optimization parameters are perfectly divisible by these dimensions whenever such a division is necessary. The computation considered is  $C = C + AB$ .

**Naive Implementation.** Matrix-matrix multiplication (MMM), as defined in Section 2.2, is naively implemented using the triple loop shown below. We use 2D array

notation (for instance, `C[i][j]`) to keep the code more readable. However, in an implementation where the matrix sizes are not known at compile time, one should resort to a linearized representation of  $C$ ,  $A$ , and  $B$  (see Section 4.1).

```
// K, M, N are compile-time constants
double C[N][M], A[N][K], B[K][M];
// Assume C is initialized to zero
for(i=0; i<N; i++)
  for(j=0; j<M; j++)
    { for(k=0; k<K; k++)
      C[i][j] += A[i][k] * B[k][j];
    }
}
```

The C language stores two-dimensional arrays in row-major order. Therefore, a cache miss to a (memory aligned) matrix element causes that element and adjacent elements in the same row being loaded into one cache line of the cache (see Fig. 7). Thus, accessing a large matrix by rows is cache efficient, while accessing it by columns is not.

Fig. 9 illustrates the data access pattern of the naive implementation. From this figure, we see the output locality of the computation: all accesses to each element in  $C$  are consecutive, and  $C$  is completed element by element, row by row. However, unless all input and output arrays fit into the cache, the naive implementation has poor locality with respect to  $A$  and  $B$ .

We analyze the naive implementation by counting the number of cache misses. We assume a cache line size of 64 bytes, or 8 (double precision) floating point values, and that  $N$  is large with respect to the cache size. To compute the first entry in  $C$ , we need to access the entire first row of  $A$  and the entire first column of  $B$ . Accessing a row of  $A$  results in  $N/8$  misses (one for each group of 8) due to the row-major storage order, while accessing a column of  $B$  results in a full  $N$  misses, yielding a total of  $(9/8)N$  misses for the first entry in  $C$ .

To analyze the computation of the second entry of  $C$ , we first observe that the parts of  $A$  and  $B$  that will be accessed first are not in the cache. That is, since  $N$  is much larger than the cache, the first few elements of the first row of  $A$  were in cache but were eventually overwritten. Similarly, the first elements of the second column of  $B$  were already in cache (each element shared a cache line with its neighbor in the first column) but also have been overwritten. This is illustrated in Fig. 10, which shows in gray the parts of  $A$  and  $B$  that are in cache after the first entry of  $C$  is computed. Consequently, the number of misses involved in computing the second entry (and every

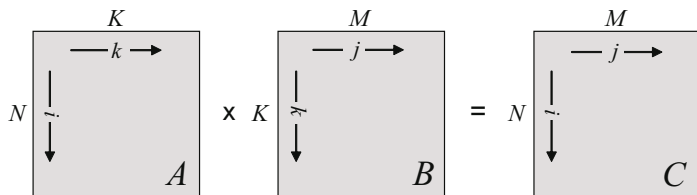
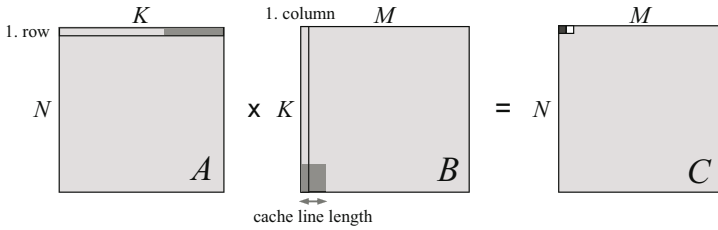


Fig. 9. Data access pattern for the naive MMM



**Fig. 10.** The state of the cache at the end of computation of the first element of  $C$  (small black square) is shown. Areas of the input matrices marked in gray are cache resident at this point. The next element of  $C$  to be computed is shown as small white square.

subsequent entry of  $C$ ), produces also  $(9/8)N$  misses. Therefore, the total number of misses generated by this algorithm (for the  $N^2$  entries in  $C$ ) is  $(9/8)N^3$ . In summary, there is no reuse and no neighbor use, a problem resolved to the extent possible by the optimizations in the next sections.

## 5.1 Cache Optimization

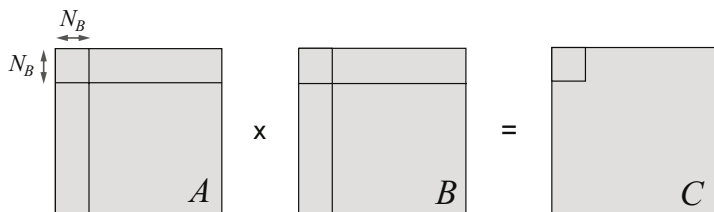
**Blocking.** One of the most important optimizations for MMM (and linear algebra problems in general) is blocking, as introduced in Section 4.2. Blocking involves performing the addition and multiplication operations on *blocks* of the original matrix, instead of individual elements. The idea is to increase locality by restricting the computation at any point to work on small chunks that fit entirely into the cache. We will also see that blocking essentially increases reuse and neighbor use, the concepts previously presented in Section 2.4.

The compiler loop transformation that implements blocking is known as *tiling* [13, 75, 76]. Blocking or tiling the MMM for each level of the memory hierarchy involves adding three more nested loops to the basic triple loop implementation. The code for the MMM blocked for one memory level with block size  $N_B$  follows.

```
// MMM loop nest (j, i, k)
for(i=0; i<N; i+=NB)
  for(j=0; j<M; j+=NB)
    for(k=0; k<K; k+=NB)
      // mini-MMM loop nest (i0, j0, k0)
      for(i0=i; i0<(i + NB); i0++)
        for(j0=j; j0<(j + NB); j0++)
          for(k0=k; k0<(k + NB); k0++)
            C[i0][j0] += A[i0][k0] + B[k0][j0];
```

Fig. 11 shows the data access pattern of blocking for the cache. The three additional innermost loops cause each matrix to be divided into blocks of size  $N_B \times N_B$ . Notice the similarity in the access pattern to the naive implementation, except at the block level instead of at the element level.

We now analyze this version of the MMM to determine the impact on the number of cache misses. We assume that the block size is larger than the cache line size, and



**Fig. 11.** Blocking for the cache: mini-MMMs

for now that several blocks can fit into the cache. This implies that accessing a block results only in  $N_B^2/8$  misses, regardless of the access sequence.

Computing the first *block* of  $C$  requires the first block row of  $A$ , and the first block column of  $B$ . This results in  $(N_B^2/8 + N_B^2/8)(N/N_B)$  cache misses. Similar to the reasoning used in the analysis of the naive version, computing each block of  $C$  results in the same amount of misses, and therefore, the total number of misses generated by this algorithm (for the  $(N/N_B)^2$  blocks in  $C$ ) is  $N^3/(4N_B)$ , which is significantly less than the  $(9/8)N^3$  misses in the naive version.

We call the smaller blocks operations mini-MMMs, following [75].  $N_B$  is an optimization parameter that must be chosen such that the working set of the mini-MMM fits entirely into the cache. A simple translation of our assumption that blocks from the two input and output matrices (our *working set*) fit into a fully associative cache is expressed by the following equation:  $3N_B^2 \leq C_s$ , where  $C_s$  is the cache size. ATLAS determines  $N_B$  by searching and trying different arbitrary values and picking the one that results in the best performance.

In contrast, [75] use a model based approach, and chooses  $N_B$  based directly on cache parameters. Their careful examination of the data access pattern of the blocked MMM reveals that the working set at a finer granularity consists only of a single element in  $C$  (since each element in  $C$  is reused completely by the innermost k0 loop before it moves on to the next element), a single row of  $A$  (since a row is fully reused before the program moves on to the next row), and the entire  $B$ . Therefore, the following relationship needs to hold:  $N_B^2 + N_B + 1 \leq C_s$ . Thus, a good choice for  $N_B$  is the largest value that satisfies this inequality.

Blocking for MMM works because it increases cache reuse and neighbor use, our guiding principles discussed in Section 2. Cache reuse is increased because once a block is brought into the cache, it is used several times before being overwritten. Neighbor use is increased for the input matrix  $B$ , since all elements in the cache line are used before eviction.

Typically, MMM is blocked for the L1 cache but blocking for the L2 cache may be superior in certain cases [75].

An additional optimization that can be done for the cache is to exchange the  $i$  and the  $j$  loops, depending upon the relative sizes of the  $A$  and  $B$  matrices.

**Loop merging.** Loop merging is not applicable to the MMM.

**Buffering.** Buffering (also known as copying) for MMM is applicable for large sizes. The basic idea behind buffering is to copy tiles of the input and output matrices into sequential order in memory to minimize cache conflict misses (and TLB misses if the

matrices span multiple pages), inside each mini-MMM. The following code illustrates buffering. The matrix  $B$  is fully buffered at the beginning since it is accessed in full during each iteration of the outermost  $i$  loop. Vertical panels of  $A$  are used during each iteration of  $j$ , and are buffered just before the  $j$  loop begins. Finally, in some cases, it might be beneficial to copy a single tile of  $C$  before the  $k$  loop, since a single tile is reused by each iteration of the  $k$  loop. Note that the benefits of buffering have to outweigh the costs, which might not hold true for very small or very large matrices.

```
// Buffer full B here
for(i=0; i<M; i+=NB)
  // Buffer a panel of A here
  for(j=0; j<N; j+=NB)
    // Copy a block (tile) of C here
    for(k=0; k<K; k+=NB)
      // mini-MMM loop nest as before (i0, j0, k0)
      ...
```

## 5.2 CPU and Register Level Optimization

We now look at optimizing the MMM for the CPU. We continue with our MMM example from the previous section.

**Blocking.** Blocking for the registers looks similar to blocking for the cache. Another set of nested triple loops is added. The resulting code is shown below:

```
// MMM loop nest (j, i, k)
for(i=0; i<N; i+=NB)
  for(j=0; j<M; j+=NB)
    for(k=0; k<K; k+=NB)
      // mini-MMM loop nest (i0, j0, k0)
      for(i0=i; i0<(i + NB); i0+=MU)
        for(j0=j; j0<(j + NB); j0+=NU)
          for(k0=k; k0<(k + NB); k0+=KU)
            // micro-MMM loop nest (j00, i00)
            for(k00=k0; k00<=(k0 + KU); k00++)
              for(j00=j0; j00<=(j0 + NU); j00++)
                for(i00=i0; i00<=(i0 + MU); i00++)
                  C[i00][j00]+=A[i00][k00]*B[k00][j00];
```

Note that the innermost loop nest now has the loop order  $kij$ ; this is explained later. As Fig. 12 shows, each mini-MMM is now computed by blocking it into a sequence of micro-MMMs. Each micro-MMM multiplies an  $M_U \times 1$  block of  $A$  by a  $1 \times N_U$  block of  $B$ , with the output being a  $M_U \times N_U$  block of  $C$ . At this level of blocking, we have a degree of freedom in choosing  $M_U$  and  $N_U$  (The  $K_U$  parameter controls the degree of unrolling, and is discussed soon). These parameters must be chosen so that a micro-MMM fits into register space (thus avoiding register spills).

ATLAS searches over arbitrary values for these parameters to choose the ones that result in the fastest code. In [75], with a reasoning that is similar to the one used in choosing  $N_B$  in the previous section, selects these parameters based on the inequality  $M_U + N_U + (M_U \times N_U) \leq N_R$ , where  $N_R$  is the number of data (integer or floating point) registers. This equality is then further refined.

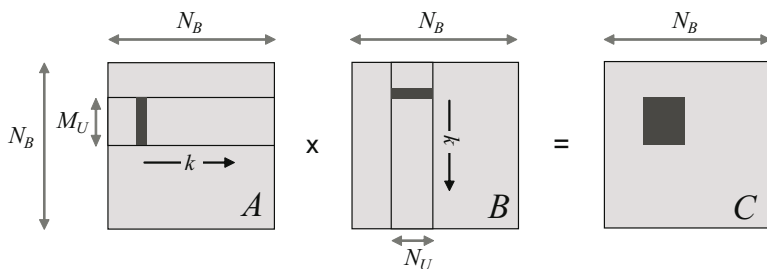


Fig. 12. mini-MMMs and micro-MMMs (from [75])

Locality is not the only objective of blocking for register space. Note that in the code above, the micro-MMM have a loop order of  $kij$ . While this reduces output locality, it also provides better instruction level parallelism (all the  $M_U N_U$  addition/multiplication pairs are independent) when combined with loop unrolling discussed next.

**Unrolling and scheduling.** Loop unrolling and scheduling, as discussed in Section 4.3, can be used to further optimize MMM. We unroll the two innermost loops to get  $M_U \times N_U$  additions and multiplications. Note that these instructions are of the form  $C+ = AB$ . As mentioned in [21], such an instruction will not execute well on machines without a fused multiply-add unit, since the addition is dependent on the multiplication, and will cause a pipeline stall until the multiplication is completed. Thus, it may be beneficial to separate the addition and the multiplication operations here, and schedule them with unrelated intervening instructions to minimize pipeline stalls.

The  $k00$  loop can also be unrolled completely to reduce loop overhead.  $K_U$  controls the degree of unrolling, and is chosen so that the fully unrolled loop body (of the  $k0$  loop) still fits into the L1 instruction cache.

**Scalar replacement.** When the innermost loops are unrolled, each array element appears multiple times in the unrolled code. For the reasons discussed earlier in Section 4.3, replacing array references by scalar variables in unrolled code enables compiler optimizations to work better. As the MMM has a good reuse ratio, references to input arrays are also replaced by first copying the value to automatic variables and then reusing the automatic variable.

**Precomputation of constants.** Since the MMM does not have constants that can be precomputed, this optimization does not apply.

### 5.3 Parameter-Based Performance Tuning

The above discussion identifies several parameters that can be used for tuning. ATLAS performs this tuning automatically by generating the variants and selecting the fastest using a search procedure.

**Blocking for cache.**  $N_B$  is the main optimization parameter used to control the block size of the mini-MMMs. If several levels of blocking are desired, additional blocking parameters arise.

**Blocking for registers.** When blocking for the registers,  $M_U$ , and  $N_U$  are the main tunable parameters, and must be chosen such that the micro-MMM does not produce register spills.  $K_U$  specifies the degree of unrolling and should be chosen as large as possible without overflowing the instruction cache.

Besides that, several other parameters can be identified for performance tuning and platform adaptation [21, 75].

#### 5.4 Program Generation for MMM: ATLAS

The parameters shown in the previous section are only a small subset of all the parameters that can be used to tune the MMM. In theory, searching over the space of all tunable parameters will lead to the fastest code. Obviously, such a search would take an impractical amount of time to complete due to the vast search space. The best approach in this scenario is to prune the search space in a reasonable way and to automate the search over the remaining space. This in essence is the approach followed by ATLAS [21], which is briefly discussed in this section. In terms of the language previously used in this tutorial, ATLAS generates a mini-MMM with the highest performance, which is then used as a kernel in a generic MMM function.

Fig. 13 shows the architecture of ATLAS. When ATLAS is first installed on a platform, it runs a set of micro-benchmarks to determine a set of hardware parameters, including the L1 cache size and the number of registers  $N_R$ . These parameters are then used to prune the originally unbounded search space to a finite one. ATLAS then proceeds by searching the space of possible mini-MMMs using a feedback loop. In this feedback loop, a search engine decides on the parameters that specify a mini-MMM, the corresponding code is generated, its performance evaluated, and the next set of parameters is tried.

Since the search space is too large, ATLAS uses an *orthogonal line search* to find the optimal values for the set of parameters it searches over. Given a function  $y = f(x_1, x_2, \dots, x_n)$  to optimize, orthogonal line search determines an approximation by solving a *sequence* of  $n$  1-dimensional optimization problems, where each problem corresponds to one of the  $n$  parameters. When optimizing for  $x_i$ , the set of optimal values already found for  $x_1 \dots x_{i-1}$  are used, and *reference values* are used for the remaining parameters  $x_{i+1} \dots x_n$ . ATLAS provides the parameter sequence, and ranges and reference values for each of the parameters, using a combination of built-in defaults and the determined microarchitectural parameters.

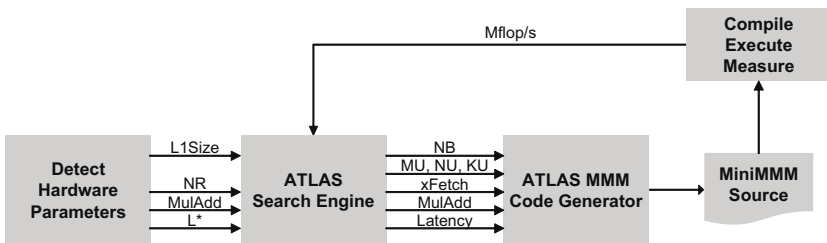


Fig. 13. Architecture of ATLAS (from [75])



It has been shown that a suitably designed model, based on a detailed understanding of the microarchitecture, can replace the search in ATLAS to find the best parameters deterministically [75].

**Discussion.** ATLAS has been very successful in generating very fast MMM code for many architectures and has been widely used. In fact, ATLAS, and its predecessor PHiPAC [22], were the first efforts on automatic performance tuning in the area of numerical computing; as such, it raised awareness to the increasing difficulty of deciding on coding choices and achieving high performance in general on machines with deep memory hierarchies. As we have seen, in this case, using program generation is crucial to efficiently evaluate the many possible choices of parameters.

However, since ATLAS is based on (properly chosen) parameters it is not clear how to extend its approach to novel architectural paradigms such as vector instructions, multicore processing, or others. To date, these are not supported by ATLAS. We argue that the reason is the lack of an internal domain-specific language that can express all the necessary transformations at a higher abstraction level, which also enables the inclusion of new transformations. This is the approach taken by Spiral, a program generator for the domain of linear transforms discussed later in Section 6.6.

## 5.5 Exercises

1. **Mini-MMM.** The goal of this exercise is to implement a fast mini-MMM to multiply two square  $N_B \times N_B$  matrices ( $N_B$  is a parameter), which is then used within an MMM.
  - (a) *Based on definition.* Use your naive implementation of the MMM as mini-MMM (code from Exercise 1 in Section 2).
  - (b) *Register blocking.* Block into micro MMMs with  $M_U = N_U = 2$ ,  $K_U = 1$ . The inner triple loop must have the  $kij$  order. Manually unroll the innermost  $i$  and  $j$  loops and schedule your code to perform alternating additions and multiplications (one operation per line of code). Perform scalar replacement on this unrolled code manually.
  - (c) *Unrolling.* Unroll the innermost  $k$  loop by a factor of 2 and 4 ( $K_U = 2, 4$ , which doubles and quadruples the loop body) and again do scalar replacement. Assume that 4 divides  $N_B$ .
  - (d) *Performance plot, search for best block size.* Determine the L1 data cache size  $C$  (in doubles, i.e., 8B units) of your computer. Measure the performance (in Mflop/s) of your four codes for all  $N_B$  with  $16 \leq N_B \leq \min(80, \sqrt{C})$  with 4 dividing  $N_B$ . Create a plot with the x-axis showing  $N_B$ , and y-axis showing performance. The plot should contain 4 lines: one line for each of the programs (MMM by definition, register blocking, and unrolling by a factor of 2 and 4). Discuss the plot, including answers to the following questions: which  $N_B$  and which code yields the maximum performance? What is the percentage of peak performance in this case?
  - (e) *Loop order.* Does it improve if in the best code so far you switch the outermost loop order from  $ijk$  to  $jik$ ? Create a plot to show the answer.

- (f) *Blocking for L2 cache.* Consider now your L2 cache instead. What is its size (in doubles)? Can you improve the performance of your fastest code so far by further increasing the block size  $N_B$  to block for L2 cache instead? Answer through an appropriate experiment and performance plot.

## 2. MMM

- Implement an MMM for multiplying two square  $N \times N$  matrices assuming  $N_B$  divides  $N$ , blocked into  $N_B \times N_B$  blocks. Use your best mini-MMM code from Exercise [1](#).
- Create a performance plot comparing this implementation and the implementation based on definition above for an interesting range of  $N$  (up to sizes where the matrices do not fit into the L2 cache). Plot the size  $N$  on the  $x$ -axis, against the performance (in Mflop/s or Gflop/s) on the  $y$ -axis.
- Analyze and discuss the plot.

## 6 DFT

In this section we describe the design and implementation of a high-performance function to compute the FFT. The approach we must take is different from the one taken to optimize the MMM in Section [5](#): we do not start with a naive implementation that is transformed into an optimized form, but design the code from scratch. This is due to the more complex structure of the available FFT algorithms. Note that, in contrast to MMM, an implementation based on the definition of the DFT is not competitive.

The first main problem is the choice of a suitable FFT algorithm, since many different variants are available that differ vastly in structure. It makes no sense to start with the wrong FFT algorithm and optimize the implementation step by step. In particular, when targeting a machine with a memory hierarchy, starting the optimization with the iterative radix-2 FFT used in Numerical Recipes (Section [2.3](#)) is suboptimal since it requires  $\log_2(\text{input size})$  many sweeps through the input data, which results in poor cache locality. Further, no unrolled and optimized basic block is used for optimal register performance.

In our discussion below we design a recursive radix-4 FFT implementation. Generalization to a mixed-radix recursive implementation is relatively straightforward in concept, but technically complex. The optimization steps taken follow to a large extent the design of FFTW 2.x [\[9\]](#). FFTW uses a program generator in addition, to automatically implement optimized unrolled basic blocks [\[23\]](#).

In all our DFT code examples the (complex) data is assumed to be stored in interleaved complex double-precision arrays (alternating real and imaginary parts of the vector elements). We pass around pointers of type `double`, and two neighboring `double` elements are one complex number. All strides are relative to complex numbers.

### 6.1 Background

In this section we provide background on the DFT and FFTs. We explain these algorithms using the Kronecker product formalism. We start with restating the DFT definition from Section [2.3](#). For code readability we denote the size of the input vector with

$N$ . As usual, matrices are written as  $A = [a_{k,\ell}]$ , where  $a_{k,\ell}$  are the matrix elements. An index range for  $k, \ell$  may be given in the subscript.

**Definition.** The discrete Fourier transform (DFT) of a complex input vector  $x$  of length  $N$  is defined as the matrix-vector product

$$y = \text{DFT}_N x, \quad \text{DFT}_N = [\omega_N^{k\ell}]_{0 \leq k, \ell < N}, \quad \omega_N = e^{-2\pi i/N}.$$

**Kronecker product formalism.** We describe fast algorithms for the DFT using the Kronecker product formalism [5]. There are several reasons for using this formalism: First, the representation is visual and index free and hence readable by humans. Second, it is easy to translate algorithms expressed this way directly into code, as we shall see later. Third, in this representation, algorithm variants are easily obtained by both inserting recursions into each other and manipulating algorithms to *match* them to a specific hardware architecture. For instance, the algorithms can be mapped to vector and multicore architectures this way [25, 26].

These are also the reasons why the program generator Spiral (explained in Section 6.6) uses this formalism as its internal domain-specific language.

We define  $I_n$  as the  $n \times n$  identity matrix. The tensor (or Kronecker) product of matrices is defined as

$$A \otimes B = [a_{k,\ell} B]_{k,\ell} \quad \text{with} \quad A = [a_{k,\ell}]_{k,\ell}.$$

In particular,

$$I_n \otimes A = \begin{bmatrix} A & & & \\ & A & & \\ & & \ddots & \\ & & & A \end{bmatrix}$$

is block-diagonal. We also introduce the iterative direct sum

$$\bigoplus_{i=0}^{n-1} A_i = \begin{bmatrix} A_0 & & & \\ & A_1 & & \\ & & \ddots & \\ & & & A_{n-1} \end{bmatrix},$$

which generalizes  $I_n \otimes A$ .

We visualize  $I_4 \otimes A$  below; the four  $A$ s are shown with different shades of gray.

$$I_4 \otimes A = \begin{array}{|c|} \hline \begin{array}{c} \text{A} \\ \text{A} \\ \text{A} \\ \text{A} \end{array} \\ \hline \end{array} \tag{1}$$

Now we look at the tensor product  $A \otimes I_n$ . This matrix also contains  $n$  blocks of  $A$ , but they are spread out and interleaved at stride  $n$ . This is best understood by visualization: the equivalent of (1) is

$$A \otimes I_4 = \begin{bmatrix} A & & & \\ & A & & \\ & & A & \\ & & & A \end{bmatrix} \tag{2}$$

where we assume that  $A$  is  $4 \times 4$ . All elements with the same shade of gray taken together constitute one  $A$ , so the matrix again contains four  $A$ s. The pattern shows that multiplying (2) to an input vector  $x$  is equivalent to multiplying  $A$  to four subvectors of  $x$ , extracted at stride 4, and writing the result into the same locations.

The stride permutation matrix  $L_m^{mn}$  permutes an input vector  $x$  of length  $mn$  as

$$in + j \mapsto jm + i, \quad 0 \leq i < m, \quad 0 \leq j < n.$$

If  $x$  is viewed as an  $n \times m$  matrix, stored in row-major order, then  $L_m^{mn}$  performs a transposition of this matrix.

**Recursive FFT.** Using the above formalism, the well-known Cooley-Tukey FFT in its recursive form can be written as a factorization of the  $\text{DFT}_N$  matrix into a product of sparse matrices. That is, for  $N = mn$ ,

$$\text{DFT}_{mn} = (\text{DFT}_m \otimes I_n) D_{m,n} (I_m \otimes \text{DFT}_n) L_m^{mn}. \tag{3}$$

Here  $D_{m,n}$  is the diagonal “twiddle” matrix defined as

$$D_{m,n} = \bigoplus_{j=0}^{m-1} \text{diag}(\omega_{mn}^0, \omega_{mn}^1, \dots, \omega_{mn}^{n-1})^j. \tag{4}$$

Equation (3) computes a DFT of size  $mn$  in four steps. First, the input vector is permuted by  $L_m^{mn}$ . Second,  $m$  DFTs of size  $n$  are computed recursively on segments of the vector. Third, the vector is scaled element wise by  $D_{m,n}$ . Lastly,  $n$  DFTs of size  $m$  are computed recursively at stride  $m$ .

The recursively called smaller DFTs are computed similarly until the base case  $n = 2$  is reached, which is computed by definition using an addition and a subtraction:

$$\text{DFT}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \tag{5}$$

In summary, (3) and (5) are sufficient to compute DFTs of arbitrary two-power sizes. To compute DFTs of other sizes, other FFT algorithms are required [5].

**Algorithms and formulas.** There is a degree of freedom in applying (3) to recursively compute a DFT, namely in factoring the given DFT input size  $N$ . For instance one can

**Table 4.** Translating formulas to code.  $x$  denotes the input and  $y$  the output vector. The subscript of  $A$  and  $B$  specifies the size of the (square) matrix. We use Matlab-like notation:  $x[b:s:e]$  denotes the subvector of  $x$  starting at  $b$ , ending at  $e$  and extracted at stride  $s$ .

formula	code
$y = (A_n B_n)x$	<code>t[0:1:n-1] = B(x[0:1:n-1]); y[0:1:n-1] = A(t[0:1:n-1]);</code>
$y = (I_m \otimes A_n)x$	<code>for (i=0; i&lt;m; i++)   y[i*n:1:i*n+n-1] =     A(x[i*n:1:i*n+n-1]);</code>
$y = (A_m \otimes I_n)x$	<code>for (i=0; i&lt;m; i++)   y[i:n:i+m-1] =     A(x[i:n:i+m-1]);</code>
$y = (\bigoplus_{i=0}^{m-1} A_n^i)x$	<code>for (i=0; i&lt;m; i++)   y[i*n:1:i*n+n-1] =     A(i, x[i*n:1:i*n+n-1]);</code>
$y = D_{m,n}x$	<code>for (i=0; i&lt;m*n; i++)   y[i] = Dmn[i]*x[i];</code>
$y = L_m^{mn}x$	<code>for (i=0; i&lt;m; i++)   for (j=0; j&lt;n; j++)     y[i+m*j]=x[n*i+j];</code>

factor  $8 \rightarrow 2 \times 4 \rightarrow 2 \times (2 \times 2)$  using two recursive applications of (3). The complete FFT algorithm for this factorization could then be written as the following *formula*:

$$\text{DFT}_8 = (\text{DFT}_2 \otimes I_4) D_{8,4} (I_2 \otimes (\text{DFT}_2 \otimes I_2)) D_{4,2} (I_2 \otimes \text{DFT}_2) L_2^4 L_2^8. \quad (6)$$

**Direct implementation.** A straightforward implementation of (3) can be easily obtained since the occurring matrix formulas have a direct interpretation in terms of code as shown in Table 4. The implementation of (3) would hence have four steps corresponding to the four factors in (3).

Observe in Table 4 that the multiplication of a vector by a tensor product containing an identity matrix can be computed using loops. The working set for each of the  $m$  iterations of  $y = (I_m \otimes A_n)x$  (see (1)) is a contiguous block of size  $n$  and the base address is increased by  $n$  between iterations. In contrast, the working sets of size  $m$  of the  $n$  iterations of  $y = (A_m \otimes I_n)x$  (see (2)) are interleaved, leading to stride  $n$  within one iteration and a unit stride base update across iterations.

**Cost analysis.** Computing the DFT using (3) requires, independent of the recursion strategy,  $n \log_2(n) + O(n)$  complex additions and  $\frac{1}{2}n \log_2(n) + O(n)$  complex multiplications.

The exact number of real operations depends on the chosen factorizations of  $n$  and is at most  $5n \log_2(n) + O(n)$ .

**Iterative FFTs.** The original FFT by Cooley and Tukey [77] was not the recursive algorithm (3), but an iterative equivalent and for  $N = 2^n$ . It can be obtained by expanding the DFT recursively always using the factorization  $N = 2 \cdot N/2$ , and then rearranging the parentheses and fusing adjacent permutations. The result is the iterative FFT

$$\text{DFT}_N = \left( \prod_{i=1}^k (I_{2^{i-1}} \otimes \text{DFT}_2 \otimes I_{N/2^i}) D'_{N,i} \right) R_N, \tag{7}$$

where the  $D'_{N,i}$  are diagonal matrices and  $R_N$  is the bit-reversal permutation [5]. Numerical Recipes implements a variant of (7), shown in Section 2.3

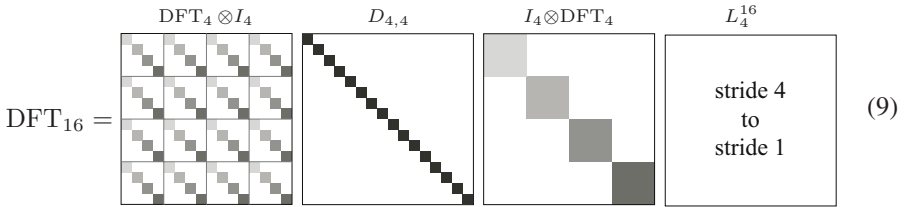
### 6.2 Cache Optimization

In this section we derive the recursive skeleton and the kernel specification for our DFT implementation.

**Blocking.** Blocking a DFT algorithm is done by choosing the recursive Cooley-Tukey FFT algorithm (3) as starting point instead of the iterative FFT used by the Numerical Recipes code in Section 2.3. The block size is the chosen *radix*  $m$  in (3), which is a degree of freedom. We assume a radix-4 implementation with  $N = 4^n$ , i.e., we factor  $N = 4 \cdot 4^{n-1}$ . The corresponding recursion is

$$\text{DFT}_{4^n} = (\text{DFT}_4 \otimes I_{4^{n-1}}) D_{4,4^{n-1}} (I_4 \otimes \text{DFT}_{4^{n-1}}) L_4^{4^n}. \tag{8}$$

We visualize (8) below for  $n = 2$ . We see four stages, corresponding to the four factors in the matrix factorization.



For  $n > 1$  our implementation will recursively apply (8) to the terms  $\text{DFT}_{4^{n-1}}$  in the right side of (8). The terms  $\text{DFT}_4$  are recursion leaves and not implemented using (8). We will discuss their implementation in Section 6.3.

This recursion is *right-expanded*—the first stage gets recursively expanded while the second stage uses radix-4 kernels. Right-expanded recursive implementations have superior data locality as only a small amount of temporary storage is needed and the second stage can be implemented in-place.

**Loop merging.** A naive implementation of (8) leads to a recursive function with four stages (corresponding to the four matrix factors) and thus four sweeps through the data. However, the stride permutation  $L_4^{4^n}$  is just a data reordering and thus is a candidate for loop merging. Similarly, the twiddle factor matrix  $D_{4,4^{n-1}}$  is a diagonal matrix and can be merged with the subsequent stage.

We now sketch the derivation of a recursive implementation of (8). We partition (8) into two expressions as

$$\text{DFT}_{4^n} = \left( (\text{DFT}_4 \otimes I_{4^{n-1}}) D_{4,4^{n-1}} \right) \cdot \left( (I_4 \otimes \text{DFT}_{4^{n-1}}) L_4^{4^n} \right), \quad (10)$$

which become two stages (instead of four) in the recursive function

```
void DFT(int N, double *Y, double *X);
```

that implements (8).

For  $n = 2$  we visualize the merging of the stride permutation with the adjacent tensor product,  $\text{DFT}_4 \otimes I_4$ , in (11) below. The merging of the diagonal  $D_{4,4}$  with the adjacent tensor product  $I_4 \otimes \text{DFT}_4$  cannot easily be visualized.

$$\text{DFT}_{16} = \begin{matrix} \text{DFT}_4 \otimes I_4 & D_{4,4} & (I_4 \otimes \text{DFT}_4) L_4^{16} \end{matrix} \quad (11)$$

The diagram shows three square matrices representing stages of a 16-point DFT. The first matrix, labeled  $\text{DFT}_4 \otimes I_4$ , is a 16x16 grid with a repeating 4x4 block pattern. The second matrix, labeled  $D_{4,4}$ , is a 16x16 diagonal matrix with a single 4x4 block of ones. The third matrix, labeled  $(I_4 \otimes \text{DFT}_4) L_4^{16}$ , is a 16x16 grid with a repeating 4x4 block pattern, similar to the first matrix but with different internal structure.

The first stage of (10),  $y = (I_4 \otimes \text{DFT}_{4^{n-1}}) L_4^{4^n} x$ , is handled as follows. According to Table 4, the tensor product  $I_4 \otimes \text{DFT}_{4^{n-1}}$  alone is translated into a loop with 4 iterations. The same is true for  $(I_4 \otimes \text{DFT}_{4^{n-1}}) L_4^{4^n}$ ; only, as (11) shows, the input is now read at stride 4 but the output is still written at stride 1. This means that the corresponding DFT function needs to have the stride as an additional parameter and has to be implemented out-of-place, i.e.,  $x$  and  $y$  need to be different memory regions. Hence it is of the form

```
void DFT_rec(int N, int n, double *Y, double *X, int s)
```

We pass  $n$  together with  $N$  to avoid computing the logarithm.

Now the function DFT above just becomes a special case of DFT\_rec and can hence be implemented using a C macro ( $\log_4()$ , computes  $n$  from  $4^n$ ):

```
#define DFT(N, Y, X) DFT_rec(N, log4(N), Y, X, 1)
```

For  $N = 4$ , we reach the leaf of the recursion and call a base case kernel function.

```
void DFT4_base(double *Y, double *X, int s);
```

The second stage,  $y = (\text{DFT}_4 \otimes I_{4^{n-1}}) D_{4,4^{n-1}} x$ , first scales the input by a diagonal matrix and then sweeps with a  $\text{DFT}_4$  kernel over it, applied at a stride. More precisely,  $\text{DFT}_4$  operates on  $x_j, x_{j+4^{n-1}}, x_{j+2 \cdot 4^{n-1}}$ , and  $x_{j+3 \cdot 4^{n-1}}$ , where  $j$  is the loop iteration number.

Again, we merge these two steps, this time by replacing the  $\text{DFT}_4$ s in  $\text{DFT}_4 \otimes I_{4^{n-1}}$  by  $\text{DFT}_4 D_j$ , where  $D_j$  is a  $4 \times 4$  diagonal matrix containing the proper diagonal elements from  $D_{4,4^{n-1}}$ . Inspection shows that  $D_j$  (as a function of the problem size  $4^n$ ) is given by

$$D_j = \text{diag}(\omega_{4^n}^0, \omega_{4^n}^j, \omega_{4^n}^{2j}, \omega_{4^n}^{3j}), \quad 0 \leq j < 4^{n-1}. \quad (12)$$

Hence, the function implementing  $y = (\text{DFT}_4 D_j)x$  also needs a stride as parameter, and  $j$  to compute the elements of  $D_j$ . Also, it can be in-place since it reads from and writes to the same locations of input and output vector. Hence it takes the form:

```
void DFT4_twiddle(double *Y, int s, int n, int j);
```

The final recursive function is given below. There are some address multiplications by 2, required to implement arrays of complex numbers as arrays (of twice the size) of real numbers.

```
// recursive radix-4 DFT implementation

// compute the exponent
#include <math.h>
#define log4(N) (int)(log(N)/log(4))

// top-level call to DFT function
#define DFT(N, Y, X) DFT_rec(N, log4(N), Y, X, 1)

// DFT kernels
void DFT4_base(double *Y, double *X, int s);
void DFT4_twiddle(double *Y, int s, int N, int j);

// recursive radix-4 DFT function
// N: problem size
// Y: output vector
// X: input vector
// s: stride to access X
void DFT_rec(int N, int n, double *Y, double *X, int s)
{ int j;

  if (N==4)
    // Y = DFT_4 X
    DFT4_base(Y, X, s);
  else {
    // Y = (I_4 x DFT_{N/4}) (L^{N_4}) X
    for(j=0; j<4; j++)
      DFT_rec(N/4, n-1, Y+(2*(N/4)*j), X+2*j*s, s*4);
    // Y = (DFT_4 x I_{N/4}) (D_{N,4}) Y
    for(j=0; j<N/4; j++)
      DFT4_twiddle(Y+2*j, N/4, n, j);
  }
}
```

**Buffering.** The kernel `DFT4_twiddle` accesses both input and output in a stride. For large sizes  $N = 4^n$ , this stride is a large two-power, which means that all elements accessed by the kernel are mapped to the same set in the cache (see Fig. 6). If the cache does not have sufficient associativity, cache thrashing occurs. Namely, each iteration of the `DFT4_twiddle` loop has to load 4 cache lines and all these cache lines get evicted before the next iteration of the `DFT4_twiddle` loop can use the already loaded remaining cache lines.



Buffering alleviates these problems to a certain degree. An initial and final copy operation introduce overheads, but all intermediate steps are done on contiguous data, preventing cache thrashing.

As an example, buffering is performed on the second loop of the preceding code, leading to the following code. We assume a cache line size of  $LS$  complex numbers (= 4 doubles). (If  $LS$  is larger than the radix size, one needs special cases for some recursion steps.) To implement buffering, we first split the  $j$  loop into  $N / (2 * LS) \times LS$  iterations. We add copying to the body of the *outer* tiled  $j1$  loop. Our copy operation handles cache lines and thus data for multiple DFTs. In particular, we copy 4 sets of  $LS$  consecutive complex elements (4 cache lines) into a local buffer. The inner tiled  $j2$  loop performs  $LS$  DFTs on the local contiguous buffer. The large, performance degrading complex stride  $4^{n-1}$  in the original  $j$  loop gets replaced by a small complex stride  $LS$  in the  $j2$  loop at the cost of two copy operations that copy whole cache lines. The threshold parameter  $th$  controls the sizes for which the second loop gets buffered.

```
// cache line size = 2 complex numbers (16 bytes)
# define LS    2

// recursive radix-4 DFT function with buffering
// N: problem size
// Y: output vector
// X: input vector
// s: stride to access X
// th: threshold size to stop buffering

void DFT_buf_rec(int N, int n, double *Y, double *X, int s, int th)
{ int i, j, j1, j2, k;
  // local buffer
  double buf[8*LS];

  if (N==4)
    // Y = DFT_4 X
    DFT4_base(Y, X, s);
  else
    { // Y = (I_4 x DFT_{N/4}) (L^N_4) X
      if (N > th)
        for(j=0; j<4; j++)
          DFT_buf_rec(N/4, n-1, Y+(2*(N/4)*j), X+2*j*s, s*4, th);
      else
        for(j=0; j<4; j++)
          DFT_rec(N/4, n-1, Y+(2*(N/4)*j), X+2*j*s, s*4);

      // Y = (DFT_4 x I_{N/4}) (D_{N,4}) Y, buffered for LS
      // j loop tiled by LS
      for(j1=0; j1<N/(4*LS); j1++)
        { // copy 4 chunks of 2*LS double to local buffer
          for(i=0; i<4; i++)
            for(k=0; k<2*LS; k++)
              buf[2*LS*i+k] = Y[(2*LS*j1)+(2*(N/4)*i)+k];
        }
    }
}
```

```

// perform LS DFT4 on contiguous data
// buf = (DFT4 Dj x I_LS) buf
for(j2=0; j2<LS; j2++)
    DFT4_twiddle(buf+2*j2, LS, n, j1*LS+j2);

// copy 4 chunks of 2*LS double to output
for(i=0; i<4; i++)
    for(k=0; k<2*LS; k++)
        Y[(2*LS*j1)+(2*(N/4)*i)+k] = buf[2*LS*i+k];
}
}
}

```

One can perform a similar buffering operation on the input  $X$  for the call to `DFT_rec`, as  $X$  is accessed at a large stride. This buffering must take place as special case for  $N = 16$  in `DFT_rec` and requires a third variant of the recursive function `DFT_rec`.

### 6.3 CPU and Register Level Optimization

This section describes the design and implementation of optimized DFT base cases (kernels). We again restrict the discussion to the recursive radix-4 FFT algorithm. Extensions to mixed-radix implementations requires different kernel sizes, all implemented following the ideas presented in this section. High-performance implementations may use kernels of up to size 64 [23, 78].

**Blocking.** We apply (3) to the  $DFT_4$ :

$$DFT_4 = (DFT_2 \otimes I_2) D_{4,2} (I_2 \otimes DFT_2) L_2^4. \quad (13)$$

As (13) is a recursive formula, an implementation based on (13) is automatically blocked.

**Unrolling and scheduling.** We implement (13) according to the rules summarized in Table 4. We aim at implementing recursion leafs. Thus the code needs to be unrolled. Due to the recursive nature of (13), kernels derived from (13) are automatically reasonably scheduled.

For DFT kernels, larger unrolled kernels lead to slightly less operations, as more twiddle factors are known at optimization time and one can take better advantage of trivial complex multiplications. However, larger kernels do not increase the available instruction level parallelism as much as in MMM, since the DFT data flow is more complicated and imposes stronger constraints on the operation ordering.

**Scalar replacement.** We next apply scalar replacement as described in Section 4.3. Every element in the input array  $X$  is only referenced twice, and every location of the output array  $Y$  is written once. Hence, we only replace the temporary array  $t$  by scalar variables, but do not replace accesses to  $X$  and  $Y$ . Experiments suggest that this strategy is sufficient for obtaining maximum performance. This leads to the following code for `DFT4_base`. From the discussion in Section 6.2 we know that this function loads at complex stride  $s$  from  $*X$  and writes at unit stride to  $*Y$ . We obtain the following code:

```
// DFT4 implementation
void DFT4_base(double *Y, double *X, int s)
{ double t0, t1, t2, t3, t4, t5, t6, t7;
  t0 = (X[0] + X[4*s]);
  t1 = (X[2*s] + X[6*s]);
  t2 = (X[1] + X[4*s+1]);
  t3 = (X[2*s+1] + X[6*s+1]);
  t4 = (X[0] - X[4*s]);
  t5 = (X[2*s+1] - X[6*s+1]);
  t6 = (X[1] - X[4*s+1]);
  t7 = (X[2*s] - X[6*s]);
  Y[0] = (t0 + t1);
  Y[1] = (t2 + t3);
  Y[4] = (t0 - t1);
  Y[5] = (t2 - t3);
  Y[2] = (t4 - t5);
  Y[3] = (t6 + t7);
  Y[6] = (t4 + t5);
  Y[7] = (t6 - t7);
}
```

**Precomputation of constants.** The kernel `DFT4_twiddle` computes  $y = (\text{DFT}_4 D_j)x$ , which contains multiplication with the complex diagonal  $D_j$  as defined in (12). The entries of  $D_j$  are complex roots of unity (twiddle factors) that depend on the recursion level and the loop counter  $j$ . Computing the actual entries of  $D_j$  requires evaluations of  $\sin \frac{k\pi}{N}$  and  $\cos \frac{k\pi}{N}$  for suitable values of  $k$  and  $N$ , which requires expensive calls to the math library. Hence these numbers should be precomputed.

We introduce an initialization function `init_DFT` that precomputes all diagonals required for size  $N$  and stores pointers to the tables (one table for each recursion level) in the global variable `double **DN`, as shown below.

```
#define PI 3.14159265358979323846
// twiddle table, initialized by init_DFT(N)
double **DN;

void init_DFT(int N)
{ int i, j, k, size_Dj = 16, n_max = log4(N);
  DN = malloc(sizeof(double*)*(n_max-1));

  for (j=1; j<n_max; j++, size_Dj*=4)
  { double *Dj = DN[j-1] = malloc(2*sizeof(double)*size_Dj);
    for (k=0; k<size_Dj/4; k++)
      for (i=0; i<4; i++)
        { *(Dj++) = cos(2*PI*i*k/size_Dj);
          *(Dj++) = sin(2*PI*i*k/size_Dj);
        }
  }
}
```

The function `DFT4_twiddle` is shown below.

```
// C macro for complex multiplication
#define CMPLX_MULT(cr, ci, a, b, idx, s) \
{ double ar, ai, br, bi; \
  ar = a[2*s*idx]; ai = a[2*s*idx+1]; \
  br = b[2*idx]; bi = b[2*idx+1]; \
  cr = ar*br - ai*bi; \
  ci = ar*bi + ai*br; \
}

// DFT4*D_j implementation
void DFT4_twiddle(double *Y, int s, int n, int j)
{ double t0, t1, t2, t3, t4, t5, t6, t7,
  X0, X1, X2, X3, X4, X5, X6, X7;
  double *Dj;

  // complex multiplications from D_N
  Dj = DN[n-2]+8*j;
  CMPLX_MULT(X0, X1, Y, Dj, 0, s);
  CMPLX_MULT(X2, X3, Y, Dj, 1, s);
  CMPLX_MULT(X4, X5, Y, Dj, 2, s);
  CMPLX_MULT(X6, X7, Y, Dj, 3, s);

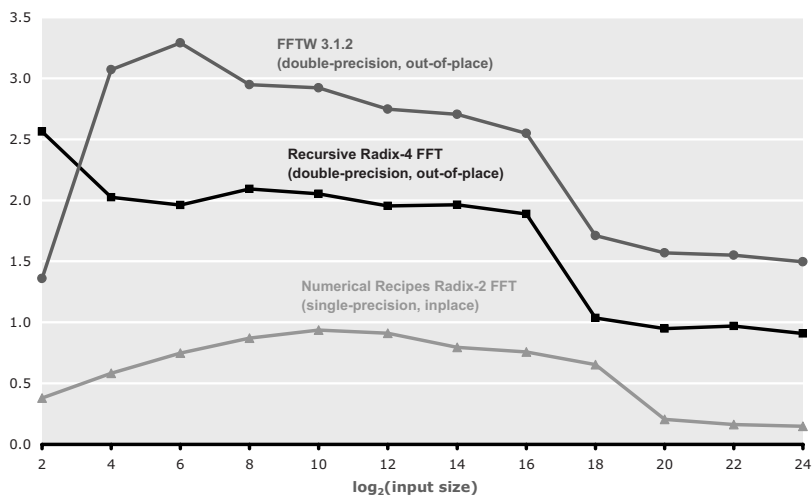
  // operations from DFT4
  t0 = (X0 + X4);
  t1 = (X2 + X6);
  t2 = (X1 + X5);
  t3 = (X3 + X7);
  t4 = (X0 - X4);
  t5 = (X3 - X7);
  t6 = (X1 - X5);
  t7 = (X2 - X6);
  Y[0] = (t0 + t1);
  Y[1] = (t2 + t3);
  Y[4*s] = (t0 - t1);
  Y[4*s+1] = (t2 - t3);
  Y[2*s] = (t4 - t5);
  Y[2*s+1] = (t6 + t7);
  Y[6*s] = (t4 + t5);
  Y[6*s+1] = (t6 - t7);
}
```

## 6.4 Performance Evaluation

We now evaluate the performance of the recursive radix-4 FFT derived in this section and compare it to the Numerical Recipes and the sequential, scalar (single core, x87) version of FFTW 3.1.2. All implementations are run on a single core of a 2.66 GHz Intel Core2 Duo, with a theoretical scalar peak performance of 5.32 Gflop/s. We compile all implementation with the Intel C++ compiler 10.0 with options “/O3/QxT” to obtain

**DFT on 2.66 GHz Core2 Duo (32-bit Windows XP, Single Core, x87)**

performance [Gflop/s]



**Fig. 14.** Performance results for three FFT implementations on a 2.66 GHz Intel Core2 Duo. All implementations are sequential and scalar (single core, x87). Higher is better.

maximum optimization. The radix-4 implementation was copied directly from the code listings above. The Numerical Recipes FFT implementation is in single-precision and in-place while both our radix-4 FFT and FFTW are double-precision and out-of-place. This gives a slight performance advantage to the Numerical Recipes FFT implementation.

Fig. 14 shows the performance results for the three FFT implementations. We see that Numerical Recipes reaches about 1 Gflop/s and drops sharply to 160 Mflop/s when the memory footprint for the problems is too large for the L2 cache. The radix-4 FFT implementation we derived in this tutorial reaches about 2 Gflop/s for problem sizes that fit into the L2 cache. For larger sizes the performance drops down to about 1 Gflop/s. FFTW 3.1.2 in sequential scalar mode shows the upper bound for practically achievable performance when using x87 instructions and a single core. FFTW reaches about 2.5–3 Gflop/s for cache-resident sizes and 1.6 Gflop/s for out-of-cache sizes.

Analysis of the above data can be summarized as follows.

- The recursive radix-4 FFT is twice as fast as Numerical Recipes for in-cache sizes and about 6 times faster for out-of-cache sizes.
- The radix-4 FFT implementation reaches more than two thirds of the performance of scalar FFTW. The performance difference is mainly due to FFTW's larger basic block sizes (codelets), its ability to choose different radices at different recursion steps, and a few additional loop optimizations.
- There is still a lot of room for further improvement using Intel's SSE instructions and both cores (see Fig. 2).

In addition, our experiments show that buffering does not produce any performance gain in this case, since the cache associativity on the Core2 architecture is 8, which is large enough for a radix-4 kernel.

## 6.5 Parameter-Based Performance Tuning

We now discuss the parameters in our DFT implementation that can be tuned to the memory hierarchy.

**Base case sizes.** The most important parameter tuning is the selection of base cases. To allow for multiple base cases `DFT_base` and `DFT_twiddle`, the program structure must become more complex, as a data structure describing the recursion and containing function pointers to the appropriate kernels replaces the two parameters `N` and `n` in `DFT_rec`. The resulting program would be very similar to FFTW 2.x.

After this infrastructural change the system can apply any function `DFT_twiddle` in the second stage of the recursion and any function `DFT_base` as recursion leaf. The tuning process needs to find for each recursion step the right kernel size. FFTW uses both a cost estimation and runtime experiments based on dynamic programming to find good parameter choices [10]. Showing the full implementation is beyond the scope of this tutorial.

**Threshold for buffering.** The second parameter decides the sizes for which buffering should be applied. This depends on the cache size of the target machine, as buffering only becomes beneficial for problem sizes that are not resident in the L2 cache.

**Buffer size.** Finally, we need to set the buffer size based on the cache line size of the target machine to prevent cache thrashing. The cache line size can be either looked up or found experimentally.

## 6.6 Program Generation for DFT: Spiral

Spiral [7] is a program generator for linear transforms. It can generate optimized fixed-size and variable-size code for the DFT, the Walsh-Hadamard transform (WHT), the discrete cosine and sine transforms, finite impulse response (FIR) filters, the discrete wavelet transform, and others. Spiral builds on the Kronecker product framework for the DFT, described in Section 6.1 but extends it to the whole domain of linear transforms. Further, Spiral automates the optimization process outlined in Sections 6.2–6.5 as well as many other optimizations including various forms of parallelization [26, 54, 79, 80]. The fastest FFT implementation shown in Fig. 2 is generated using Spiral.

In contrast to ATLAS, Spiral is not based on searching a parameterized space, but on a domain-specific language (DSL) that enables the enumeration and systematic optimization of algorithms. More specifically, there are two key ideas underlying Spiral:

1. *Mathematical, structural, declarative DSL.* Spiral uses a DSL to describe algorithms. The DSL is called SPL [81] and is directly derived from the transform domain: it is precisely (an extension of) the Kronecker formalism described in Section 6.1. The language describes only the structure of algorithms and is hence

declarative. This property enables structural algorithm optimizations including parallelization that is not practical to perform on C code.

2. *Optimization through rewriting.* Spiral uses rewriting systems [82] for both the generation of alternative algorithms and the structural optimization of algorithms at a high level of abstraction. The rewriting rules for the former are divide-and-conquer algorithms specified as in (3) and for the latter, they are known matrix identities.

**Architecture.** The input to Spiral is a formally specified transform (for instance,  $\text{DFT}_{384}$ ); the output is a highly optimized C program implementing the transform. These highly optimized programs may use language extensions or software libraries to access special machine features like multiple cores or SIMD vector instructions. We show the architecture of Spiral in Fig. 15 and discuss it below.

- *Algorithm level.* This stage is responsible for generating and optimizing algorithms for the specified transforms.

- *Formula generation.* A transform like  $\text{DFT}_{384}$  is considered to be a non-terminal. Spiral uses *breakdown rules* to describe recursive algorithms for linear transforms. For example, (3) and (7) are breakdown rules expressing larger DFTs in terms of smaller DFTs. Base cases terminate the recursion. For instance, (5) is the DFT base rule.

A rewriting system recursively applies breakdown rules to the specified transform to produce alternative algorithms represented as SPL expressions, also called formulas.

- *Formula optimization.* Formulas are structurally optimized, also using a rewriting system. Loop fusion is performed using rewriting rules which essentially perform the same reasoning and restructuring as described in Section 6.2. The loop fusion by rewriting requires the extension of SPL to a more powerful

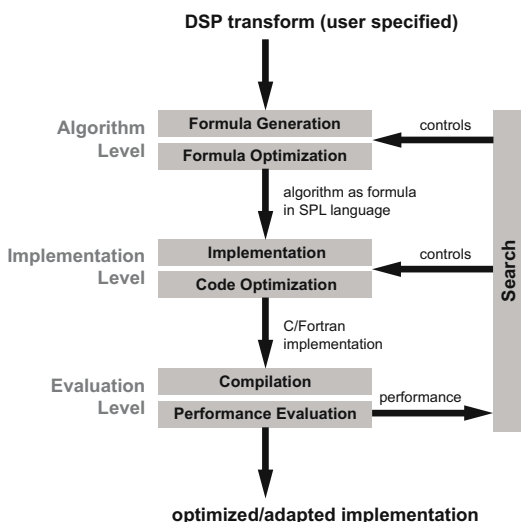


Fig. 15. The architecture of Spiral (from [7])

language called  $\Sigma$ -SPL [24]. Further, rewriting is used for various forms of parallelization including the efficient mapping to multiple processor cores or SIMD vector instructions. The next section will provide more details on this topic.

- *Implementation level.* Spiral contains a special-purpose compiler that translates formulas into code. The compiler is based on (an extension of) Table 4. Moreover, it performs all kernel-level optimizations described in Section 6.3. Depending upon an unrolling threshold, subformulas smaller than the threshold are treated as kernels, while larger formulas are implemented using loops.
- *Evaluation level.* This stage is responsible for compiling and measuring the runtime of the generated code.
- *Search.* The measured runtime guides Spiral in picking a new candidate formula by changing the breakdown of the non-terminal. The feedback loop is guided by a search strategy, usually a form of dynamic programming. The main purpose of the search is adaptation to the platform’s memory hierarchy.

**Structural optimization through rewriting.** A core component of Spiral’s optimization process is the structural optimization of formulas using a rewriting system. As briefly discussed above, two major optimization goals are achieved through rewriting: 1) loop merging [24], and 2) the mapping of algorithms to parallel architectures like multicore CPUs or SIMD vector extensions [26, 54]. Loop merging is beyond the scope of this tutorial as it requires the introduction of a new language,  $\Sigma$ -SPL. Thus, we only briefly discuss the mapping to parallel architectures.

Analysis of the access pattern of tensor products shows that certain tensor products can be mapped very well to some architectures but only poorly to others. As example, in (3) the construct

$$I_m \otimes \text{DFT}_n \quad (14)$$

has a perfect structure for  $m$ -way parallel machines with either shared or distributed memory. However, implementing it with SIMD vector instructions introduces considerable overhead [54]. Similarly, the construct

$$\text{DFT}_m \otimes I_n \quad (15)$$

has a perfect structure for  $n$ -way vector SIMD architectures. However, implementing it on shared memory machines leads to false sharing, while on distributed memory machines tiny messages would be required, which degrades performance.

Using algebraic identities [53] one can change the structure of formulas. For instance, the identity

$$\text{DFT}_m \otimes I_n = L_m^{mn} (I_n \otimes \text{DFT}_m) L_n^{mn} \quad (16)$$

replaces a vector formula by a parallel formula and introduces two stride permutations.

Spiral uses a rewriting system to perform formula manipulations like (16), using a tagging mechanism to steer the manipulation toward the final formula optimized for a certain architecture. Spiral’s rewriting system consists of three main components.



- *Tags* encode target architecture types and parameters. They contain high-level information about the target architecture. For instance, Spiral uses the tags “ $\text{vec}(\nu)$ ” for SIMD vector extensions ( $\nu$  encodes the vector length of the architecture) and “ $\text{smp}(p, \mu)$ ” for shared memory ( $p$  is the number of processors and  $\mu$  the length of cache lines).
- *Base cases* describe formula constructs that are guaranteed to be mapped efficiently to the target hardware. Spiral uses special operator variants to encode base cases. For instance, a  $p$ -way parallel base case is denoted by the tagged operator “ $\otimes_{\parallel}$ ”;  $A_n$  is any  $n \times n$  matrix expression.
- *Rewriting rules* encode formula manipulation identities, but in addition “know” the target machine and thus deduce the “right” parameters for identities with degrees of freedom. For instance, the identity (I16) is translated into the rewriting rule

$$\underbrace{A_m \otimes I_n}_{\text{smp}(p, \mu)} \rightarrow \underbrace{L_m^{mn}}_{\text{smp}(p, \mu)} \left( I_p \otimes_{\parallel} (I_{n/p} \otimes A_m) \right) \underbrace{L_n^{mn}}_{\text{smp}(p, \mu)} .$$

This rule has the additional knowledge of the target system’s processor count, and utilizes this knowledge when applying the helper identity

$$I_{mn} = I_m \otimes I_n .$$

The stride permutations  $L_m^{mn}$  and  $L_n^{mn}$  will be handled by further rewriting.

For every type of parallelism, these three components are added to Spiral to enable the corresponding structural optimization. In addition, every class of target machines may require a small extension of the SPL compiler to translate tagged operators into target code. For instance, “ $\otimes_{\parallel}$ ” will be translated into OpenMP parallel for loops, when Spiral generates shared memory code using OpenMP.

**Discussion.** Spiral fully automates the process of optimizing linear transforms for a large class of state-of-the-art architectures. The code it generates is competitive with expertly hand-tuned implementations and often outperforms these. The key is Spiral’s domain-specific, declarative, mathematical language to describe algorithms. Spiral’s algorithm (breakdown rule) database contains the algorithmic knowledge of more than a hundred journal papers on transform algorithms. Spiral’s rewriting system is the key to structural optimization and parallelization of algorithms. With this approach it is possible to re-target Spiral to new parallel platforms. So far Spiral successfully generated (at least prototypical) fast implementations for SIMD vector extensions, multicore CPUs, cluster computers, graphics processors (GPUs), and the Cell BE processor. In addition, Spiral generates hardware designs for field-programmable gate arrays (FPGAs), and hardware-software partitioned implementations.

While Spiral focuses on transforms, the basic principles underlying it may be applicable to other numerical problem domains.

## 6.7 Exercises

1. **WHT: Operations count.** The Walsh–Hadamard transform (WHT) is related to the DFT but has a simpler structure and simpler algorithms. The WHT is defined only

for 2-power input sizes  $N = 2^n$ , as given by the matrix

$$\text{WHT}_{2^n} = \underbrace{\text{DFT}_2 \otimes \text{DFT}_2 \otimes \dots \otimes \text{DFT}_2}_{n \text{ factors}}$$

where  $\text{DFT}_2$  is as defined in (5).

- (a) How many entries of the WHT are zeros and why? Determine the number of additions and the number of multiplications required when computing the WHT by definition.
- (b) The WHT of an input vector can be computed iteratively or recursively using the following formulas:

$$\text{WHT}_{2^n} = \prod_{i=0}^{n-1} (I_{2^{n-i-1}} \otimes \text{DFT}_2 \otimes I_{2^i}) \quad (\text{iterative}) \quad (17)$$

$$\text{WHT}_{2^n} = (\text{DFT}_2 \otimes I_{2^{n-1}})(I_2 \otimes \text{WHT}_{2^{n-1}}) \quad (\text{recursive}) \quad (18)$$

- (c) Determine the exact operations counts (again, additions and multiplications separately) of both algorithms. Also determine the degree of reuse as defined in Section 2.1

## 2. WHT: Implementation

- (a) Implement a recursive implementation of the WHT based on (18).
- (b) Implement the triple loop (iterative) version of the WHT using (17). Create a performance plot (size versus Mflop/s) for sizes  $2^1$ – $2^{20}$  comparing the iterative and the recursive versions. Discuss the plot.
- (c) Create unrolled WHTs of sizes 4 and 8 based on the recursive WHT algorithm. (The number of operations should match the cost computed in Exercise 1c on page 254).
- (d) Now implement recursive radix-4 and radix-8 implementations of the WHT based on the formulas

$$\text{WHT}_{2^n} = (\text{WHT}_4 \otimes I_{2^{n-2}})(I_4 \otimes \text{WHT}_{2^{n-2}}) \quad (\text{radix-4})$$

$$\text{WHT}_{2^n} = (\text{WHT}_8 \otimes I_{2^{n-3}})(I_8 \otimes \text{WHT}_{2^{n-3}}) \quad (\text{radix-8})$$

In these implementations, the left hand side WHT (of size 4 or 8) should be your unrolled kernel (which then has to handle input data at a stride) called in a loop; the right hand side is a recursive call (also called in a loop). Further, in both implementations, you may need one step with a different radix to handle all input sizes.

Measure the performance of both implementations, again for sizes  $2^1$ – $2^{20}$  and add it to the previous plot (four lines total).

- (e) Try to further improve the code or perform other interesting experiments. For example, what happens if one considers more general algorithms based on

$$\text{WHT}_{2^n} = (\text{WHT}_{2^i} \otimes I_{2^{n-i}})(I_{2^i} \otimes \text{WHT}_{2^{n-i}})$$

The unrolled code could be the WHT on the left hand side of the above equation. Alternatively, one could run a search to find the best radix in each step independently.

## 7 Conclusions

Writing fast libraries for numerical problems is difficult and requires a thorough understanding of the interaction between algorithms, software, and microarchitecture. Looking ahead, the situation is likely to get worse due to the recent shift to parallelism in mainstream computing, triggered by the end of frequency scaling. We hope this guide conveys the problem, its origin, and a set of basic methods to write fast numerical code.

However, problems also open research opportunities. In this case the problem is the need to automate high performance library development, a difficult challenge that, in its nature, is at the core of computer science. To date this problem has been attacked mostly by the scientific computing and compiler community, and the list of successes is still short. We believe that other areas of computer science need to get involved, including programming languages, and in particular domain-specific languages, generative programming, symbolic computation, and optimization and machine learning. For researchers in these areas, we hope that this tutorial can serve as an entry point to the problem and the existing work on automatic performance tuning.

## Acknowledgment

This work was supported by DARPA through the DOI grant NBCH1050009 and the ARO grant W911NF0710416, by NSF through awards 0325687 and 0702386, and by an Intel grant.

## References

1. Moore, G.E.: Cramming more components onto integrated circuits. Readings in computer architecture, 56–59 (2000)
2. Meadows, L., Nakamoto, S., Schuster, V.: A vectorizing, software pipelining compiler for LIW and superscalar architecture. In: Proceedings of Rise (1992)
3. Group, S.S.C.: SUIF: A parallelizing & optimizing research compiler. Technical Report CSL-TR-94-620, Computer Systems Laboratory, Stanford University (May 1994)
4. Franke, B., O’Boyle, M.F.P.: A complete compiler approach to auto-parallelizing C programs for multi-DSP systems. *IEEE Trans. Parallel Distrib. Syst.* 16(3), 234–245 (2005)
5. Van Loan, C.: *Computational Framework of the Fast Fourier Transform*. SIAM, Philadelphia (1992)
6. Press, W.H., Flannery, B.P., Teukolsky, S.A., Vetterling, W.T.: *Numerical Recipes in C: The Art of Scientific Computing*, 2nd edn. Cambridge University Press, Cambridge (1992)
7. Püschel, M., Moura, J.M.F., Johnson, J., Padua, D., Veloso, M., Singer, B.W., Xiong, J., Franchetti, F., Gačić, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: SPIRAL: Code generation for DSP transforms. Proceedings of the IEEE Special issue on Program Generation, Optimization, and Adaptation 93(2), 232–275 (2005)
8. Website: Spiral (1998), <http://www.spiral.net>
9. Frigo, M., Johnson, S.G.: FFTW: An adaptive software architecture for the FFT. In: Proc. IEEE Int’l Conf. Acoustics, Speech, and Signal Processing (ICASSP), vol. 3, pp. 1381–1384 (1998)

10. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proceedings of the IEEE Special issue on Program Generation, Optimization, and Adaptation 93(2), 216–231 (2005)
11. Website: FFTW, <http://www.fftw.org>
12. Goto, K., van de Geijn, R.: On reducing TLB misses in matrix multiplication, FLAME working note 9. Technical Report TR-2002-55, The University of Texas at Austin, Department of Computer Sciences (November 2002)
13. Whaley, R.C., Dongarra, J.: Automatically Tuned Linear Algebra Software (ATLAS). In: Proc. Supercomputing (1998)
14. Moura, J.M.F., Püschel, M., Padua, D., Dongarra, J.: Scanning the issue: Special issue on program generation, optimization, and platform adaptation. Proceedings of the IEEE, special issue on Program Generation, Optimization, and Adaptation 93(2), 211–215 (2005)
15. Bida, E., Toledo, S.: An automatically-tuned sorting library. Software: Practice and Experience 37(11), 1161–1192 (2007)
16. Li, X., Garzaran, M.J., Padua, D.: A dynamically tuned sorting library. In: Proc. Int'l Symposium on Code Generation and Optimization (CGO), pp. 111–124 (2004)
17. Im, E.-J., Yelick, K., Vuduc, R.: Sparsity: Optimization framework for sparse matrix kernels. Int'l J. High Performance Computing Applications 18(1), 135–158 (2004)
18. Demmel, J., Dongarra, J., Eijkhout, V., Fuentes, E., Petitet, A., Vuduc, R., Whaley, C., Yelick, K.: Self adapting linear algebra algorithms and software. Proceedings of the IEEE Special issue on Program Generation, Optimization, and Adaptation 93(2), 293–312 (2005)
19. Website: BeBOP, <http://bebop.cs.berkeley.edu/>
20. Vuduc, R., Demmel, J.W., Yelick, K.A.: OSKI: A library of automatically tuned sparse matrix kernels. In: Proc. SciDAC. Journal of Physics: Conference Series, vol. 16, pp. 521–530 (2005)
21. Whaley, R., Petitet, A., Dongarra, J.: Automated empirical optimization of software and the ATLAS project. Parallel Computing 27(1-2), 3–35 (2001)
22. Bilmes, J., Asanović, K., whye Chin, C., Demmel, J.: Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In: Proc. Int'l Conference on Supercomputing (ICS), pp. 340–347 (1997)
23. Frigo, M.: A fast Fourier transform compiler. In: Proc. Programming Language Design and Implementation (PLDI), pp. 169–180 (1999)
24. Franchetti, F., Voronenko, Y., Püschel, M.: Formal loop merging for signal transforms. In: Proc. Programming Language Design and Implementation (PLDI), pp. 315–326 (2005)
25. Franchetti, F., Voronenko, Y., Püschel, M.: FFT program generation for shared memory: SMP and multicore. In: Proc. Supercomputing (2006)
26. Franchetti, F., Voronenko, Y., Püschel, M.: A rewriting system for the vectorization of signal transforms. In: Daydé, M., Palma, J.M.L.M., Coutinho, Á.L.G.A., Pacitti, E., Lopes, J.C. (eds.) VECPAR 2006. LNCS, vol. 4395. Springer, Heidelberg (2006)
27. Bientinesi, P., Gunnels, J.A., Myers, M.E., Quintana-Orti, E., van de Geijn, R.: The science of deriving dense linear algebra algorithms. ACM Trans. on Mathematical Software 31(1), 1–26 (2005)
28. Gunnels, J.A., Gustavson, F.G., Henry, G.M., van de Geijn, R.A.: FLAME: Formal linear algebra methods environment. ACM Trans. on Mathematical Software 27(4), 422–455 (2001)
29. Quintana-Orti, G., Quintana-Orti, E.S., van de Geijn, R., Van Zee, F.G., Chan, E.: Programming algorithms-by-blocks for matrix computations on multithreaded architectures (submitted for publication)

30. Baumgartner, G., Auer, A., Bernholdt, D.E., Bibireata, A., Choppella, V., Cociorva, D., Gao, X., Harrison, R.J., Hirata, S., Krishnamoorthy, S., Krishnan, S., Lam, C.C., Lu, Q., Nooijen, M., Pitzer, R.M., Ramanujam, J., Sadayappan, P., Sibiryakov, A.: Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proceedings of the IEEE* 93(2), 276–292 (2005); Special issue on Program Generation, Optimization, and Adaptation
31. Czarnecki, K., Eisenecker, U.: *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading (2000)
32. Lämmel, R., Saraiva, J., Visser, J. (eds.): *GTTSE 2005*. LNCS, vol. 4143. Springer, Heidelberg (2006)
33. Püschel, M.: How to write fast code. Course 18-645, Electrical and Computer Engineering, Carnegie Mellon University (2008), <http://www.ece.cmu.edu/~pueschel/teaching/18-645-CMU-spring08/course.html>
34. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C. (eds.): *Introduction to algorithms*. MIT Press, Cambridge (2001)
35. Demmel, J.W.: *Applied numerical linear algebra*. SIAM, Philadelphia (1997)
36. Tolimieri, R., An, M., Lu, C.: *Algorithms for discrete Fourier transforms and convolution*, 2nd edn. Springer, Heidelberg (1997)
37. Hennessy, J.L., Patterson, D.: *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco (2002)
38. Bryant, R.E., O’Hallaron, D.R.: *Computer Systems: A Programmer’s Perspective*. Prentice-Hall, Englewood Cliffs (2003)
39. Strassen, V.: Gaussian elimination is not optimal. *Numerische Mathematik* 14(3), 354–356 (1969)
40. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation* 9, 251–280 (1990)
41. Blackford, L.S., Demmel, J., Dongarra, J., Duff, I., Hammarling, S., Henry, G., Heroux, M., Kaufman, L., Lumsdaine, A., Petitet, A., Pozo, R., Remington, K., Whaley, R.C.: An updated set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. on Mathematical Software* 28(2), 135–151 (2002)
42. Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Croz, J.D., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: *LAPACK Users’ Guide*, 3rd edn. SIAM, Philadelphia (1999)
43. Website: ATLAS, <http://math-atlas.sourceforge.net/>
44. Website: Goto BLAS, <http://www.tacc.utexas.edu/general/staff/goto/>
45. Website: LAPACK, <http://www.netlib.org/lapack/>
46. Website: ScaLAPACK, <http://www.netlib.org/scalapack/>
47. Blackford, L.S., Choi, J., Cleary, A., D’Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.C.: *ScaLAPACK Users’ Guide*. Society for Industrial and Applied Mathematics, Philadelphia (1997)
48. Website: PLAPACK, <http://www.cs.utexas.edu/users/plapack/>
49. Chtchelkanova, A., Gunnels, J., Morrow, G., Overfelt, J., van de Geijn, R.: Parallel implementation of BLAS: General techniques for level 3 BLAS. *Concurrency: Practice and Experience* 9(9), 837–857 (1997)
50. Website: FLAME, <http://www.cs.utexas.edu/users/flame/>
51. Johnson, S.G., Frigo, M.: A modified split-radix FFT with fewer arithmetic operations. *IEEE Trans. Signal Processing* 55(1), 111–119 (2007)
52. Nussbaumer, H.J.: *Fast Fourier Transformation and Convolution Algorithms*, 2nd edn. Springer, Heidelberg (1982)

53. Johnson, J.R., Johnson, R.W., Rodriguez, D., Tolimieri, R.: A methodology for designing, modifying, and implementing FFT algorithms on various architectures. *Circuits Systems Signal Processing* 9(4), 449–500 (1990)
54. Franchetti, F., Püschel, M.: Short vector code generation for the discrete Fourier transform. In: *Proc. IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS)*, pp. 58–67 (2003)
55. Bonelli, A., Franchetti, F., Lorenz, J., Püschel, M., Ueberhuber, C.W.: Automatic performance optimization of the discrete Fourier transform on distributed memory computers. In: Guo, M., Yang, L.T., Di Martino, B., Zima, H.P., Dongarra, J., Tang, F. (eds.) *ISPA 2006. LNCS*, vol. 4330. Springer, Heidelberg (2006)
56. Website: FFTPACK, <http://www.netlib.org/fftpack/>
57. GNU: GSL <http://www.gnu.org/software/gsl/>
58. Mirković, D., Johnsson, S.L.: Automatic performance tuning in the UHFFT library. In: Alexandrov, V.N., Dongarra, J., Juliano, B.A., Renner, R.S., Tan, C.J.K. (eds.) *ICCS-ComputSci 2001. LNCS*, vol. 2073, pp. 71–80. Springer, Heidelberg (2001)
59. Website: UHFFT, <http://www2.cs.uh.edu/~mirkovic/fft/parfft.htm>
60. Website: FFTE, <http://www.ffte.jp>
61. Website: ACML, <http://developer.amd.com/acml.jsp>
62. Website: Intel MKL, <http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm>
63. Website: Intel IPP, <http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/302910.htm>
64. Website, I.B.M.: ESSL and PESSL, <http://www-03.ibm.com/systems/p/software/essl.html>
65. Website: NAG, <http://www.nag.com/>
66. Website: IMSL, <http://www.vni.com/products/imsl/>
67. Hill, M.D., Smith, A.J.: Evaluating associativity in CPU caches. *IEEE Trans. Comput.* 38(12), 1612–1630 (1989)
68. Intel Corporation: Intel 64 and IA-32 Architectures Optimization Reference Manual (2007), <http://www.intel.com/products/processor/manuals/index.htm>
69. Advanced Micro Devices (AMD) Inc.: Software Optimization Guide for AMD Athlon 64 and AMD Optero Processors (2005), <http://developer.amd.com/devguides.jsp>
70. GNU: GCC:optimization options, <http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>
71. Intel: Quick-reference guide to optimization with intel compilers version 10.x, [http://cache-www.intel.com/cd/00/00/22/23/222300\\_222300.pdf](http://cache-www.intel.com/cd/00/00/22/23/222300_222300.pdf)
72. Intel: Intel VTune
73. Microsoft: Microsoft Visual Studio
74. GNU: Gnu gprof manual, [http://www.gnu.org/software/binutils/manual/gprof-2.9.1/html\\_mono/gprof.html](http://www.gnu.org/software/binutils/manual/gprof-2.9.1/html_mono/gprof.html)
75. Yotov, K., Li, X., Ren, G., Garzaran, M.J., Padua, D., Pingali, K., Stodghill, P.: Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE Special issue on Program Generation, Optimization, and Adaptation* 93(2), 358–386 (2005)
76. Wolfe, M.: Iteration space tiling for memory hierarchies. In: *SIAM Conference on Parallel Processing for Scientific Computing* (1987)
77. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex Fourier series. *Math. of Computation* 19, 297–301 (1965)
78. Püschel, M., Singer, B., Xiong, J., Moura, J.M.F., Johnson, J., Padua, D., Veloso, M., Johnson, R.W.: SPIRAL: A generator for platform-adapted libraries of signal processing algorithms. *Int'l Journal of High Performance Computing Applications* 18(1), 21–45 (2004)

79. D'Alberto, P., Milder, P.A., Sandryhaila, A., Franchetti, F., Hoe, J.C., Moura, J.M.F., Püschel, M., Johnson, J.: Generating FPGA accelerated DFT libraries. In: Proc. Symposium on Field-Programmable Custom Computing Machines (FCCM) (2007)
80. Milder, P.A., Franchetti, F., Hoe, J.C., Püschel, M.: Formal datapath representation and manipulation for implementing DSP transforms. In: Proc. Design Automation Conference (DAC) (2008)
81. Xiong, J., Johnson, J., Johnson, R., Padua, D.: SPL: A language and compiler for DSP algorithms. In: Proc. Programming Language Design and Implementation (PLDI), pp. 298–308 (2001)
82. Dershowitz, N., Plaisted, D.A.: Rewriting. In: Handbook of Automated Reasoning, vol. 1, pp. 535–610. Elsevier, Amsterdam (2001)

# A Gentle Introduction to Multi-stage Programming, Part II\*

Walid Taha

Department of Computer Science, Rice University, Houston, TX, USA  
taha@rice.edu

**Abstract.** As domain-specific languages (DSLs) permeate into mainstream software engineering, there is a need for economic methods for implementing languages. Following up on a paper with a similar title, this paper focuses on dynamically typed languages, covering issues ranging from parsing to defining and staging an interpreter for an interesting subset of Dr. Scheme. Preliminary experimental results indicate that the speedups reported in previous work for smaller languages and with smaller benchmarks are maintained.

## 1 Introduction

A natural question to ask when we consider implementing a new language is whether an existing language implementation can be reused. If reuse is possible, then we have reduced the new problem to one that we have (or someone else has) already solved. The reduction process materializes as a translator from the new language to the old. In “A Gentle Introduction to Multi-stage Programming,” [8] we introduced the reader to the basics of a semantically inspired approach to building such translators, namely, a staged interpreter. We refer to that paper as Part I. It introduces the basic approach consisting of three steps:

1. Write an interpreter and check its correctness.
2. Stage the interpreter by adding staging annotations.
3. Check the performance of the staging implementation.

Based on practical experience, when we get to the third step we should expect that we need to convert to continuation-passing style to achieve satisfactory staging.

The focus of Part I was a simple language called Lint, which has only one type, namely integers, and supports only functions from one integer to another integer. The goal of this paper (Part II) is to expand the reader’s repertoire of staging expertise to the point where he or she can implement a dynamically typed language as expressive as a substantial subset of the Dr. Scheme programming language.

---

\* Supported by NSF CCR SoD 0439017, CSR/EHS 0720857, and CCF CAREER 0747431.



## 1.1 Prerequisites

To follow the explanation of the reference interpreter (Section 3), basic familiarity with OCaml, lambda abstractions, and the OCaml List library are needed.

Some familiarity with continuation-passing style (CPS) and CPS conversion is useful but not necessary. Section 4 explains the conversion process and provides a detailed discussion of how to convert an interpreter. Friedman, Wand, and Haynes' text 3 explains how to perform CPS conversion, and Sabry's thesis 6 provides an accessible introduction to its meta-theory.

Familiarity with the basics of multi-stage programming and with the basics of staged interpreters is needed for Section 5. For a reader not familiar with these topics, "A Gentle Introduction to Multi-stage Programming," 8 would provide the necessary background.

## 1.2 Contributions

The key novelties driving this tutorial paper are the scale and the type discipline of the language being interpreted. We focus on a larger language than has previously been analyzed in the context of writing staged interpreters, and the focus is on the sources of difficulty that arise during the process of building a staged interpreter for this language.

Our focus is on a class of languages that is both increasingly popular in practice and is (often surprisingly) easy to interpret in a modern, statically typed, functional language: dynamically typed languages. Writing such an interpreter is facilitated by the fact that we can easily define one data type that would serve as the universal value domain.

The new expository material presented in this paper includes the following:

1. The use of a small, practical parser combinator library as well as the use of universal concrete and abstract syntax to facilitate practical language prototyping. We have found that the absence of a default starting point for building simple parsers and doing simple file IO is a practical problem for someone trying to build a staged interpreter in MetaOCaml. This is especially the case when one has background in languages other than OCaml.
2. Case-by-case analysis of an OCaml interpreter for a more expressive language than the one we covered in Part I. This language includes higher-order functions with multiple arguments, a dynamic data structure (lists), and mutable structures.
3. Detailed explanation of how to translate a direct style interpreter into CPS.

## 1.3 Organization

Section 2 introduces a method that we have found useful for quickly building parsers. As that section points out, a practical method for circumventing the issue of building parsers and designing concrete syntax for a programming language is to use a universal syntax, such as that of HTML, XML, or the LISP syntax for representing data (s-expressions). Section 3 presents an interpreter for a

basic subset of Dr. Scheme that we call Aloe (Another Language of Expressions). This subset includes booleans, integers, strings, mutable variables, mutable lists, and higher-order functions. This section explains how to interpret variable arity operators and functions, syntactic sugar, and how to use side effects to implement recursive definitions. Section 4 explains how to convert the staged interpreter into CPS, giving special attention to the more interesting cases. Section 5 explains how to stage the interpreter that resulted from the previous step. Section 6 presents a new optimization technique for untyped languages, and shows how this technique can be used to improve the Aloe staged interpreter.

The complete code for the parsers and the various interpreters and test inputs described in this paper, along with other examples, are available online at <http://www.metaocaml.org/examples>.

## 2 Parsing

Concrete syntax tends to be much more concise than abstract syntax, so it is notationally convenient. Furthermore, if we want to be able to store and load programs in files, it must be done with some concrete syntax. To accommodate this practical concern, this section will provide the reader with a minimal tool for dealing with the issue of parsing.

In the source code accompanying this paper, we use Hutton and Meijer’s parser combinators [4]. We have reimplemented this library in OCaml for convenience, and the mapping is mostly mechanical. The key change is that Haskell is lazy and so allows pure, memoized streams to be implemented concisely. We simply used lambda abstraction to delay computations. Certainly, memoizing implementations are possible [10], but the examples considered here are simple enough that there is no pressing need for this optimization.

To avoid having to define a parser for every new language that we consider, it is helpful to use a single, universal concrete representation for programs. Representing programs concretely (where “concretely” means “as strings of characters”) is no different from representing any other form of data. Universal concrete representations include HTML and the XML and the LISP data formats (widely known as s-expressions). Because it is syntactically lighter-weight, we use s-expressions for this paper. Specifically, we use the following grammar for s-expressions:

$$\text{s-expression} ::= \text{integer} \mid \text{symbol} \mid \text{string} \mid (\text{s-expression}^*)$$

where integer, symbol and string are primitive grammars for integers, symbols, and strings, respectively; and where  $e^*$  means zero or more repetitions of  $e$ . We use the combinator library described above to write a single parser for s-expressions. Successful parsing produces a value of the following OCaml type:

```
type sxp =
  | I of int      (* Integer *) | A of string  (* Atoms *)
  | S of string  (* String *)  | L of sxp list (* List *)
```

Whereas s-expressions are a universal concrete syntax, this data type can be viewed as a universal abstract syntax. Our interpreters for Aloe always take a value of this type as input.

To illustrate how s-expressions using the traditional concrete syntax would be represented in the OCaml type `sxp`, we consider a few small examples. The list `(1 2 3 4)` would be represented by the value

```
L [I 1; I 2; I 3; I 4]
```

The type naturally allows for nesting of lists. For example, `((1 2) (3 4))` would be represented as

```
L [L [I 1; I 2]; L [I 3; I 4]]
```

The type also naturally allows us to mix elements of different types, so we can represent `(lambda (x) (repeat x \"ping \"))`

```
L [A "lambda"; L [A "x"]; L [A "repeat"; A "x"; S "ping "]]
```

To automate the process of parsing an s-expression, we provide the following utilities:

```
read_file : string -> string
parse     : string -> (sxp * string) list
print     : sxp -> string
```

The first function simply takes the name of a file and reads it into a string. The second function takes a string and tries to parse it. It returns a list of possible ways in which it could have been parsed, as well as the remaining (unparsed) string in each case. For s-expressions, we always have at most one way in which a string can be parsed. The third function takes an s-expression and returns a string that represents it.

### 3 An Interpreter for Aloe

This section presents an interpreter for a small programming language that we call Aloe. This language is the running example for this paper, and the interpreter drives the discussion of CPS conversion and staging.

The Aloe programming language is a subset of Dr. Scheme that includes booleans, integers, strings, mutable variables, mutable lists, and higher-order functions. We begin the design of the interpreter by considering the appropriate definitions for values and environments, and then move to defining the interpreter itself. We will conclude this section by describing a timing benchmark, a timing experiment, and the results from this timing experiment. These results serve as the baseline for assessing the performance of the staged interpreter.

#### 3.1 Denotable Values and Tags

The first question to consider when writing an interpreter for an untyped language is to determine the kinds of values that the language supports. For Aloe, we are interested in the following kinds of values:

```

type dom = Bool of bool | Int of int | Str of string
         | Fun of int * (dom list -> dom) | Undefined
         | Void | Empty | Cons of dom ref * dom ref

```

Thus, our language supports three interesting base types: booleans, integers, and strings. It also supports functions and mutable lists. Each function value is tagged with an integer that represents the number of arguments it expects. In addition, the set of values includes two special values, `Undefined` and `Void`. The first special value is used for un-initialized locations, and the second for denoting the absence of a result from a side-effecting computation.

It is worth noting that the type we have defined for values is *computational* and not a purely mathematical type. This allows us to avoid having to specify explicitly where non-termination or exceptions can occur. It also makes it easy for us to represent values that can change during the lifetime of a computation by using the OCaml `ref` type constructor.

### 3.2 Exceptions and Untagging

To allow for the possibility of error during the computation of an Aloe program, we introduce one OCaml exception:

```

exception Error of string

```

and we immediately use this exception to define specific untagging operations that allow us to extract the actual value from a tagged value when we expect a certain tag to be present. For example, for the `Fun` tag we write:

```

let unFun d =
  match d with
  | Fun (i,f) -> (i,f)
  | _ -> raise (Error "Encountered a non-function value")

```

### 3.3 Environments and Assignable Variables

We represent environments simply as functions, where the empty environment `env0` produces an error on any lookup.<sup>1</sup> We define both a simple environment extension function `ext` that introduces one new binding, as well as one that extends the environment with several bindings at a time `lext`:

```

let env0 x =
  raise (Error ("Variable not found in environment "^x))

let ext env x v = fun y -> if x=y then v else env y

```

<sup>1</sup> Some readers are surprised by the use of functions to represent environments, rather than using a first-order collection type. When studying programming language semantics, and especially denotational or translational semantics, it is common to think of environments as simply being functions.

```

let rec lext env xl vl =
  match xl with
  | [] -> env
  | x::xs -> match vl with
    | [] -> raise (Error "Not enough arguments")
    | y::ys -> lext (ext env x y) xs ys

```

An important technical aside for Aloe is that, being a subset of Dr. Scheme, some but not all variables can be assigned. For example, functional arguments are immutable. To reflect this difference, we require that all environments map names to values of the following type:

```
type var = Val of dom | Ref of dom ref
```

### 3.4 Concrete Syntax

As noted earlier, we use the OCaml data type for s-expressions to represent the abstract syntax for our programs. Nevertheless, it is still useful to state the concrete syntax for the language that we are interested in.

```

I    is the set of integers
S    is the set of strings
A    is the set of symbols (“A” for “Atomic”)

U ::= not | empty? | car | cdr

B ::= + | - | * | < | > | = | cons | set-car! | set-cdr!

E ::= true | false | empty | I | "S" | A | (U E) | (B E E) |
      (cond (E E)(else E)) | (set! A E) | (and E E*) | (or E E*)
      | (begin E E*) | (lambda (A*) E) | (E E*)

P ::= E | (define A E)P | (define (A A*) E)P

```

The first three lines indicate that we assume that we are given three sets, one for integers, one for strings, and one for symbols (or “atoms”). Integers are defined as sequences of digits possibly preceded by a negative sign. Strings are sequences of characters with some technical details the definition of which we relegate here to the implementation. Symbols are also sequences of characters, with the most notable restriction being the absence of spaces.

The next line defines the set  $U$  of unary operator names. This set consists of four terminal symbols. The next set  $B$ , which consists of nine terminals, is the names of binary operators. The set of expressions  $E$  contains terminals to denote booleans and the empty list, and it also embeds integers, strings, and symbols. When we get to symbols, the user should note that there is a possibility for ambiguity here. The expression `true`, for example, can match either the first or the sixth (symbol) production. For this reason we consider our productions order

dependent, and the derivation of a string always uses the production that appears left-most in the definition. This does not change the set of strings defined, but it makes the derivation of each string unique. The significance of the order of production is also important for ensuring the proper behavior from the main case analysis performed in the interpreter.

The last line defines the set of programs  $P$ . A program can be an expression, or a nested sequence of variable or function definitions.

*Note 1 (Practical Consideration: Validating the Reference Interpreter).* We caution the reader that even though Aloe is a small language, we spent considerable time removing seemingly trivial bugs from the first version of the interpreter. Staging provides no help with this problem, and in fact any problems present in the original interpreter and that go unnoticed are also present in the staged interpreter. Thus, we strongly recommend developing an extensive acceptance test that illustrates the correct functionality of each of the language constructs while developing the original interpreter. Such tests will also be useful later when developing the staged interpreter and when assessing the performance impact of staging.

A helpful by-product of using both the syntax and the semantics of Dr. Scheme for Aloe was that it was easy to validate the correct behavior of our test examples using the standard Dr. Scheme implementation. Because the correctness of language implementations is of such great importance, the benefits of devising a new syntax for your language should be weighed carefully against the benefits of having such a direct method of validating the implementation. This is not just relevant in cases when we are evaluating new implementation technology such as staged interpreters. Consider how the development of ML implementations could have been different if it used Scheme syntax, or XML if it used s-expression syntax. Change in syntax can often impede reuse and obfuscate new ideas.

### 3.5 The Interpreter for Expressions

The Aloe interpreter consists of two main functions, one interpreting expressions, and the other interpreting programs. To follow the order in which the syntax is presented, we start by covering the interpreter for expressions, which takes an expression and an environment and returns a value. It is structured primarily as a `match` statement over the input expression. The `match` statement is surrounded by a `try` statement, so that exceptions are caught immediately, some informative debugging information is printed, and the exception is raised again. Thus, the overall structure of the interpreter is as follows:

```
let rec eval e env =
  try (match e with
       ...)
  with
    Error s -> (print_string ("\n"^(print e)^\n");
                raise (Error s))
```

where the ... represents the body of the interpreter. The reader should note that we choose to have the interpreter traverse the syntax represented by s-expressions to save space. This choice makes the different branches of the match statement order sensitive. Thus, this tutorial does trade good interpreter design for presentation space. The reader interested in better interpreter design is encouraged to consult a standard reference [3].

In the rest of this section, we discuss the key issues that arise when interpreting the various constructs of Aloe.

*Note 2 (Practical Consideration: Reporting Runtime Errors).* Our Aloe interpreter provides minimal contextual information to the user when reporting runtime error. More comprehensive reporting about errors as well as debugging support would not only be useful to the users of the language but would also help the implementor of the language as well. Thus, these issues should be given careful consideration when implementing any language of size comparable to or larger than the Aloe.

**Atomic Expressions, Integers, and Strings.** The semantics of most atomic expressions in Aloe are fairly straightforward:

```
| A "true"   -> Bool true | A "false"  -> Bool false
| A "empty" -> Empty
| A x -> (match env x with Val v -> v | Ref r -> !r)
| I i -> Int i | S s -> Str s
```

Booleans, the empty list, integers, and strings are interpreted in a direct manner. Variables are defined as any symbols that did not match the first three cases in the `match` statement are interpreted by first looking up the name in the environment. Because an environment lookup might fail, it is possible that an exception may be raised when we apply `env` to the string `x`. If we do get a value back, the interpreter checks to see whether it is assignable or not. If it is a simple value, we return it directly. If it is an assignable value, then we de-reference the assignable value and return the value to which it is pointing.

**Unary and Binary Operators.** The interpretation of unary and binary operators is a bit more involved, but still largely direct. It should be noted, however, that we have to explicitly define the action of all primitive operations somewhere in our interpretation. For convenience this can be done inline, as we do for each case in our case analysis:

```
| L [A "not"; e1]   -> Bool (not (unBool (eval e1 env)))
| L [A "+"; e1; e2] -> Int ( (unInt (eval e1 env)
                             + (unInt (eval e2 env))))
```

Interpreting logical negation in Aloe is done by evaluating the argument, removing the `Bool` tag, applying OCaml's logical negation to the resulting value, and then tagging the resulting value with `Bool`. The pattern of untagging and re-tagging repeats in our interpreter, which is a necessity when being explicit about

the semantics of a dynamically typed language. Because OCaml is statically typed, this forces us to make the tag manipulation explicit. While this may seem verbose at first, we see in later discussion that being explicit about tags may be convenient for discussing different strategies for implementing the same dynamically typed computation.

The binary operation of addition follows a similar pattern, as do most of the unary and binary operations in Aloe. An interesting binary operation is the `cons` operation, which creates a new list from a new element and an old list:

```
| L [A "cons"; e1; e2] ->
  Cons (ref (eval e1 env), ref (eval e2 env))
```

This implementation of the list constructor is sometimes described as being *unchecked*, in that it does not check that the second element is a list. Another interesting case is mutation, namely, of the `set-car!` or `set-cdr!` of a list. Both operations are interpreted similarly. The first is interpreted as follows:

```
| L [A "set-car!"; e1; e2] ->
  (match (eval e1 env) with
  | Cons (h,t) -> (h:=(eval e2 env);Void)
  | _ -> raise (Error ("Can't assign car of "
    ^ (print e1))))
```

Performing this computation first evaluates the first argument, checks that it is a non-empty list, and if so, evaluates the second expression and assigns the result to the head of the list. If the first value is not a non-empty list, an error is detected. For `set-cdr!`, the same is done, and the tail of the list is modified.

**Variable Arity Constructs.** It is not unusual for programming language constructs to allow a varying number of arguments. An example of such a variable arity construct is the equality construct `=` which takes one or more arguments and returns “true” only if all of them are equal integers. We interpret this construct as follows:

```
| L ((A "=") :: e1 :: l) ->
  Bool (let v = unInt (eval e1 env) in
    let l = List.map (fun x -> unInt (eval x env)) l
    in List.for_all (fun y -> y=v) l)
```

First, the first expression is evaluated, and its integer tag is removed. This reflects the fact that this operator is intended to work only on integer values. Then, we map the same operation to the elements of the rest of the list of arguments. Finally, we check that all the elements of that list are equal to the first element.

Logical conjunction and disjunction are implemented in a similar manner.

**Conditionals and Syntactic Sugar.** Conditional expressions are easy to define. However, we limit them to having two arguments and leave the generalization as an exercise to the reader in applying the variable arity technique presented above.



We use conditionals to illustrate how to deal with syntactic sugar. In particular, Aloe includes `if` statements, which can be interpreted by a recursive call to the interpreter applied to the de-sugared version of the expression:

```
| L [A "if"; c2; e2; e3] ->
  eval (L [A "cond"; L [c2 ; e2]; L [A "else"; e3]]) env
```

The key issue that requires attention using this technique to support syntactic sugar is that we should make sure that it does not introduce non-termination. While this may seem inconsequential in a setting in which the language being interpreted can itself express diverging computation, the distinction between divergence in the interpretation and divergence in the result of the interpretation will become evident when we stage such interpreters.

**Lambda Abstraction.** If lambda abstractions in Aloe were allowed only to have one argument, then the interpretation of lambda abstractions would be expressible in one line:

```
| L [A "lambda" ; L [S x] ; e] ->
  Fun (1, fun l -> match l with
      [v] -> eval e (ext env x (Val v)))
```

The pattern matching requires that there is only one argument, that it is a string, and that the value of that string is bound to `x`. The pattern also requires that this be followed precisely by one expression. The interpretation returns a value tagged with the `Fun` tag. The first component of this value represents the number of arguments that this function expects (in this case one). The second argument is an OCaml lambda abstraction that takes a value and pattern matches it to assert that it is a list of one element, which is called `v`. The result of the OCaml lambda abstraction is the result of evaluating the body of the Aloe lambda abstraction, namely, the expression `e`. Evaluation of this expression occurs under the environment `env` extended with a mapping from the name `x` to the value `Val v`. We tag the value `v` with the tag `Val` to reflect the fact that we do not allow the arguments to lambda abstractions to be mutated.

To handle the fact that lambda abstractions in Aloe can handle multiple arguments, the interpretation becomes a bit more verbose, but in reality it is essentially doing little more than what the case for one argument is doing:

```
| L [A "lambda" ; L axs ; e] ->
  let l = List.length axs
  in Fun (l, fun v ->
      eval e (lxt env
              (List.map (function A x -> x) axs)
              (List.map (fun x -> Val x) v)))
```

Here, we are simply allowing the argument to be a list of names and handling this extra degree of generality by mapping over lists and using the function for extending the environment with a list of mappings (introduced earlier).

**Function Application.** Function application is similarly easier to understand if we first consider only the case of single-argument functions:

```
| L [e1; e2] -> let (1,f) = unFun (eval e1 env) in
                 let arg = eval e2 env
                 in f [arg]
```

The pattern match assumes that we only have an application of one expression to another. The first `let` statement evaluates the first expression, removes the `Fun` tag, checks that the first component is 1, and calls the second component `f`. The second `let` statement evaluates the argument expression, and calls the resulting value `arg`. While these two `let` statements can be interchanged, the order is highly significant in a language that allows side effects, as Aloe does. Finally, the result of the interpretation is simply the application of the function `f` to the singleton list `[arg]`. The interpretation for function application in full generality is as follows:

```
| L (e::es) ->
  let (i,f) = unFun (eval e env) in
  let args = List.map (fun e -> eval e env) es in
  let l = List.length args
  in if l=i
     then f args
     else raise (Error ("Function has "^(string_of_int l)^
                        " arguments but called with "^(string_of_int i)))
```

This generalization also evaluates the operator expression first, then it uses a list map to evaluate each of the argument expressions. Once that has been done, the arguments are counted, and we check that the number of arguments we have is consistent with the number of arguments that the function expects. If that is the case, then we simply perform the application. Otherwise, an error is raised.

### 3.6 The Interpreter for Programs

The interpreter for Aloe programs takes a program and an environment and produces a value. Compared with expressions, Aloe programs are relatively simple, and thus the interpreter for programs can be presented in one-shot, as follows:

```
let rec peval p env =
  match p with
  | [e1] -> eval e1 env
  | (L [A "define"; A x; e1])::p ->
    let r = ref Undefined in
    let env' = ext env x (Ref r) in
    let v = eval e1 env' in
    let _ = (r := v)
    in peval p env'
  | (L [A "define"; L ((A x)::xs); e1])::p ->
```

```

peval (L [A "define"; A x;
         L [A "lambda" ; L xs ; e1]]::p) env
| _ -> raise (Error "Program form not recognized")

```

The first case, the case in which a program is simply an expression, is easy to recognize because it is the only case in which a program is a singleton list. In that case, we simply use the interpreter for expressions. The second case is a `define` statement. We wish to interpret all `define` statements as recursive, and there are many ways in which this can be achieved. In this interpreter, we employ a useful trick that can provide an efficient implementation in the presence of side effects—in the interpretation, we create a reference cell initialized to the special value `Undefined`. Then, we create an extension of the current environment mapping the variable that we are about to define to this reference. Next, we evaluate the body of the reference. Finally, we update the reference with the result of the evaluation of the body and continue the evaluation of the rest of the program in the extended environment. Clearly, this technique only produces a useful value if the definition of the variable that we are about to define is not *strict* in its use of that variable. This is generally the case, for example, if the definition is a lambda abstraction or an operation that produces a function value (which are often used, for example, to represent lazy streams).

The last case of the interpreter produces an error if any other syntactic form is encountered at the top level of a program.

With this, we have completed our overview of the key features of the reference interpreter for the Aloe language.

### 3.7 A Benchmark Aloe Program

To collect some representative performance numbers, we use one Aloe program that defines and uses a collection of functions, including a number of alternative definitions for the factorial, Fibonacci, and Takeuchi (`tak`) functions using numbers and lists to perform the core computation, as well as a CPS version of the insertion sort. As a sanity check for the correctness of the implementation, the suite includes a function that applies these various functions to different inputs and compares the output. To facilitate the use of the suite for performance evaluation, the main function executes this test 1000 times. For simplicity, all the functions that form the test suite are included in one file called `test.aloe`.

### 3.8 The Experiment

To study their relative performance, the same experiment is carried out for the interpreter and for the staged versions of the interpreter. The code for the experiment for the interpreter described above is as follows:

```

let test1 () =
  let f = "test.aloe" in
  let _ = Trx.init_times () in
  let s = Trx.time 10 "read_file" (fun () -> read_file f) in

```

```

let [(L p,r)]
    = Trx.time 10 "parse" (fun () -> parse ("(^s^")) in
let a = Trx.time 1 "peval"      (fun () -> peval p env0) in
let a = Trx.time 1 "readeval"  (fun () -> freadeval f) in
let _ = Trx.print_times ()
in a

```

The functions `Trx.init_times`, `Trx.time`, and `Trx.print_times` are all standard functions that come with MetaOCaml, and they are provided to assist with timing. The function `freadeval` performs all the steps in one shot. The experiment times ten different readings of the file into a string, ten parsings of the string into an abstract syntax tree, a single evaluation of the parse program, and a combined run through of all of these steps.

### 3.9 Benchmarking Environment

All results reported in this paper are for experiments performed on a machine with the following specifications: MacBook running Mac OS X version 10.4.11, 2 GHz Intel Core 2 Duo, 4 MB L2 cache per processor, 2 GB 667 MHz DDR2 DRAM. All results were collected using MetaOCaml version 3.09.1 alpha 030 using the interactive top-level loop.

#### 3.10 Baseline Results

The results of the first experiment are as follows:

```

# test1 ();;
__ read_file _____ 10x avg = 1.982000E - 01 ms
__ parse _____ 10x avg = 6.398430E + 01 ms
__ peval _____ 1x avg = 1.408170E + 04 ms
__ readeval _____ 1x avg = 1.417668E + 04 ms

```

For this baseline implementation, reading the file is fast, parsing is a little bit slower, but evaluation has the dominant cost.

## 4 Converting into Continuation-Passing Style (CPS)

Consel and Danvy [1] recognized the utility of CPS converting programs before they are partially evaluated. The same is observed for CPS converting programs before they are staged [7,9]. Intuitively, having a program in CPS makes it possible to explore specialization opportunities in all branches of a conditional statement even when the condition is not statically known.

We begin this section with a brief review of CPS conversion, and then proceed to discussing the CPS conversion of the interpreter that we developed above.

## 4.1 CPS Conversion

Consider the Fibonacci function, which can be implemented in OCaml as follows:

```
let rec fib n = if n < 2 then n
                else fib (n-1) + fib (n-2)
```

This function has type `int -> int`. Converting this function into CPS yields the following:

```
let rec fib_cps n k = if n < 2 then k n
                      else fib_cps (n-1)
                          (fun r1 -> fib_cps (n-2)
                              (fun r2 -> k
                                  (r1 + r2)))

let k0 = fun r -> r
let fib n = fib_cps n k0
```

Where `fib_cps` is a function of type `int -> (int -> 'a) -> 'a`. This new code is derived as follows.

**Functions Get an Extra Parameter.** We add the extra parameter `k` to the function that we are converting. This parameter is called the *continuation*, and its job is to process whatever value was being simply returned in the original function. So, because the original function returns a value of type `int`, the continuation has type `int -> 'a`. This type confirms that the continuation is a function that expects an integer value. It also says that the continuation can return a value of any type: CPS conversion does not restrict what we do with the value after it is “returned” by applying the continuation to it. Note, however, that the final return value of the new function (`fib_cps`) is also `'a`. In other words, whatever value the continuation returns, it is also the value that is returned by the converted function.

**Only the Branches in Conditionals are Converted.** The `if` statement is converted by converting the branches. In particular for the purposes of staging interpreters, the condition need not be converted, and can stay the same as before.

**Simple Values are Returned by Applying the Continuation.** Any simple value such as a constant, a variable, or any value that does not involve computation is converted by simply applying the continuation parameter to it. Thus, the true branch of the conditional is converted from being `n` to `k n`.

**For Composite Computations, Convert Sub-Expressions.** The most interesting part of the Fibonacci example is the `else` branch, as it has two function calls and an addition operation. Converting any such composite computation proceeds by identifying the first expression to be evaluated and creating a continuation (a function) which spells out what is done after the first computation is done. Technically, identifying the first computation requires familiarity with

the evaluation with the order of evaluation in our language. For the purposes of staging, any reasonable choice, independently of the actual language semantics, seems to work fine. In our example, we consider the left-most function call to be the first one. Thus the converted code for this branch starts with `fib_cps (n-1) ...`. The rest of the code for this branch builds the continuation for this computation. In general, when we are building a new continuation during CPS conversion, we create a lambda abstraction that names the value that is passed from the result of the first computation to this continuation for further processing. This explains the new code up to the level of `(fun r1 -> ...)`. In building the body of the lambda abstraction of this continuation, we simply repeat the process for the rest of the code in the original function. The next “first” computation is the right-most function call, and so the continuation begins with `fib_cps (n-2) ...`. Then we need to construct a continuation for this computation. All that remains at this point is add the two values `r1` and `r2` and apply the continuation `k` to this sum.

**Using a CPS Function as a Regular Function.** The last two lines of the example above show how a function such as `fib_cps` can be packaged up to behave to the external world just as the original function did. All that is needed is to construct an initial continuation `k0` that simply takes its argument and returns it unchanged. Then, we define `fib` as a function that calls `fib_cps` with the same argument and the initial continuation.

**Indentation of CPS Code.** For consistency and clarity, a particular indentation style is often used to make code written in CPS easier to read. For example the code above is written as follows:

```
let rec fib_cps n k = if n<2 then k n
                      else fib_cps (n-1) (fun r1 ->
                                           fib_cps (n-2) (fun r2 ->
                                                           k (r1 + r2)))
```

Writing the code in this indentation style allows us to make a half-accurate, semi-formal pun with the following code:

```
let k r = r
let rec fib n = if n<2 then k n
                else let r1 = fib (n-1) in
                     let r2 = fib (n-2) in
                     k (r1 + r2)
```

**Where to Stop Converting.** Often, we will find that full CPS conversion is not necessary. In our experience, it is enough to start by converting the main interpreter program, and only convert helper functions as needed. In general, the functions that need to be converted are the functions that need to be in CPS for all the recursive calls to the main interpreter program to be in CPS.

## 4.2 Effect of CPS Conversion on the Type of the Interpreter

As noted in Part I, it is generally useful to convert a program into CPS before staging it. To convert our interpreter for the expressions

```
eval : sxp -> (string -> var) -> dom
```

into CPS requires systematically rewriting the code to

1. Take in an extra “continuation” argument in every function call, and to
2. apply this continuation to every value that we would normally simply return in the original code.

This yields a new function

```
keval : sxp -> (string -> var) -> (dom -> dom) -> dom
```

## 4.3 CPS Converting the Interpreter

We now turn to CPS converting the code of an interpreter similar to the one for Aloe. We conclude the section by reporting the results of running our timing experiments on the CPS-converted interpreter.

## 4.4 Taking in the Continuation

Only minor change is needed to the outer-most structure of the interpreter for expressions:

```
let rec keval e env k =
  try
    (match e with
     ...)
  with Error s -> (print_string ("\n"^(print e)^\n");
                  raise (Error s))
```

In the first line, we have added an extra parameter `k`, through which we pass the continuation function. This is the function that we apply in the rest of the interpreter to every value that we simply returned in the original interpreter.

A natural question to ask when we consider the next line is: why not simply add an application of `k` around the `try` statement, and be done with the conversion to CPS? While this would be valid from the point of view of external behavior, to achieve our goal of effective staging, it is important that we push the applications of the continuation `k` as far down as possible to the leaves of our program. In particular, this means that we push the applications down over `try` statements and `match` statements.

A useful observation to make at this point is that pushing this single application of the continuation from around a `match` statement duplicates this application around all the branches. While this duplication is inconsequential in normal evaluation of a `match` statement, it is significant when evaluating the staged version of a `match` statement.

## 4.5 Cases That Immediately Return a Value

When we get to the branches of the `match` statement, the simple cases in which the interpretation returns a value without performing an interesting computation, the CPS version of this code simply applies the continuation `k` to this value as follows:

```
| A "true"   -> k (Bool true)
| A "false"  -> k (Bool false)
| A "empty"  -> k (Empty)
```

The cases for integer and string literals are similar.

## 4.6 Match Statements and Primitive Computation

The case for variables allows us to illustrate two points. First, if we encounter another nested `try`, `match`, or `if` statement, we simply push the continuation to the branches:

```
| A x ->
  (match env x with
   | Val v -> k v
   | Ref r -> k (! r))
```

In the second branch, we also notice that even though the expression `! r` is a computation (and not a value) that consists of applying the de-referencing function `!` to the variable `r`, we leave this expression intact and simply apply the continuation `k` to its result. For general function applications, we see that this is not the case. However, for primitive functions such as de-referencing we simply leave their application intact in the CPS-converted program. The use of primitive functions is seen in several other cases in the interpreter, including logical negation, arithmetic operations, and many others.

## 4.7 Simple, Non-primitive Function Calls

Function calls are converted by replacing the call to the original function with a call to the new function. Because our goal is to push CPS conversion as deeply as possible, we assume that we have already CPS-converted the function being applied. An easy special case is when the function we are converting is an application of a function inside its definition (we are converting a recursive call inside a recursive definition), we do both things at the same time.

Converting a function call involves providing a continuation at the call site, and this requires some care. Passing the current continuation `k` to the recursive call would mean that we simply want the result of the current call to be used as the rest of the computation for the recursive call. This would only be correct if we were immediately returning result of this recursive call. In our interpreter, this situation arises only in the case of our interpretation of syntactic sugar, such as our interpretation of the `if` statement in terms of the `cond` statement:





The converted code is indented to suggest a particular, convenient way of reading the code. In particular, while the expression `(fun r1 -> ...` only ends at the very end of the statement, and even though this whole expression is technically being passed to the first application of `keval` as a argument, we know how this argument is going to be used: it is applied to the result of the `keval e1 env` computation. This means that we can read the code line by line as follows:

1. Apply `keval` to `e1` and `env`, and “call” the result `r1`. The name `r1` is used in the following lines to refer to this value,
2. Apply `keval` to `e2` and `env`, and “call” the result `r2`, and finally
3. “Return” or “continue” with the value `Int ((unInt r1) + (unInt r2))`.

As we gain more familiarity with CPS-converted code, we find that this reading is both accurate and intuitive. The reader would be justified in thinking that CPS conversion seems to add a somewhat imperative, step-by-step feel to the code.

#### 4.10 Passing Converted Functions to Higher-Order Functions

The equality construct (Subsection [3.5](#)) allows us to illustrate two important issues that arise with CPS conversion. This section addresses the first issue, which concerns what conversion should do when we are passing a converted function as an argument to another (higher-order) function.

For example, the reference interpreter creates functions that internally make calls to the interpreter `eval` and passes them to `map` so that they can be applied to a list of arguments. What continuation do we pass to these calls? Clearly, passing the current continuation to each of these elements would not be appropriate: it would have the effect of running the rest of the computation on each of the elements of the list as a possible, alternate, result. In fact, generally speaking, the result of CPS should only apply the current continuation exactly once. Only in situations where we are essentially backtracking do we consider sequentially applying the same continuation more than once. A practical alternative for what to pass to `eval` in this situation would be to pass the identity function as the continuation. This is not unreasonable, but passing the identity function as the continuation essentially means that we are locally switching back to direct style rather than CPS. The most natural way to convert the expression that maps the interpretation function to the list elements is to change the `map` function itself to accommodate functions that are themselves in CPS, as follows:

```
let rec kmap f l k = match l with
  | [] -> k []
  | x::xs -> kmap f xs (fun r1 ->
    f x (fun r2 ->
      k (r2::r1)))
```

We need to not only change the list but also to really push the CPS conversion process through. In addition we need to replace the use of `List.for_all` by a

function that follows CPS. Fortunately, there is no need to rewrite the whole `List.for_all` function; instead, we can rewrite it using the `List.fold_left` function. Thus, CPS converting the code for the equality construct we get:

```
| L ((A "=") :: e1 :: l) ->
  keval e1 env (fun r1 ->
    let v = unInt r1
    in kmap (fun x k -> keval x env (fun r -> k (unInt r))) l
      (fun r ->
        k (Bool (List.fold_left
          (fun bc nc -> (bc && nc=v))
          true r))))
```

where the `List.fold_left` application realizes the `List.for_all` application in the original code.

Before proceeding further, we recommend that the reader work out the derivation of this code from the original, direct-style code. While converting an expression into CPS it is useful to keep in mind that pushing the conversion as deep into the code as possible will generally improve the opportunities for staging.

#### 4.11 Needing to Convert Libraries

The second issue that the equality construct allows us to illustrate is an undesirable side-effect of CPS conversion: converting our code may require converting libraries used by the code as well. In our experience, this has only been a limited problem. In particular, interpreters tend to require only a few simple library routines for the interpretation itself rather than for the operations performed by or on the values interpreted. Again, this distinction becomes clear when we consider staging the interpreter.

#### 4.12 Lambda Abstraction

Except for one interesting point, the case for lambda abstraction is straightforward:

```
| L [A "lambda" ; L axs ; e] ->
  let l = List.length axs
  in k (Fun (l, fun v ->
    keval e (lctx env
      (List.map (function A x -> x) axs)
      (List.map (fun x -> Val x) v))
      (fun r -> r)))
```

The interesting point is that we pass in the identity function as the continuation to `keval`. This is the correct choice here because what we need to return (or pass to the continuation `k`) is a `Fun`-tagged function that takes in an argument and returns the interpretation of the expression `e` in the context of that argument. We simply do not have the continuation for the result of the interpretation at

this point because that continuation only becomes available if and when this function is applied.

Note that it is possible to change the type of the `Fun` tag to allow it to carry functions that are themselves in CPS. In that case, we would construct a function that takes a continuation along with the argument, and we can pass this function to `eval`. This choice, however, is not necessitated by the decision to CPS-convert the interpreter itself, and, we see in the next section, it is possible to get the basic benefits of staging without making this change to our value domain.

The rest of the cases in the interpreter are relatively straightforward.

*Note 3 (Practical Consideration: The time needed to convert to CPS).* In our experience, converting the interpreter into CPS takes almost as much time as writing the interpreter itself in the first place. While this process can be automated, we find that doing the conversion by hand helps us better understand the code and leaves us better prepared for staging this code.

### 4.13 Experiment 2

The results of running our experiment on the CPS-converted interpreter are as follows:

```
# test2 ();;
__ read_file _____ 10x avg = 1.526000E - 01 ms
__ parse _____ 10x avg = 6.843940E + 01 ms
__ kpeval _____ 1x avg = 2.082415E + 04 ms
__ readeval _____ 1x avg = 2.092970E + 04 ms
```

First, while there was no change to the implementations of `read_file` and `parse`, there is some fluctuation in that reading. In our experience, it seems that there is more fluctuation with smaller values. Repeating the experiment generally produced `kpeval` and `readeval` timings within 1-2% of each other.

Contrasting these numbers to the baseline readings, we notice that CPS conversion has slowed down the implementation by about 45%.

## 5 Staging the CPS-Converted Interpreter

In this section, we explain the issues that arise when CPS-converting the interpreter introduced above. We focus on the cases where we need to do more than simply add staging annotations.

### 5.1 Types for the Staged Interpreter

The reader will recall from Part I that MetaOCaml provides a `code` type constructor that can distinguish between regular values and delayed (or staged) values. For conciseness, we elide the so-called environment classifier parameter from types. For example, we simply write `int code` rather than the full `('a, int) code` used in the MetaOCaml implementation.

Adding staging annotations transforms our interpreter from

```
keval : exp -> (string -> var) -> (dom -> dom) -> dom
into
```

```
seval : exp -> (string -> svar) -> (dom code -> dom code)
      -> dom code
```

where `svar` is a modified version of the `var` type defined as follows:

```
type svar = Val of dom code | Ref of (dom ref) code
```

## 5.2 A Quick Staging Refresher

The reader will also recall that MetaOCaml has three staging constructs. *Brackets* delays a computation. So, where as `1+1` has type `int` and evaluates to `2`, the bracketed term `.< 1+1 >.` has type `int code` and evaluates to `.< 1+1 >..` *Escape* allows us to perform a computation on a sub-component of a bracketed term. Thus,

```
let lefty left right = left in
.< 1+ .~(lefty .<2+3>. .<4+5>.) >.
```

contains the escaped expression `.~(lefty .<2+3>. .<4+5>.)`. The whole example evaluates to `.< 1+(2+3)>..` *Run* is the last construct, and it allows us to run a code value. Thus, `.! .<1+2>.` evaluates to `3`.

The rest of the paper does make heavy use of these constructs, so a good understanding of how these constructs are used for staging is needed. If the reader has not already read Part I at this point, we recommend doing so before continuing.

## 5.3 Staging the Interpreter

It would be clearly useful if staging the code of the interpreter was only a matter of adding a few staging annotations at a small number of places in the interpreter. A few features of the interpreter remain unchanged. For example, the overall structure of the interpreter, and all the code down to the main `match` statement do not change. But when we consider the different cases in the `match` statement of the Aloe interpreter, it transpires that only one case can be effectively staged by simply adding staging annotations. That is the case for `if` statements. We can expect this to be generally the case for the interpretation of all syntactic sugar, because those cases are by definition interpreted directly into calls to the interpreter with modified (source abstract syntax tree) inputs.

## 5.4 Cases That Require only Staging Annotations

For most of the cases in the interpreter, staging is achieved by simply adding staging annotations. Examples include booleans, list construction, integers, strings,

variables, and unary, binary, and variable-arity operators. We include here some simple examples for illustration:

```
| A "true" -> k .<Bool true>.
| I i -> k .<Int i>.
| A x ->
  (match env x with
   | Val v -> k v
   | Ref r -> k .<! .~r >.)
| L [A "not"; e1] ->
  seval e1 env (fun r ->
    k .<Bool (not (unBool .~r))>.)
```

In all cases, the application of the continuation occurs outside brackets and is therefore always performed statically.

## 5.5 Lambda Abstraction

With one exception, the case for lambda abstraction is staged simply by adding brackets and escapes. In particular, because we know the number of arguments that the lambda abstraction takes statically, we would like to generate the code for extracting the individual parameters from the parameter list statically. This can be achieved by essentially eta-expanding the list of arguments by taking advantage of the fact that we know the number of arguments. A function that performs this operation would have the type:

```
eta_list : int -> 'a list code -> 'a code list
```

The CPS version of such a function is expressed as follows:

```
let keta_list l v k =
  let rec el_acc l v a k =
    if l<=0 then k []
    else .<match .~v with
      | x::xs -> .~(el_acc (l-1) .<xs>. (a+1) (fun r ->
        k (.<x>. :: r)))
      | _ -> raise (Error "Expecting more arguments")>.
    in el_acc l v 0 k
```

The staged version of the lambda abstraction is simply:

```
| L [A "lambda" ; L axs ; e] ->
  let l = List.length axs
  in k .<Fun (l, fun v ->
    .~(keta_list l .<v>. (fun r ->
      seval e (lnext env
        (List.map (function A x -> x) axs)
        (List.map (fun x -> Val x) r))
      (fun r -> r))))>.)
```

Note that it would have been difficult to perform `lex` statically without performing something similar to eta-expansion.

## 5.6 Function Application

Similarly, function application requires only one main change. Because the abstract syntax tree provides us with a static list of expressions that we map to a static list of interpretations of these expressions, we need a function to convert the second type of list into a corresponding code fragment for inclusion in the generated code. In other words, we need a function with the following type:

```
lift_list : 'a code list -> 'a list code
```

This function can be expressed as follows:

```
let rec lift_list l =
  match l with | [] -> .<[]>.
               | x::xs -> .< ~x :: ~(lift_list xs)>.
```

Using this function, we stage the application case as follows:

```
| L (e::es) ->
  seval e env (fun r1 ->
    .<let (i,f) = unFun ~r1
      in .~(kmap (fun e -> seval e env) es (fun r2 ->
        let args = r2 in
        let l = List.length args
        in .<if l= i
        then let r = f .~(lift_list args)
           in .~(k .<r>.)
        else raise
          (Error ("Function has ^(string_of_int l)^
                " arguments but called with ^
                (string_of_int i)))>.)>.)>.)>.)
```

*Note 4 (Practical Consideration: What to Expect When Staging).* To help give the reader a clear picture of the process of staging, we describe the author's experience in developing the interpreter presented in this section. The first pass of putting staging annotations without running the type-checker was relatively quick and revealed only a small number of questions. The main question was about what to do with the semantics of lambda, and then the need for introducing the two-level eta-expansion [2]. Once this pass was completed, we ran the compiler on this staged code. There was a syntax error every 20-50 lines. Once those were fixed, the compiler began reporting typing errors. There were approximately twice as many typing errors as there were syntax errors. Many of these typing errors also revealed interesting issues that required more care while staging than originally anticipated during the first pass.

## 5.7 Experiment 3

The results of running the experiment on the staged interpreter are as follows:

```
# test3 ();;
__ read_file _____ 10x avg = 1.707000E - 01 ms
__ parse _____ 10x avg = 6.788850E + 01 ms
__ speval _____ 1x avg = 1.018300E + 01 ms
__ compile _____ 1x avg = 2.407380E + 02 ms
__ run _____ 1x avg = 6.653610E + 02 ms
__ readeval _____ 1x avg = 9.668440E + 02 ms
```

Looking at the overall time from file to result, staging provided us with a speedup of about 14 times over the original unstaged version. While the speedup is greater when compared to the CPS'ed version, CPS conversion was carried out only as a step towards staging.

## 6 The Interpretation of a Program as Partially Static Data Structures

In almost any imaginable language that we may consider, there are many programs that contain computation that can be performed *before* the inputs to the program are available. In other words, even when we ignore the possibility of having one of the inputs early, programs themselves are a source of partially static data. If we look closely at the way we have staged programs in the previous section, we notice that we made no attempt to search for or take advantage of such information. A standard source of partially static information is closed expressions, meaning expressions that contain no variables, and therefore, contain no unknown information. Some care must be taken with this notion, because some closed programs can diverge. Another, possibly more interesting and more profitable type of partially static information that can be found in programs in *untyped* languages is *partial information about types*. This information can be captured by the data type tags that allow the runtime of an untyped language to uniformly manipulate values of different types. Because the introduction and the elimination of such tags at runtime can be expensive, reducing such unnecessary work can be an effective optimization technique.

### 6.1 A Partially Static Type for Denotable Values

For Aloe, we can further refine our types for the staged interpreter to facilitate taking advantage of partially static type information in a given untyped program. In particular, instead of having our staged interpreter produce only values of type `dom code`, we allow it to produce values of a staged `dom` type, which we call `sdom` defined as follows:



```

type sdom =
  | SBool of bool code | SInt of int code | SStr of string code
  | SFun of int * (sdom list -> sdom) | SUndefined | SVoid
  | SEmpty | SCons of dom ref code * dom ref code
  | SAny of dom code.

```

In essence, this type allows us to push tags out of the code constructor when we know their value statically. The last constructor allows us to also express the case when there is no additional static information about the tags (which was what we assumed all the time in the previous interpreter).

An important special case above is the case of `Cons`. Because each of the components of a cons cell is mutable, we cannot use the same techniques that we consider here to push information about tags out of the `ref` constructor.

A side effect of this type is that case analysis can become somewhat redundant, especially in cases in which we expect only a particular kind of data. To minimize and localize changes to our interpreter when we make this change, we introduce a matching function for each tag along the following lines:

```

let matchBool r k =
  let m = "Expecting boolean value" in
  match r with
  | SBool b -> k b
  | SAny c -> .<match .~c with Bool b -> .~(k .<b>.)
                                     | _ -> raise (Error m)>.
  | _ -> k .<raise (Error m)>.

```

It is crucial in the last case that we do not raise an error immediately. We return to this point in the context of lazy language constructs.

We also change the type for values stored in the environment as follows:

```

type svar = Val of sdom | Ref of dom ref code

```

Again, we cannot really expect to pull tag information over the `ref` constructor, as side-effects change the value stored in a reference cell.

## 6.2 Refining the Staged Interpreter

To take advantage of partially static information present in a typical program and that can be captured by the data type presented above, we make another pass over the staged interpreter that we have just developed.

**The Easy Cases.** The basic idea of where we get useful information is easy to see from the simple cases in our interpreter:

```

  | A "true"   -> k (SBool .<true>.)
  | A "empty" -> k SEmpty
  | I i -> k (SInt .<i>.)
  | S s -> k (SStr .<s>.)

```

In all of these cases, it is easy to see that changing the return type from `dom code` to `sdom` allows us to push the tags out of the brackets. Naturally, the first case has more static information than we preserve in the value we return, but we focus on issues relating to tag information.

**Marking the Absence of Information.** The first case in which we encounter an absence of static tag information is environment lookup:

```
| A x -> match env x with | Val v -> k v
                          | Ref r -> k (SAny .<(! .~r)>.)
```

Static tag information is absent in `r` because it has `dom code` type, rather than `sdom`. We accommodate this situation by using the `SAny` tag. The intuition here is that, because the de-referencing has to occur at runtime, there is no easy way to statically know the tag on that value. Similar absence of information about the resulting tag also occurs in the interpretation of `car` and `cdr` because they also involve de-referencing.

**Reintroducing Information.** Ground values are not the only source of static information about tags. Generally speaking, the tags on the result of most primitive computations are known statically. For example, we can refine the case for logical negation as follows:

```
| L [A "not"; e1] ->
  xeval e1 env (fun r ->
    matchBool r (fun x ->
      k (SBool .<not .~x>)))
```

Knowing that the tag always has to be `SBool` in the rest of the computation allows us to make sure that this tag does not occur in the generated code when this code fragment occurs in a context that expects a boolean.

Similarly, tag information is reintroduced by all other primitive operations in `Aloe`.

**Strictness and Laziness.** Care is needed when refining the cases of lazy language constructs. In particular, unlike a static type system, dynamic languages allow lazy operations to succeed even when an unneeded argument has a type that would lead to failure if it was needed. The multi-argument logical operators of `Aloe`, `and` and `or` are examples of such lazy language constructs. It is also interesting to note that this issue does not arise with conditionals, because even though they are lazy they are indifferent about the type tags on the arguments on which they are lazy.

The refined interpretation for `and` is as follows:

```
| L ((A "and") :: es) ->
  let rec all l k =
    (match l with
     | [] -> k .<>true>.
```

```

| x::xs ->
  xeval x env (fun r ->
    matchBool r (fun x ->
      .<if .~x
        then .~(all xs k)
        else .~(k .<>false>.>.>)))
in all es (fun r -> k (SBool r))

```

This code is essentially what we would expect from the last two examples. What is interesting here is that using `matchBool` instead of the dynamic `if` `unBool` operation would be incorrect if we were not careful about the last case in the definition of `matchBool`. In particular, if that case immediately raised an exception, then an expression such as `(and true 7)` would fail. In that case, we would be evaluating the types and checking them too strictly. By making sure that we return a code fragment that would raise an error only if we evaluate it, we ensure that the correct semantics is preserved.

**Loss of Information at Mutable Locations.** Intuitively, the boundaries at which we have to lose static information about tags are places where the connection between the source and the target of this information computation must be postponed to the second stage. Cases that involve assignment and the construction of new reference cells are examples of this situation. The places where there is a clear need for losing the static tag information in the following two cases are marked by the use of the `lift` function:

```

| L [A "set!"; A x; e2] ->
  (match env x with
  | Val v ->
    raise (Error "Only mutable variables can be set!")
  | Ref v ->
    xeval e2 env (fun r ->
      .<let _ = (.~v:= .~(lift r)) in .~(k SVoid)>.>))
| L [A "cons"; e1; e2] ->
  xeval e1 env (fun r1 ->
    xeval e2 env (fun r2 ->
      k (SCons (.<ref .~(lift r1)>., .<ref .~(lift r2)>.>))))

```

The `lift` function has type `sdom -> dom code` and is defined as follows:

```

let rec lift x =
match x with
| SBool b    -> .<Bool .~b>.
| SInt i     -> .<Int .~i>.
| SStr s     -> .<Str .~s>.
| SFun (n,f) -> .<Fun (n,fun v ->
  .~(keta_list n .<v>. (fun args ->
    (lift (f (List.map (fun x -> SAny x)
      args))))))>.>.

```

```

| SUndefined -> .<Undefined>.
| SVoid      -> .<Void>.
| SEmpty     -> .<Empty>.
| SCons (h,t) -> .<Cons (~h, ~t)>.
| SAny c     -> c

```

Most cases are self evident. The case functions is probably the most interesting, and there we unfold the statically known function into a dynamic function that explicitly unpacks its argument and also lifts the result of this computation. Lifting the result of the function call is easier than this code makes it seem, and the function does not necessarily need to be recursive. In particular, our interpreter only constructs functions that return values tagged with `SAny`.

**Loss of Information at Function Boundaries.** Pushing static tag information across function boundaries can be difficult. This can be seen by analyzing what happens both in lambda abstractions and in function applications. In a lambda abstraction, we create a function with a body that computes by an interpretation. What continuation should this interpretation use? Previously, we used the identity function, but now the type of the continuation is different. It expects a `sdom` value, and yet it must still return a `code` value. The natural choice seems to be to use `lift` as the continuation and to mark this loss of information in the result by using `SAny`:

```

| L [A "lambda" ; L axs ; e] ->
  let l = List.length axs
  in k (SFun (l, fun r ->
            SAny (xeval e (lctx env
                          (List.map (function A x -> x) axs)
                          (List.map (fun x -> Val x) r))
                    lift)))

```

In the case of application, because we generally do not know what function ultimately results from the expression in the function position, we cannot propagate information across this boundary. If we introduce additional machinery, we can find useful situations in which information can be propagated across this boundary. The simplest solution, however, is as follows:

```

| L (e::es) ->
  xeval e env (fun r1 ->
    .<let (i,f) = unFun ~.(lift r1)
      in ~.(kmap (fun e -> xeval e env) es (fun r2 ->
        let args = (List.map lift r2) in
        let l = List.length args
        in .<if l= i
          then let r = f ~.(lift_list args)
              in ~.(k (SAny .<r>))
          else raise
            (Error ("Function has "^(string_of_int l)^

```

```

      " arguments but called with "^
      (string_of_int i))>.)>.)
| _ -> raise (Error "Expression form not recognized")

```

Using `lift` on the result in the function position and on the arguments means that we are blocking this information from being propagated further. The one use of `SAny` reflects the fact that, without performing the application, we do not know anything statically about what applying the function returns.

### 6.3 Experiment 4

For a rough quantitative assessment of the impact of this refinement of our staged interpreter, we collect the results of running our experiment with this interpreter:

```

# test4 ();;
__ read_file _____ 10x avg = 1.679000E - 01 ms
__ parse _____ 10x avg = 6.519040E + 01 ms
__ xpeval _____ 1x avg = 1.045800E + 01 ms
__ compile _____ 1x avg = 2.199970E + 02 ms
__ run _____ 1x avg = 4.569750E + 02 ms
__ readeval _____ 1x avg = 7.614950E + 02 ms

```

It appears that for overall runtime we tend to get a speedup of around 30% when we take advantage of some of the partially static tag information in our test program. When we look only at the run time for the generated computations, the speedup from this optimization could be as high as 45%.

It is interesting to note that our generation time did not go up when we introduced this optimization, and in fact, compilation time (for the generated code) went down. The shorter compilation time is possibly due to the smaller programs that are generated with this optimization (they contain fewer tagging and untagging operations).

Comparing the overall runtime to the original interpreter, we have a speedup of about 18 times. Comparing just the run time for the generated computation to the original interpreter, we have a speedup of about 30 times.

## 7 Conclusions

In this paper we have explained how to apply the basic techniques introduced in Part I to a large language with higher-order functions and multiple types. We also introduced an optimization technique that can be incorporated into staged interpreters and that is of particular utility to dynamic languages. We find that reasonable speedups are attainable through the use of staging.

To best address the most novel and most interesting ideas, we have not attempted to produce the most effective staged interpreter. For example, we do not consider inlining, which was outlined in Part I. And while we do consider partially static information relating to the typing tags that values carry, we ignore

partially static information relating to the values themselves. In fact, we also ignored the propagation of this information across function boundaries, which is a topic we expect to be able to address in future work. Similarly, we have used the simplest possible implementation of multiple-argument functions, and one can imagine that alternative strategies (possibly using arrays) might yield better results. These are only a few examples of additional optimizations that are available to the multi-stage programmer and that can be applied to improve the performance of a staged interpreter.

**Acknowledgments.** I would very much like to thank the organizers and the participants of the Generative and Transformational Techniques in Software Engineering (GTTSE 2007) and University of Oregon Programming Languages Summer School (2007) for organizing these events and for all the excellent input that they provided. I would like in particular to thank Ralf Lämmel for his constant support, and Dan Grossman, and Ron Garcia for their technical input on the material of the lectures. Dan asked excellent questions at the Oregon Summer School. His fresh perspective lead to significant new material being introduced. I thank Raj Bandyopadhyay, Jun Inoue, Cherif Salama and Angela Zhu for proof reading and commenting on a version of this paper. Ray Hardesty helped us greatly improve our writing.

## References

1. Consel, C., Danvy, O.: For a better support of static data flow. In: Hughes, R.J.M. (ed.) FPCA 1991. LNCS, vol. 523, pp. 496–519. ACM Press, Cambridge (1991)
2. Danvy, O., Malmkjaer, K., Palsberg, J.: Eta-expansion does the trick. Technical Report RS-95-41, University of Aarhus, Aarhus (1995)
3. Friedman, D.P., Want, M., Haynes, C.T.: Essentials of Programming Languages. MIT Press, Cambridge (2003)
4. Hutton, G., Meijer, E.: Monadic Parsing in Haskell. *Journal of Functional Programming* 8(4), 437–444 (1998)
5. Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291-1000, USA, <ftp://cse.ogi.edu/pub/tech-reports/README.html>
6. Sabry, A.: The Formal Relationship between Direct and Continuation-Passing Style Optimizing Compilers: A Synthesis of Two Paradigms. PhD thesis, Rice University (August 1994)
7. Taha, W.: Multi-Stage Programming: Its Theory and Applications. PhD thesis, Oregon Graduate Institute of Science and Technology (1999) Available from [5]
8. Taha, W.: A gentle introduction to multi-stage programming. In: Lengauer, C., Batory, D., Consel, C., Odersky, M. (eds.) Domain-Specific Program Generation. LNCS, vol. 3016, pp. 30–50. Springer, Heidelberg (2004)
9. Thiemann, P.: Correctness of a region-based binding-time analysis. In: Mathematical Foundations of Programming Semantics, Thirteenth Annual Conference, Pittsburgh, Pennsylvania, Carnegie Mellon University, p. 26. Elsevier, Amsterdam (1997)
10. Wadler, P., Taha, W., MacQueen, D.B.: How to add laziness to a strict language without even being odd. In: Proceedings of the 1998 ACM Workshop on ML, Baltimore, pp. 24–30 (1998)

# WebDSL: A Case Study in Domain-Specific Language Engineering

Eelco Visser

Software Engineering Research Group  
Delft University of Technology  
visser@acm.org

**Abstract.** The goal of domain-specific languages (DSLs) is to increase the productivity of software engineers by abstracting from low-level boilerplate code. Introduction of DSLs in the software development process requires a smooth workflow for the production of DSLs themselves. This requires technology for designing and implementing DSLs, but also a methodology for using that technology. That is, a collection of guidelines, design patterns, and reusable DSL components that show developers how to tackle common language design and implementation issues. This paper presents a case study in domain-specific language engineering. It reports on a project in which the author designed and built WebDSL, a DSL for web applications with a rich data model, using several DSLs for DSL engineering: SDF for syntax definition and Stratego/XT for code generation. The paper follows the stages in the development of the DSL. The contributions of the paper are three-fold. (1) A tutorial in the application of the specific SDF and Stratego/XT technology for building DSLs. (2) A description of an incremental DSL development process. (3) A domain-specific language for web-applications with rich data models. The paper concludes with a survey of related approaches.

## 1 Introduction

Abstraction is the key to progress in software engineering. By encapsulating knowledge about low level operations in higher-level abstractions, software developers can think in terms of the higher-level concepts and save the effort of composing the lower-level operations. By stacking layers of abstraction, developers can avoid reinventing the wheel in each and every project. That is, after working for a while with the abstractions at level  $n$ , patterns emerge which give rise to new abstractions at level  $n + 1$ .

Conventional abstraction mechanisms of general purpose programming languages such as methods and classes, are no longer sufficient for creating new abstraction layers [32, 82]. While libraries and frameworks are good at encapsulating functionality, the language which developers need to use to reach that functionality, i.e. the application programmers interface (API), is often awkward. That is, utterances take the form of (complex combinations of) method calls. In some cases, an API provides support for a more appropriate language,

but then utterances take the form of string literals that are passed to library calls (e.g. SQL queries) and which are not checked syntactically, let alone semantically, by the host language. Application programs using such frameworks typically consist of large amounts of boilerplate code, that is, instantiations of a set of typical usage patterns, which is needed to cover the variation points of the framework. Furthermore, there is often a considerable distance between the conceptual functionality of an application and its encoding in the program code, leading to disproportionate efforts required to make small changes. The general-purpose host language of the framework has no knowledge of its application domain, and cannot assist the developer with for instance verification or optimization.

In recent years, a number of approaches, including model-driven architecture [76], generative programming [32, 33], model-driven engineering [61, 82], model-driven software development [87], software factories [30, 51], domain-specific modeling [60], intentional software [84], and language oriented programming [36], have been proposed that aim at introducing new *meta-abstraction mechanisms* to software development. That is, mechanisms that enable the creation of new layers of abstraction.

**Domain-Specific Languages.** Common to all these approaches is the encapsulation of design and implementation knowledge from a particular application or technical domain. The commonalities of the domain are implemented directly in a conventional programming language or indirectly in code generation templates, while the variability is configurable by the application developer through some configuration interface. This interface can take the form of a wizard for simple domains, or full fledged languages for domains with more complex variability [32]. Depending on the approach, such languages are called *modeling languages*, *domain-specific languages*, or even *domain-specific modeling languages*.

In this paper the term *domain-specific language* is used with the following definition:

A domain-specific language (DSL) is a high-level software implementation language that supports concepts and abstractions that are related to a particular (application) domain.

Lets examine the elements of this definition:

A DSL is a *language*, that is, a collection of sentences in a textual or visual notation with a formally defined syntax and semantics. The structure of the sentences of the language should be defined by means of a grammar or meta-model, and the semantics should be defined by means of an abstract mathematical semantics, or by means of a translation to another language with a well understood semantics. Thus, the properties and behavior of a DSL program or model should be predictable.

A DSL is *high-level* in the sense that it abstracts from low-level implementation details, and possibly from particularities of the implementation platform. High-level is a matter of perspective, though. Algol was introduced as a language for the *specification of algorithms* [8] and was high-level with respect to assembly



language. Now we consider the Algol-like languages such as C and Java as low-level implementation languages.

A DSL should support *software implementation*. This does not require that a DSL be a procedural language, like many familiar programming languages. Indeed, declarative DSLs are preferable. However, DSLs should contribute in the creation of components of executable software systems. There are many examples of declarative languages that specify computations. For example, a context-free grammar does not consist of instructions to be executed ('directly') by a computer. Rather it is a declarative definition of the sentences of a language. Yet a grammar may also be used to generate an executable parser for that language.

Finally, the concepts and abstractions of a DSL are *related to a particular domain*. This entails that a DSL does not attempt to address all types of computational problems, or not even large classes of such problems. This allows the language to be *very expressive* for problems that fall in the domain and completely useless for other problems. For problems that are on the edge of the domain (as perceived by the DSL designer), the language may not be adequate. This gray area typically leads to pressure for the DSL to grow beyond its (original) domain. What makes a suitable domain cannot be determined in general; the closest we can get is maybe the circular definition that a domain is a coherent area of (software) knowledge that can be captured in a DSL.

The success of a DSL is measured in terms of the improvement of the software development process it enables. First, it is important that the DSL is actually effective in its intended domain, that is, applications that are considered to fit the domain should be expressible with the DSL<sup>[4]</sup>. This can be expressed as the *completeness* of the DSL or its *coverage* of the domain. Next, building an application with a DSL should take substantially less *effort* than with other means. An approximation of this metric, is the number of DSL *lines of code* (LOC) that is needed for an application compared to what would be needed with conventional programming techniques. An *expressive* DSL requires few lines of code. There is a natural tension between coverage and expressivity. *Non-functional requirements* are just as important as functional requirements. In addition to providing the required functionality, a system should be efficient, safe, secure, and robust, to the extent required. Finally, first-time development of applications may be cheap, but systems usually have a long life span. The question then is how well the DSL supports *maintenance* and how *flexible* it is in supporting new requirements. Van Deursen and Klint [98] discuss maintainability factors of DSLs.

**History.** Domain-specific languages pre-date the recent modeling approaches mentioned above by decades. The name of the programming language for scientific computing FORTRAN, developed by Backus in the late 1950s, is an abbreviation of 'formula translation' [7]. The language borrowed notation from mathematics so that programmers could write mathematical formulas directly, instead of encoding these in low-level stack and register operations, resulting in a

---

<sup>1</sup> 'Application' can be read either as a complete software system or as a component of a software system; DSLs do typically not address all aspects of a software system.

dramatic improvement of programmer productivity. The 1970s Structured Query Language (SQL) [24] provided special notation for querying databases based on Codd's [27] relational database model. So called *little languages* [12] prospered in the Unix environment. Languages such as LEX (lexical analysis), YACC (parsing), PIC (for drawing pictures), and Make (for software building) were developed in the 1970s and 1980s. Another strand in the history are the so called *fourth generation languages* supported by application generators [91], which were supposed to follow-up the third generation general purpose languages. There are several surveys of domain-specific languages, including [73, 85, 86, 98, 99].

**Textual vs Visual.** One aspect of the recent modeling approaches that could be perceived as novel is the preference for visual (graphical) languages in many approaches. For example, model-driven architecture and its derivatives are largely based on the use of UML diagrams to model aspects of software systems. Using UML profiles, the general purpose UML can be used for domain-specific modeling. MetaCase [60] and the Visual Studio DSL Tools [30] provide support for defining domain-specific diagram notations. There is no fundamental difference in expressivity between visual and textual languages. The essence of a language is that it defines structures to which meaning is assigned. Viewing and creating these structures can be achieved with a variety of tools, where various representations are interchangeable. On the one hand, visual diagrams can be trivially represented using text, for instance by taking an XML rendering of the internal structure. On the other hand, textual models can be trivially represented 'visually' by displaying the tree or graph structure resulting from parsing followed by static semantic analysis. Of course, there are non-trivial visualizations of textual models that may provide an alternative view. Some notations are more appropriate for particular applications than others. However, most (successful) DSLs created to date are textual, so text should not be easily discarded as a medium. Another factor is the impact on tools required for viewing and creating models.

**Systematic Development.** Rather than a preference for visual languages, more significant in recent approaches is the emphasis — with support from industry (e.g. Microsoft) and standardization organizations (e.g. OMG) — on the *systematic development and deployment* of DSLs in the software development process. While the DSLs and 4GLs of the past were mostly designed as one-off projects by a domain stakeholder or tool vendor, DSLs should not just be *used* during the software development process, but the *construction* of DSLs should also become part of that process. Where developers (or communities of developers across organizations) see profitable opportunities for linguistic abstraction, new DSLs should be developed. Rather than language design artistry, this requires a solid engineering discipline, which requires an effective collection of techniques and methods for developing domain-specific languages. In their survey of DSL development methods, Mernik et al. [73] describe patterns for decision, analysis, design, and implementation of DSLs. They conclude that most existing work focuses on supporting the implementation of DSLs, but fails to provide support, be it methodological or technological, for earlier phases in the

DSL life cycle. Thus, a challenge for a software engineering discipline in which DSLs play a central role is a systematic and reproducible DSL development methodology. As for the *use* of DSLs, important criteria for the effectiveness of such a methodology are the effort it takes to *develop* new DSLs and their subsequent *maintainability*.

In previous work I have focused on the creation of language implementation technology, that is, a set of DSLs and associated tools for the development and deployment of language processing tools. The SDF syntax definition formalism [53, 101], the Stratego/XT program transformation language and tool set [17, 19, 103], and the Nix deployment system [37, 39] provide technology for defining languages and the tools needed for their operation. Publications resulting from this research typically present innovations in the technology, illustrated by means of case studies. This paper for a change does not present technological innovations in meta technology, but rather an application of that technology in domain-specific language engineering, with an attempt at exploring the design space of DSL development methodology.

**WebDSL.** This paper presents a case study in domain-specific language engineering. The paper tracks the design and implementation of WebDSL, a DSL for web applications with a rich data model. The DSL is implemented using Stratego/XT and targets high-level Java frameworks for web engineering. The contributions of this paper are

- A tutorial on DSL design, contributing to the larger goal of building a methodology for the design and implementation of domain-specific languages. This includes an incremental (agile) approach to analysis, design, and implementation, and the illustration of best practices in language design, such as the use of a core language and the introduction of syntactic abstractions to introduce higher-level abstractions.
- A tutorial on the application of Stratego/XT to building (textual) domain-specific languages, illustrating the utility of techniques such as term rewriting, concrete object syntax, and dynamic rewrite rules.
- The introduction of WebDSL, a domain-specific language for the implementation of web applications with a rich data model.

The next section describes the development process and introduces the setup of sections 3 to 9, which discuss the stages in the development of WebDSL. Sections 10 to 12 evaluate the resulting WebDSL language and its development process, also with respect to related work.

## 2 Process Definition and Domain Analysis

According to the DSL development patterns of Mernik et al. [73], the DSL life cycle consists of (1) a decision phase in which the decision whether or not to build a DSL is taken, (2) an analysis phase in which the application domain is analyzed, (3) a design phase in which the architecture and language are designed, and

finally, (4) an implementation phase in which the DSL and supporting run-time system are constructed. We can add (5) a deployment phase, in which DSLs and the applications constructed with them are used, and (6) a maintenance phase in which the DSL is updated to reflect new requirements. In this paper, I propose an incremental, iterative, and technology-driven approach to DSL development in which analysis, design, and implementation are combined in the spirit of agile software development [11]. Deployment and maintenance are left for future work. In this section, I describe and motivate this process model and relate it to the patterns of Mernik et al. [73]. The bulk of the paper will then consist of a description of the iterations in the design of WebDSL.

## 2.1 When to Develop a DSL?

The development of a DSL starts with the decision to develop one in the first place. Libraries and frameworks form a good alternative for developing a DSL. Many aspects of application development can be captured very well in libraries. When a domain is so fresh that there is little knowledge about it, it does not make sense to start developing a DSL. First the regular software engineering process should be applied in order to determine the basic concepts of the field, develop a code base supported with libraries, etc. When there is sufficient insight in the domain and the conventional programming techniques fail to provide the right abstractions, there may be a case for developing a DSL. So, what were the deciding factors for developing WebDSL?

The direct (personal) inspiration for developing WebDSL are wiki systems such as MediaWiki used for wikipedia, and more concretely TWiki used for program-transformation.org and other web sites maintained by the author. Wikis enable a community — the entire web population or the members of an organization — to contribute to the content of a site using a browser as editor. However, the data model for that content is poor, requiring all structured information to be encoded in the text of a page. This lack of structure entails that querying data and data validation depend on text operations. The initial goal of WebDSL is to combine the flexible, online editing of content as provided by wikis with a rich data model that allows presentation of and access to the underlying data in a variety of ways.

The scope of WebDSL is *interactive dynamic web applications with a rich application-specific data model*. That is, web applications with a database for data storage and a user interface providing several views on the data in the database, but also the possibility to modify those data via the browser. An additional assumption is that the data model is static, i.e. it is designed during development and cannot be changed online.

The engineering of web applications is a fairly mature field. There is an abundance of libraries and frameworks supporting the construction of web applications. The state-of-the art for the construction of robust industrial strength web applications are the Java and C# web engineering platforms. Based on the portability of Java and the availability of infrastructure for generation of Java in Stratego/XT, I have decided to restrict my attention to this platform for this

case study. While current frameworks provide good support for the basic mechanics of web applications — such as handling requests, parsing form data, and producing XHTML — there is a strong case for the development of a DSL for this domain; several of the decision patterns of Mernik et al. [73] apply to the domain of web applications.

*Task Automation.* Compared to the CGI programming of early web applications, a mature web engineering platform takes care of low-level concerns. For example, Java servlets deal with the mechanics of receiving requests from and sending replies to clients. Java Server Faces (JSF) deal with the construction of web pages *and* with the analysis of form data received from the client. Despite such facilities, web programming often requires a substantial amount of boilerplate code; many Java classes or XML files that are very similar, yet not exactly the same either. Conventional abstraction mechanisms are not sufficient for abstracting over such patterns. Thus, one case for a web DSL is programming-*task automation*, i.e. preventing the developer from having to write and maintain boilerplate code.

*Notation.* The current platform provides an amalgam of often verbose languages addressing different concerns, which are not integrated. For example, the Java-JPA-JSF-Seam platform is a combination of XHTML extended with JSF components and EL expressions (Java-*like* expressions embedded in XML attributes), Java with annotations for declaration of object-relational mapping and *dependency injection*, and SQL queries ‘embedded’ in Java programs in the form of string literals. A concise and consistent notation, that linguistically integrates the various aspects of web application construction would lighten development and maintenance. Note that linguistic integration does not necessarily mean a loss of separation of concerns, but rather that different concerns can be expressed in the same language.

*Verification.* Another consequence of the lack of integration of web application technologies is the lack of static verification of implementations. Components linked via dependency injection are only checked at run-time or deployment-time. Queries embedded in strings are not checked syntactically or for compatibility with the data model until run-time. References in EL expressions in XHTML files are only checked at run-time. These issues clearly illustrate that the abstraction limits of GPLs have been reached; the static typechecking of Java programs does not find these problems. A static verification phase, which would be enabled by an integrated language would avoid the tedious debugging process that these problems cause.

*GUI Construction.* The user interface portion of a web application is typically defined by means of a template mechanism. JSP-style templates consist of plain text with anti-quotations in which fragments of Java code are used to insert ‘dynamic’ content derived from data objects. The framework has no knowledge of the structure of the HTML code generated by the template, so it is very easy to generate non well-formed documents. Java Server Faces templates are more

advanced in that they define the complete document by means of a structured XML document, which is parsed at deployment-time. XHTML is generated by rendering this structure. Insertion of content from data object is achieved by means of ‘EL expressions’ in XML attributes. Still, templates are very verbose and concerned with low-level details. Furthermore, the EL expressions are only parsed and checked at run-time.

*Analysis and Optimization.* There are also opportunities for domain-specific analysis and optimization. For example, optimization of database queries in the style of Wiedermann and Cook [108] might be useful in improving the performance of applications without resorting to manual tuning of generated queries. These concerns are not (yet) addressed in WebDSL.

## 2.2 Domain Analysis

*Domain analysis* is concerned with the analysis of the basic properties and requirements of the problem domain. For example, a first analysis of the domain would inform us that the development of a web application involves a data model, an object-relational mapping, a user interface, data input and output methods, data validation, page flow, and access control. Additionally, it may involve file upload, sending and receiving email, versioning of data, internationalization, and higher-level concerns such as work-flow. A more thorough analysis studies each of the concerns of a domain in more detail, and establishes terminology and requirements, which are then input for the design of a DSL.

**Deductive.** The traditional development process for domain-specific languages follows a top-down or deductive track and starts with an exhaustive domain analysis phase, e.g. [29, 73, 98]. The advantage of this approach is a thorough analysis. The risk of such a deductive (top-down) approach is that the result is a language that is difficult to implement. Furthermore, a process developing an all encompassing DSL for a domain runs the usual risks of top-down design, such as over design, late understanding of requirements, leading to discovery of design and implementation problems late in the process.

**Inductive.** Rather than designing a complete DSL before implementation, this paper follows an inductive approach by incrementally introducing abstractions that allow one to capture a set of common programming patterns in software development for a particular domain. This should enable a quick turn-around time for the development of such abstractions. Since the abstractions are based on concrete programming patterns, there are no problems with implementing them.

*Technology-driven.* Rather than designing a DSL based on an analysis of the domain in the abstract, the approach is *technology-driven*, i.e. considers best practices in the implementation of systems in the domain. This is similar to *architecture-centric* model-driven software development [87] or designing DSLs based on a *program family* [28]. After the initial determination of the scope of the domain, domain analysis then is concerned with exploring the technology that is available, and analyzing how it is typically used.

Selecting a specific technology helps in keeping a DSL design project grounded; there is a specific reference architecture to target in code generation. However, a risk with this approach is that the abstractions developed are too much tied to the particularities of the target technology. In domains such as web applications there are many *virtual machines*. Each combination of implementation languages, libraries, and frameworks defines a virtual machine to target in software development. Each enterprise system/application may require a different virtual machine. This is similar to the situation in embedded systems, where the peculiarities of different hardware architectures have to be dealt with. Thus, a consideration for the quality of the resulting DSL is the amount of leakage from the (concrete) target technology into the abstractions of the DSL; how easy is it to port the DSL to other virtual machines?

*Iterative.* Developing the DSL in iterations can mitigate the risk of failure. Instead of a big project that produces a functional DSL in the end, an iterative process produces useful DSLs for sub-domains early on. This can be achieved by extending the coverage of the domain incrementally. First the domain concerns addressed can be gradually extended. For example, the WebDSL project starts with a data model DSL, addressing user interface issues only later in the project. Next, the coverage within each concern does not have to be complete from the start either. The WebDSL coverage of user interface components is modest at first, concentrating on the basic architecture, rather than covering all possible fancy features. This approach has the advantage that DSLs for relevant areas of the domain are available early and can start to be used in development. The feedback from applying the DSL under development can be very valuable for evaluating the design of abstractions and improving them. Considering the collection of patterns will hopefully lead to a deeper insight in how to make even better abstractions for the application domain.

### 2.3 Outline

The rest of this paper discusses the iterations in the design and implementation of WebDSL. These iterations are centered around three important DSL design patterns: *finding programming patterns*, *designing a core language*, and *building syntactic abstractions* on top of the core language.

**Programming Patterns.** The first step in developing a new DSL is to explore the technology for building systems in the domain to find common programming patterns. That is, program fragments that occur frequently with slight variations. This exploration can take the form of inspecting legacy code, but preferably the technical literature and reference implementations. These typically present ideal programming patterns, as opposed to legacy code exposed to design erosion. The idea then is to capture the variability in the patterns by an appropriately designed abstraction. The commonality in the patterns is captured in code templates used in the generator that translates the abstractions to target code.



In Sections 3 to 5 we explore the domain of web applications built with Java/JSF/JPA/Seam and the techniques for implementing a DSL for this domain. Section 3 starts with looking at programming patterns for the implementation of *data models* using the Java Persistence API (JPA). A simple DSL for declaration of JPA entities is then developed, introducing the techniques for its implementation, including syntax definition and term rewriting in Stratego/XT<sup>2</sup>. Section 4 develops a generator for deriving from a data model declaration, standardized pages for viewing and editing objects. In Section 5 the coverage of the data model DSL is increased in various directions.

**Core Language.** The abstractions that result from finding programming patterns tend to be coarse grained and capture large chunks of code. In order to implement a variation on the functionality captured in the generator templates, complete new templates need to be developed. The templates for generating view and edit pages developed in Section 4 are very specific to these interaction patterns. Extending this approach to include other, more sophisticated, interaction patterns would lead to a lot of code duplication *within the generator*. To increase the coverage of the DSL it is a good idea to find the *essential abstractions* underlying the larger templates and develop a *core language* that supports freely mixing these abstractions. In Section 6 a core language for web user interfaces is developed that covers page flow, data views, and user interface composition. In Section 7 the core language is extended with typechecking, data input, and queries.

**Abstraction Mechanisms.** A good core language ensures an adequate coverage of the domain. However, this may come at a loss of abstraction. Core language constructs are typically relatively low-level, which leads to frequently occurring patterns combining particular constructs. To capture such patterns and provide high-level abstractions to DSL programmers we need abstraction mechanisms.

Some of these patterns can be captured in templates or modules in a library of common components. In Section 8 WebDSL is extended with abstraction mechanisms for web developers. Template definitions allow developers to create reusable page elements. Modules support the division of an application into reusable files.

Other patterns require reflection over types or other properties of program elements, which may not be so easily defined using the abstraction facilities of the language. Advanced reflection and analysis mechanisms carry a run-time cost and considerably increase the complexity of the language. Such patterns are typically more easily defined using *linguistic abstraction*, i.e. the extension of the language with *syntactic abstractions*, which are implemented by means of transformations to the core language — as opposed to transformations to the target language. Building layers of abstractions on top of a core language is a key feature of software development with DSLs; new abstractions are defined relatively

<sup>2</sup> While the *concepts* underlying Stratego/XT are explained (to the extent necessary for the tutorial), the details of operating Stratego/XT are not. To get acquainted with the tools the reader should consult the Stratego/XT tutorial and manual [18].



easily, by reusing the implementation knowledge captured in the generator for the core language. Section 9 illustrates this process by defining a number of syntactic abstractions for data input and output.

### 3 Programming Patterns: Data Model

The first step in the process of designing a DSL is to consider common programming patterns in the application domain. We will turn these patterns into templates, i.e. program fragments with holes. The holes in these templates can be filled with values to realize different instantiations of the programming pattern. Since the configuration data needed to fill the holes is typically an order of magnitude smaller than the programming patterns they denote, a radical decrease in programming effort is obtained. That is, when exactly these patterns are needed, of course. With some thought the configuration data can be turned into a proper domain-specific language. Instead of doing a ‘big design up front’ to consider all aspects a DSL for web applications should cover and the language constructs we would need for that, we develop the DSL in iterations. We start with relatively large patterns, i.e., complete classes.

#### 3.1 Platform Architecture

As argued before, we take a particular technology stack as basis for our WebDSL. That is, this technology stack will be the platform on which code generated from DSL models will run. That way we have a concrete implementation platform when considering design and implementation issues and it provides a concrete code base to consider when searching for programming patterns. Hopefully, we will arrive at a design of abstractions that transcend this particular technology.

In this work we use the Seam architecture for web applications. That is, applications consist of three layers or tiers. The presentation layer is concerned with producing web pages and interpreting events generated by the user. For this layer we use JavaServer Faces (JSF) [72]. The persistence layer is concerned with storing data in the database and retrieval of data from the database. This layer really consists of two parts. The database proper is a separate service implemented by a relational database. In the implementation of a web application, however, we approach the database via an object-relational mapping (ORM) framework, which takes care of the communication with the database and translates relational data into objects that can be used naturally in an object-oriented setting. Thus, after defining a proper mapping between objects and database tables, we need no longer worry about the database side. Finally, to connect the JSF pages defining the user-interface with the objects obtained from the database we use EJB3 session beans [56, 74].

While it used to be customary for these types of frameworks to require a large portion of an application to be implemented in XML configuration files, this trend has been reversed in the Seam architecture. Most of the configuration is now expressed as annotations in Java classes building on the concept of

*dependency injection* [46]. A little XML configuration remains, for instance, to define where the database is to be found. This configuration is mostly static and will not be a concern in this paper.

In this section, we start with considering *entity beans*, i.e. Java classes that implement persistent objects. We will build a generator for such classes, starting with a syntax definition for a data model language up to the rewriting rules defining Java code generation. As such, this section serves as an introduction to these techniques. In the next section we then consider the generation of basic web pages for viewing and editing the content of persisted objects.

### 3.2 Programming Patterns for Persistence

The Java Persistence API (JPA) [90] is a standard proposed by Sun for object-relational mapping (ORM) for Java. The API is independent of vendor-specific ORM frameworks such as Hibernate; these frameworks are expected to implement JPA, which, Hibernate 3 indeed does [10]. While earlier versions of Hibernate used XML configuration files to define the mapping between database schemas and Java classes, the JPA approach is to express these mappings using Java 5 annotations in Java classes. Objects to be persisted in a database are represented using ‘plain old Java objects (POJOs)’. Classes are mapped to database tables and properties (fields with getters and setters) are mapped to database columns. We will now inspect the ingredients of such classes as candidates for code generation.

**Entity Class.** An *entity class* is a Java class annotated with the `@Entity` annotation and with an empty constructor, which guarantees that the persistence framework can always create new objects.

```
@Entity
public class Publication {
    public Publication () { }
    // properties
}
```

An entity class is mapped to a database table with the same name. If desired, an alternative name for the table can be specified, but we will not be concerned with that (for the time being at least). In general, for many of the patterns we consider here there are alternatives that have (subtly) different semantics. For now, we consider ‘vanilla’ patterns. Later, if and when the need arises we can introduce more variability.

**Identity.** Entities should have an *identity* as primary key. This identity can be any value that is a unique property of the object. The annotation `@Id` is used to indicate the property that represents the identity. However, the advice is to use an identity that is not directly linked to the logic of the object, but rather to use a synthetic identity, for which the database can generate unique values [10]. This then takes the following pattern:

```

@Id @GeneratedValue
private Long id;
public Long getId() { return id; }
private void setId(Long id) { this.id = id; }

```

**Properties.** The values of an object are represented by *properties*, class member fields with getters and setters. Such properties are mapped to columns in the database table for the enclosing class.

```

private String title;
public String getTitle() { return title; }
public void setTitle(String title) { this.title = title; }

```

**Entity Associations.** No annotations are needed for properties with simple types. However, properties referring to other entities, or to collections of entities, require annotations. The following property defines an association to another entity:

```

@ManyToOne
private Person author = new Person();
public Person getAuthor() { return author; }
public void setAuthor(Person author) { this.author = author; }

```

The `@ManyToOne` annotation states that many `Publications` may be authored by a single `Person`. Alternatively, we could use a `@OneToOne` annotation to model that only one `Publication` can be authored by a `Person`, which implies ownership of the object at the other end of the association.

### 3.3 A Data Model DSL

Entity classes with JPA annotations are conceptually simple enough. However, there is quite a bit of boilerplate involved. First of all, the setters and getters are completely redundant, and also the annotations can be become fairly complex. However, the essence of an entity class is simple, i.e., a class name, and a list of properties, i.e., (name, type) pairs. This information can be easily defined in a structure of the form `A{ prop* }` with `A` a name (identifier) and `prop*` a list of properties of the form `x : t`, i.e., a pair of a field name `x` and a type `t`. For example, the following entity declarations

```

entity Publication {
    title    : String
    author   : Person
    year     : Int
    abstract : String
    pdf      : String
}
entity Person {
    fullname : String
    email    : String
    homepage : String
}

```

define the entities `Publication` and `Person`, which in Java take up easily 100 lines of code.

The collection of data used in a (web) application is often called the *domain model* of that application. While this is perfectly valid terminology it tends to give rise to confusion when considering domain-specific languages, where the domain is the space of all applications. Therefore, in this paper, we stick to the term *data model* for the data in a web application.

### 3.4 Building a Generator

In the rest of this section we will examine how to build a generator for the simple data modeling language sketched above. A generator typically consists of three main parts, a parser, which reads in the model, the code generator proper, which transforms an abstract syntax representation of the model to a representation of the target program, and a pretty-printer, which formats the target program and writes it to a text file. Thus, we need the following ingredients. A definition of the concrete syntax of the DSL, for which we use the syntax definition formalism SDF2. A parser that reads model files and produces an abstract representation. A definition of that abstract representation. A transformation to the abstract representation of the Java program to be generated, for which we use term rewrite rules. And finally, a definition of a pretty-printer.

### 3.5 Syntax Definition

For syntax definition we use the syntax definition formalism SDF2 [101]. SDF2 integrates the definition of the lexical and context-free syntax. Furthermore, it is a modular formalism, which makes it easy to divide a language definition into reusable modules, but more importantly, it makes it possible to *combine* definitions for different languages. This is the basis for rewriting with concrete syntax and language embedding; we will see examples of this later on.

The syntax of the basic domain modeling language sketched above is defined by the following module `DataModel`. The module defines the lexical syntax of identifiers (`Id`), integer constants (`Int`), string constants (`String`), whitespace and comments (`LAYOUT`). Next the context-free syntax of models, entities, properties, and sorts is defined. Note that SDF productions have the non-terminal being defined on the right of the `->` and the body on the left-hand side.

```

module DataModel
exports
  sorts Id Int String Definition Entity Property Sort
lexical syntax
  [a-zA-Z][a-zA-Z0-9\_] * -> Id
  [0-9]+ -> Int
  "\"" ~["\n"] * "\"" -> String
  [ \t\n\r] -> LAYOUT
  "/" ~["\n\r"] * ["\n\r"] -> LAYOUT
context-free syntax

```

---

<sup>3</sup> Integer and string constants are not used in this version of the language.

```

Definition*      -> Model      {cons("Model")}
Entity           -> Definition
"entity" Id "{" Property* "}" -> Entity    {cons("Entity")}
Id ":" Sort      -> Property   {cons("Property")}
Id              -> Sort       {cons("SimpleSort")}

```

**Abstract Syntax.** An SDF syntax definition defines the concrete syntax of strings in a language. For transformations we want an abstract representation, i.e. the tree structure underlying the grammar. This structure can be expressed concisely by means of an algebraic signature, which defines the constructors of abstract syntax trees. Such a signature can be derived automatically from a syntax definition (using `sdf2rtg` and `rtg2sig`). Each context-free production gives rise to a constructor definition using the name declared in the `cons` attribute of the production as constructor name, and the non-literal sorts as input arguments. Thus, for the `DataModel` language defined above, the abstract syntax definition is the following:

```

signature
constructors
  Model      : List(Definition) -> Model
             : Entity -> Definition
  Entity     : Id * List(Property) -> Entity
  Property   : Id * Sort -> Property
  SimpleSort : Id -> Sort
             : String -> Id

```

Signatures describe well-formed terms. Terms are isomorphic with structures of the following form:

$$t := c(t_1, \dots, t_n)$$

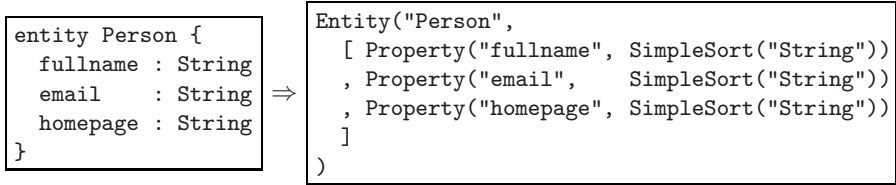
That is, a term is an application of a constructor  $c$  to zero or more terms  $t_i$ . In practice, the syntax is a bit richer, i.e., terms are defined as

$$t := s \mid i \mid f \mid c(t_1, \dots, t_n) \mid [t_1, \dots, t_n] \mid (t_1, \dots, t_n)$$

including special notation for string ( $s$ ), integer ( $i$ ), and float ( $f$ ) constants, and for lists ( $[]$ ), and tuples ( $()$ ). A *well-formed term* according to a signature is defined according to the following rules. (1) If  $t_1, \dots, t_n$  are well-formed terms of sorts  $s_1, \dots, s_n$ , respectively, and  $c : s_1 * \dots * s_n \rightarrow s_0$  is a constructor declaration in the signature, then  $c(t_1, \dots, t_n)$  is a well-formed term of sort  $s_0$ . (2) If  $t_1, \dots, t_n$  are well-formed terms of sort  $s$ , then  $[t_1, \dots, t_n]$  is a well-formed term of sort `List(s)`. (3) If  $t_1, \dots, t_n$  are well-formed terms of sorts  $s_1, \dots, s_n$ , respectively, then  $(t_1, \dots, t_n)$  is a well-formed term of sort  $(s_1, \dots, s_n)$ .

**Parsing.** A parser reads a textual representation of a model, checks it against the syntax definition of the language, and builds an abstract syntax representation of the underlying structure of the model text. Parse tables for driving the `sgrl` parser can be generated automatically from a syntax definition (using `sdf2table`). The `sgrl` parser produces an abstract syntax representation in the

Annotated Term (ATerm) Format [96], as illustrated by the following parse of a data model:



### 3.6 Code Generation by Rewriting

Programs in the *target* language can also be represented as terms. For example, Figure 1 shows the abstract representation of the basic form of an entity class (as produced by the `parse-java` tool, which is based on an SDF definition of the syntax of Java 5). This entails that code generation can be expressed as a term-to-term transformation. Pretty-printing of the resulting term then produces the program text. The advantage of generating terms over the direct generation of text is that (a) the structure can be checked for syntactic and type consistency, (b) a pretty-printer can ensure a consistent layout of the generated program text, and (c) further transformations can be applied to the generated code. For example, in the next section we will see that an interface can be derived from the generated code of a class.

**Term rewriting.** Term rewriting is a formalism for describing term transformations [6]. A *rewrite rule*  $p_1 \rightarrow p_2$  defines that a term matching the term pattern  $p_1$  can be replaced with an instantiation of the term pattern  $p_2$ . A term

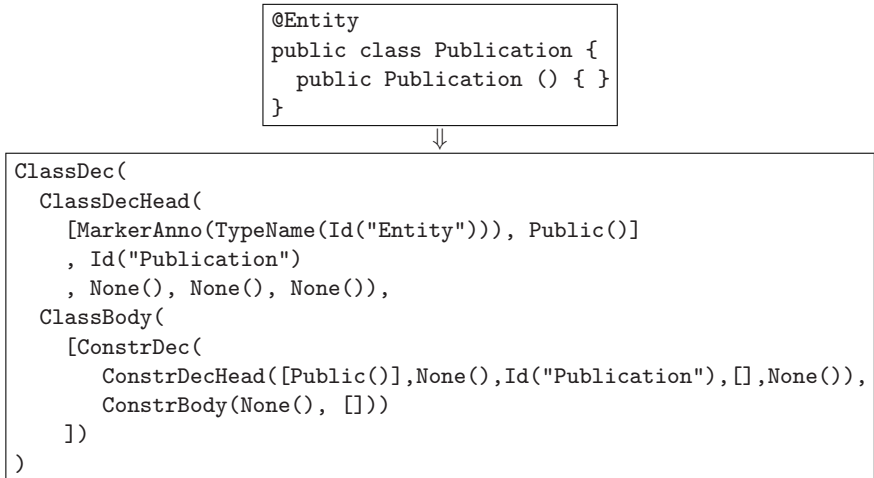


Fig. 1. Abstract syntax for a Java class

```

entity-to-class :
Entity(x, prop*) ->
ClassDec(
  ClassDecHead(
    [MarkerAnno(TypeName(Id("Entity"))), Public()]
    , Id(x)
    , None(), None(), None()),
  ClassBody(
    [ConstrDec(
      ConstrDecHead([Public()],None(),Id(x),[],None()),
      ConstrBody(None(), []))
    ])
)

```

Fig. 2. Term rewrite rule

pattern is a term with *variables*. In standard term rewriting, rewrite rules are applied exhaustively until a normal form is obtained. Term rewriting engines employ a built-in *rewriting strategy* to determine the order in which subterms are rewritten. *Stratego* [20, 105] is a transformation language based on term rewriting. Rewrite rules are *named* and can be *conditional*, i.e., of the form  $l : p_1 \rightarrow p_2$  where  $s$ , with  $l$  the name and  $s$  the condition. Stratego extends basic term rewriting by providing *programmable rewriting strategies* that allow the developer to determine the order in which rules are applied. The rewrite rule in Figure 2 defines the transformation of an **Entity** term in the data model language to the basic Java class pattern that we saw above. Note that the rule generalizes over the particular class by using instead of the name "Publication", a *variable*  $x$  for the class and the constructor. Thus, the rule generates for an arbitrary **Entity**  $x$ , a class  $x$ .

In Stratego, a rewrite rule is a special case of a *rewriting strategy* [105]. A strategy is an algorithm that transforms a term into another term, or fails. A strategy definition can invoke rewrite rules and other strategies by name. Strategies can be parametrized with strategies and terms, supporting the definition of reusable strategies.

**Concrete Syntax.** The `entity-to-class` rewrite rule defines a template for code generation. However, the term notation, despite its advantages for code generation as noted above, is not quite as easy to read as the corresponding program text. Therefore, Stratego supports the definition of rewrite rules using the *concrete syntax* of the subject language [102]. For example, the following rule is the concrete syntax equivalent of the rule in Figure 2.

```

entity-to-class :
|[ entity x_Class { prop* } ]| ->
|[ @Entity
  public class x_Class {
    public x_Class () { }
  } ]|

```

Note that the identifier `x_Class` is recognized by the Stratego parser as a *meta-variable*, i.e. a pattern variable in the rule.

While rewrite rules using concrete syntax have the readability of textual templates, they have all the properties of term rewrite rules. The code fragment is parsed using the proper syntax definition for the language concerned and thus syntax errors in the fragment are noticed at compile-time of the generator. The transformation produces a term and not text; in fact, the rule is equivalent to the rule using terms in Figure 2. And thus the advantages of term rewriting discussed above hold also for rewriting with concrete syntax.

### 3.7 Pretty-Printing

Pretty-printing is the inverse of parsing, i.e. the conversion of an abstract syntax tree (in term representation) to a, hopefully readable, program text. While this can be done with any programmatic method that prints strings, it is useful to abstract from the details of formatting program texts by employing a specialized library. The GPP library [35] supports formatting through the *Box language*, which provides constructs for positioning text blocks. For pretty-printing Java and XML, the Stratego/XT tool set provides custom built mappings to Box. For producing a pretty-printer for a new DSL that is still under development it is most convenient to use a pretty-printer *generator* (`ppgen`), which produces a pretty-print *table* with mappings from abstract syntax tree constructors to Box expressions. The following is a pretty-print table for our `DataModel` language:

```
[
  Entity          -- V[V is=2[ KW["entity"] H[_1 KW["{}"] _2] KW["}"]],
  Entity.2:iter-star -- _1,
  Property        -- H[_1 KW[":" ] _2],
  SimpleSort      -- _1
]
```

Here *V* stands for *vertical* composition, *H* stands for *horizontal* composition, and *KW* stands for *keyword*. While a pretty-printer generator can produce a *correct* pretty-printer (such that `parse(pp(parse(prog))) = parse(prog)`), it is not possible to automatically generate pretty-printers that generate a *pretty* result (although heuristics may help). So it is usually necessary to tune the pretty print rules.

### 3.8 Generating Entity Classes

Now that we have seen the techniques to build the components of a generator we can concentrate on the rules for implementing the `DataModel` language. Basically, the idea is to take the program patterns that we found during the analysis of the solution domain, and turn them into transformation rules, by factoring out the application-specific identifiers. Thus, an entity declaration is mapped to an entity class as follows:



```

entity-to-class :
| [ entity x_Class { prop* } ] | ->
| [ @Entity public class x_Class {
  public x_Class () { }
  @Id @GeneratedValue private Long id;
  public Long getId() { return id; }
  private void setId(Long id) { this.id = id; }
  ~*cbds
} ] |
where cbds := <mapconcat(property-to-gettersetter(|x_Class))> prop*

```

Since an entity class always has an identity (at least for now), we include it directly in the generated class. Furthermore, we include, through the anti-quotation `~*`, a list of class body declarations `cbds`, which are obtained by mapping the properties of the entity declaration with `property-to-gettersetter`. Here `mapconcat` is a strategy that applies its argument strategy to each element of a list, concatenating the lists resulting from each application.

**Value Types.** The mapping for value type properties simply produces a private field with a public getter and setter.

```

property-to-gettersetter(|x_Class) :
| [ x_prop : s ] | ->
| [ private t x_prop;
  public t get#x_prop() { return title; }
  public void set#x_prop(t x) { this.x = x; } ] |
where t := <builtin-java-type> s

```

This requires a bit of *name mangling*, i.e. from the name of the property, the names of the getter and setter are derived. This is achieved using an extension of Java for name composition. The `#` operator combines two identifiers into one, observing Java naming conventions, i.e. capitalizing the first letter of all but the first identifier. Note that the name of the enclosing class (`x_Class`) is passed to the rule as a term parameter. Stratego distinguishes between strategy and term parameters of a rule or strategy by means of the `|`; the (possibly empty) list of parameters before the `|` are strategies, the ones after the `|` are terms.

The fact that the property is for a value type is determined using the strategy `builtin-java-type`, which defines a mapping for the built-in types of the `DataModel` language to types in Java that implement them. For example, the `String` type is defined as follows:

```

builtin-java-type :
SimpleSort("String") -> type|[ java.lang.String ]|

```

**Reference Types.** Properties with a reference to another type are translated to a private field with getters and setters with the `@ManyToOne` annotation. For the time being, we interpret such an association as a non-exclusive reference.

```

property-to-getsetter(|x_Class) :
  |[ x_prop : s ]| ->
  |[ @ManyToOne
    private t x_prop;
    public t get#x_prop() { return x_prop; }
    public void set#x_prop(t x_prop) { this.x_prop = x_prop; } ]|
  where t := <defined-java-type> s

```

**Propagating Declared Entities.** The previous rule decides that the property is an association to a reference type using the strategy `defined-java-type`, which maps entities declared in the data model to the Java types that implement them. Since the collection of these entity types depends on the data model, the `defined-java-type` mapping is defined *at run-time* during the transformation as a *dynamic rewrite rule* [20]. That is, before generating code for the entity declarations, the following `declare-entity` strategy is applied to each declaration:

```

declare-entity =
  ?Entity(x_Class, prop*)
  ; rules(
    defined-java-type :
      SimpleSort(x_Class) -> type|[ x_Class ]|
  )

```

This strategy first matches ( $?p$  with  $p$  a term pattern) an entity declaration and then defines a rule `defined-java-type`, which inherits from the match the binding to the variable `x_Class`. Thus, for each declared entity a corresponding mapping is defined. As a result, the `property-to-getsetter` rule fails when it is applied to a property with an association to a non-existing type (and an error message might be generated to notify the user). In general, dynamic rewrite rules are used to add new rewrite rules at run-time to the transformation system. A dynamic rule inherits variable bindings from its definition context, which is typically used to propagate context-sensitive information.

### 3.9 Composing a Code Generator

Using the ingredients discussed above, the basic version of the code generator for WebDSL is defined as the following Stratego strategy:

```

webdsl-generator =
  xtc-io-wrap(webdsl-options,
    parse-webdsl
    ; alltd(declare-entity)
    ; collect(entity-to-class)
    ; output-generated-files
  )

```

The strategy invokes `xtc-io-wrap`, a library strategy for handling command-line options to control input, output, and other aspects of a transformation tool. The

Fullname	Eelco Visser
Email	visser@acm.org
Homepage	http://www.eelcovisser.net
Photo	/img/eelcovisser.jpg
Address	
Street	Mekelweg 4
City	Delft
Phone	+31 (015) 27 87088
user	EelcoVisser
Edit	

Fig. 3. person page

Fullname	Eelco Visser
Email	visser@acm.org
Homepage	http://www.eelcovisser.net
Photo	/img/eelcovisser.jpg
Address	
Street	Mekelweg 4
City	Delft
Phone	+31 (015) 27 87088
user	EelcoVisser
Save	

Fig. 4. editPerson page

argument of `xtc-io-wrap` is a sequence of strategy applications ( $s_1; s_2$  is the sequential composition of two strategies). `parse-webdsl` parses the input model using a parse table generated from the syntax definition, producing its abstract syntax representation. The `alltd` strategy is a generic traversal, which is used here to find all entity declarations and generate the `defined-java-type` mapping for each. The generic `collect` strategy is then used to create a set of Java entity classes, one for each entity declaration. Finally, the `output-generated-files` strategy uses a Java pretty-printer to map a class to a program text and write it to a file with the name of the class and put it in a directory corresponding to the package of the class.

## 4 Programming Patterns: View/Edit Pages

The next step towards full fledged web applications is to create pages for viewing and editing objects in our `DataModel` language. That is, from a data model generate a basic user interface for creating, retrieving, updating and deleting (CRUD) objects. For example, consider the following data model of `Persons` with `Addresses`, and `Users`.

```

entity Person {
    fullname : String
    email    : String
    homepage : String
    photo    : String
    address  : Address
    user     : User
}

entity Address {
    street : String
    city   : String
    phone  : String
}

entity User {
    username : String
    password : String
    person   : Person
}

```

For such a data model we want to generate view and edit pages as displayed in Figures 3 and 4. Implementing this simple user interface requires an understanding of the target architecture. Figure 5 sketches the architecture of a JSF/Seam application for the `editPerson` page in Figure 4. The `/editPerson.seam` client

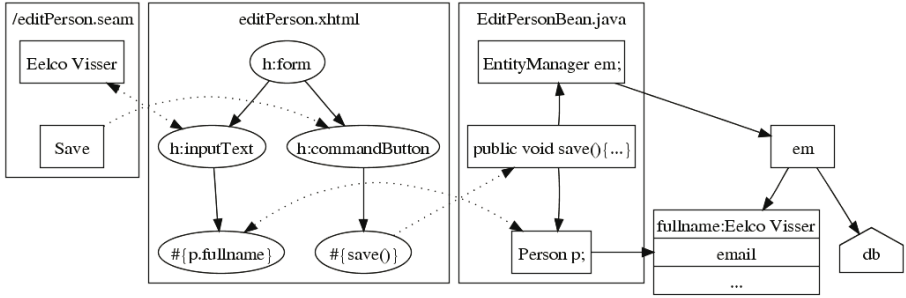


Fig. 5. Sketch of JSF/Seam architecture

view of the page on the far left of Figure 5 is a plain web page implemented in HTML, possibly with some JavaScript code for effects and cascading style sheets for styling. The rendered version of this code is what is shown in Figure 4. The HTML is *rendered* on the server side from the JavaServer Faces (JSF) component model [72] defined in the `editPerson.xhtml` file. In addition to regular HTML layout elements, the JSF model has components that interact with a *session bean*. The `EditPersonBean` session bean retrieves data for the JSF model from the database (and from session and other contexts). For this purpose the session bean obtains an `EntityManager` object through which it approaches the database, with which it synchronizes objects such as `Person p`. When the input field at the client side gets a new value and the form is submitted by a push of the `Save` button, the value of the input field is assigned to the field pointed at by the expression of the `h:inputText` component (by calling the corresponding setter method). Subsequently, the `save()` action method of the session bean, which is specified in the `action` attribute of the `h:commandButton` corresponding to the `Save` button, is called. This method then invokes the entity manager to update the database.

Thus, to implement a view/edit interface for data objects, the generator must produce for each page a JSF XHTML document that defines the layout of the

```

<html ...> ... <body>
<h:form>
  <table>
    <tr><td> <h:outputText value="Fullname"/> </td>
      <td> <h:inputText value="#{editPerson.person.fullname}"/>
    </td> </tr>
    <tr><td><h:commandButton value="Save" action="#{editPerson.save()}" />
      </td> <td></td></tr>
  </table>
</h:form>
</body> </html>

```

Fig. 6. editPage.xhtml with JSF components

user interface and the data used in its elements, and a Seam session bean that manages the objects referred to in the JSF document.

#### 4.1 Generating JSF Pages

Figure 6 illustrates the structure of the JSF XHTML document for the edit page in Figure 4. Besides common HTML tags, the document uses JSF components such as `h:form`, `h:outputText`, `h:inputText`, and `h:commandButton`. Such a document can again be generated using rewrite rules transforming entity declarations to XHTML documents.

```
entity-to-edit-page :
| [ entity x_Class { prop* } ] | ->
%><html ...> ... <body><h:form><table>
  <%= rows ::* %>
  <tr><td>
    <h:commandButton value="Save" action="#{<%=editX%>.save()}" />
  </td><td></td></tr>
</table></h:form></body></html><%=
where editX := <concat-strings>["edit", x_Class]
; x_obj := <decapitalize-string> x_Class
; rows := <map(row-in-edit-form(|editX, x_obj))> props
```

This rule generates the overall setup of an edit page from an entity declaration. Just as was the case with generation of Java code, this rule uses the concrete syntax of XML in the right-hand side of the rule [15]. (The quotation marks `%>` and `<%` were inspired by template engines such as JSP [100]). The XML fragment is syntactically checked at compile-time of the generator and the rule then uses the underlying abstract representation of the fragment. For this syntax embedding we do not have `#` operator to create composite identifiers. Instead names are create by simple string manipulation (concatenation in this case). Note that the ellipses `...` are not part of the formal syntax, but just indicate that some elements were left out of this paper to save space.

The `entity-to-edit-page` rule calls `row-in-edit-form` to generate for each property a row in the table.

```
row-in-edit-form(|editX, x_obj) :
prop@[ x_prop : s ] | ->
%><tr><td><h:outputText value="<%=x_prop%>" /></td>
  <td><%= input %></td></tr><%=
where input := <property-to-edit-component(|editX, x_obj)> prop
```

The left column in the table contains the name of the property, and the right column an appropriate input component, which is generated by the `property-to-edit-component` rule. In the case of the `String` type a simple `inputText` component is generated.

```
property-to-edit-component(|editX, x_obj) :
| [ x_prop : String ] | ->
%><h:inputText value="#{<%=editX%>.<%=x_obj%>.<%=x_prop%>}" /><%=
```

Other types may require more complex JSF configurations. For instance, an entity association (such as the `user` property of `Person`) requires a way to enter references to existing entities. The page in Figure 4 uses a drop-down selection menu for this purpose, which is generated by the following rule:

```
property-to-edit-component(|editX, x_obj) :
  |[ x_prop : s ]| ->
  %> <h:selectOneMenu value="#{<%=editX%>.<%=x_obj%>.<%=x_prop%>}">
    <s:selectItems value="#{<%=editX%>.<%=x_prop%>List}"
      var="<%= x %>" label="#{<%= x %>.name}"
      noSelectionLabel="" />
    <s:convertEntity />
  </h:selectOneMenu <%
  where SimpleSort(_) := s; <defined-java-type> s; x := <new>
```

The `h:selectOneMenu` JSF component sets the value of `editX.x_prop` to the object corresponding to the item selected from the `editX.x_prop#List` list. This list should be provided by the `editX` session bean with the objects to select from, which could be a list of all objects of type `s`.

The generation of a view page is largely similar to the generation of an edit page, but instead of generating an `inputText` component, an `outputText` component is generated:

```
property-to-view-component(|editX, x_obj) :
  |[ x_prop : String ]| ->
  %><h:outputText value="#{<%=editX%>.<%=x_obj%>.<%=x_prop%>}" /><%
```

## 4.2 Seam Session Beans

As explained above, the JSF components get the data to display from an EJB session bean. The Seam framework provides an infrastructure for implementing session beans such that the connections to the environment, such as the application logger and the entity manager, are made automatically via *dependency injection* [46]. To get an idea, here is the session bean class for the `editPerson` page:

```
@Stateful
@Name("editPerson")
public class EditPersonBean implements EditPersonBeanInterface{
  @Logger private Log log;
  @In private EntityManager em;
  @In private FacesMessages facesMessages;
  @Destroy @Remove public void destroy() { }
  // specific fields and methods
}
```

EJB3 and Seam use Java 5 annotations to provide application configuration information within Java classes, instead of the more traditional XML configuration files. The use of annotations is also an alternative to implementing interfaces;

instead of having to implement a number of methods with a fixed name, fields and methods can be named as is appropriate for the application, and declared to play a certain role using annotations.

The `@Stateful` annotation indicates that this is a stateful session bean, which means that it can keep state between requests. The `@Name` annotation specifies the Seam *component* name. This is the prefix to object and method references from JSF documents that we saw in Figure 6. Seam scans class files at deployment time to link component names to implementing classes, such that it can create the appropriate objects when these components are referenced from a JSF instance. The `destroy` method is indicated as the method to be invoked when the session bean is `@Removed` or `@Destroyed`.

The fields `log`, `em`, and `facesMessages` are annotated for *dependency injection* [46]. That is, instead of creating the references for these objects using a factory, the application context finds these fields based on their annotations and injects an object implementing the expected interface. In particular, `log` and `facesMessages` are services for sending messages, for system logging, and user messages, respectively. The `em` field expects a reference to an `EntityManager`, which is the JPA database connection service.

All the above was mostly boilerplate that can be found in any session bean class. The real meat of a session bean is in the fields and methods specific for the JSF page (or pages) it supports. In the view/edit scenario we are currently considering, a view or edit page has a property for the object under consideration. That is, in the case of the `editPerson` page, it has a property of type `Person`:

```
private Person person;
public void setPerson(Person person) { this.person = person; }
public Person getPerson() { return person; }
```

Next, a page is called with URL `/editPerson.seam?person=x`, where  $x$  is the identity of the object being edited. The problem of looking up the value of the `person` parameter in the request object, is also solved by dependency injection in Seam. That is, the following field definition

```
@RequestParameter("person") private Long personId;
```

declares that the value of the `@RequestParameter` with the name `person` should be bound to the field `personId`, where the string value of the parameter is automatically converted to a `Long` value.

To access the object corresponding to the identity passed in as parameter, the following `initialize` method is defined:

```
@Create
public void initialize() {
    if (personId == null) {
        person = new Person();
    } else {
        person = em.find(Person.class, personId);
    }
}
```

The method is annotated with `@Create` to indicate that it should be called upon creation of the bean (and thus the page). The method uses the entity manager `em` to find the object with the given identity. The case that the request parameter is `null` occurs when no identity is passed to the request. Handling this case supports the creation of new objects.

Finally, a push of the `Save` button on the `editPage` leads to a call to the `save()` method of the bean class, which invokes the entity manager to save the changes to the object to the database:

```
public String save() {
    em.persist(this.getPerson());
    return "/person.seam?person=" + person.getId();
}
```

The return value of the method is used to determine the page flow after saving, which is in this case to go to the view page for the object just saved.

### 4.3 Generating Session Beans

Generating the session beans for view and edit pages comes down to taking the programming patterns we saw above and generalizing them by taking out the names related to the entity under consideration and replacing them with holes. Thus, the following rule sketches the structure of such a generator rule:

```
entity-to-session-bean :
| [ entity x_Class { prop* } ] | ->
| [ @Stateful @Name("~viewX")
  public class x_ViewBean implements x_ViewBeanInterface {
    ...
    @Destroy @Remove public void destroy() { }
  } ] |
where viewX := ...; x_ViewBean := ...; x_ViewBeanInterface := ...
```

Such rules are very similar to the generation rules we saw in Section 3.

### 4.4 Deriving Interfaces

A stateful session bean should implement an interface declaring all the methods that should be callable from JSF pages. Instead of having a separate (set of) rule(s) that generates the interface from an entity, such an interface can be generated automatically from the bean class. This is one of the advantages of generating structured code instead of text. The following strategy and rules define a (generic) transformation that turns a Java class into an interface with all the public methods of the class.

```
create-local-interface(|x_Interface) :
class -> | [ @Local public interface x_Interface { ~*methodsdecs } ] |
where methodsdecs := <extract-method-signatures> class
```



```

extract-method-signatures =
  collect(method-dec-to-abstract-method-dec)

method-dec-to-abstract-method-dec :
  MethodDecHead(mods, x , t, x_method, args, y) ->
  AbstractMethodDec(mods, x, t, x_method, args, y)
  where <fetch(?Public())> mods

```

The name of the interface defined is determined by the parameter `x_Interface`. The `collect(s)` strategy is a generic traversal that collects all subterms for which its parameter strategy `s` succeeds. In this case the parameter strategy turns a method declaration header into the declaration of an abstract method, if the former is a public method.

## 5 Programming Patterns: Increasing Coverage

In the previous two sections we analyzed basic patterns for persistent data and view/edit pages in the Seam architecture. We turned these patterns into a simple DSL for data models and a generator for entity classes and view/edit pages. The analysis has taught us the basics of the architecture. We can now use this knowledge to expand the DSL and the generator to cover more sophisticated web applications; that is, to *increase the coverage* of our DSL. Surely we should consider creating custom user interfaces, instead of the rigid view/edit pages that we saw in the previous section. However, before we consider such an extension, we first take a look at the coverage that the data model DSL itself provides.

### 5.1 Strings in Many Flavors

The association types that we saw in the previous sections were either `Strings` or references to other defined entities. While strings are useful for storing many (if not most) values in typical applications, the type name does not provide us with much information about the nature of those data. By introducing application-domain specific value types we can generate a lot of functionality ‘for free’. For example, the following data models for `Person` and `User` still use mostly string valued data, but using alias types the role of those data is declared.

```

entity Person {
  fullname : String
  email    : Email
  homepage : URL
  photo    : Image
  address  : Address
  user     : User
}

entity User {
  username : String
  password : Secret
  person   : Person
}

```

Thus, the type `Email` represents email addresses, `URL` internet addresses, `Image` image locations, `Text` long pieces of text, and `Secret` passwords. Based on these

types a better tuned user interface can be generated. For example, the following rules generate different input fields based on the type alias:

```
property-to-edit-component(|x_component) :
  |[ x_prop : Text ]| ->
  %><h:inputTextarea value="#{<%=x_component%>.<%=x_prop%>}" /><%=

property-to-edit-component(|x_component) :
  |[ x_prop : Secret ]| ->
  %><h:inputSecret value="#{<%=x_component%>.<%=x_prop%>}" /><%=
```

A text-area, providing a large input box, is generated for a property of type `Text`, and a password input field, turning typed characters into asterisks, is generated for a property of type `Secret`.

## 5.2 Collections

Another omission so far was that associations had only singular associations. Often it is useful to have associations with collections of values or entities. Of course, such collections can be modeled using the basic modeling language. For example, define

```
entity PersonList { hd : Person tl : PersonList }
```

to model lists of `Person`. However, in the first place this is annoying to define for every collection, and furthermore, misses the opportunity for attaching standard functionality to collections. Thus, we introduce a general notion of generic sorts, borrowing from Java 5 generics the notation `X<Y, Z>` for a generic sort `X` with sort parameters `Y` and `Z`. For the time being this notation is only used to introduce collection associations using the generic sorts `List` and `Set`. For example, a `Publication` with a list of authors and associated to several projects can then be modeled as follows:

```
entity Publication {
  title      : String
  authors    : List<Person>
  year       : Int
  abstract   : Text
  projects   : Set<Project>
  pdf        : File
}
```

**Many-to-Many Associations.** Introduction of collections requires extending the generation of entity classes. The following rule maps a property with a list type to a Java property with list type and persistence annotation `@ManyToMany`, assuming that objects in the association can be referred to by many objects from the parent entity:

```
property-to-property-code(|x_Class) :
  |[ x_prop : List<y> ]| ->
  |[ @ManyToMany private List<t> x_prop = new ArrayList<t>(); ]|
```

Collections also require an extension of the user interface. This will be discussed later in the paper.

### 5.3 Refining Associations

Yet another omission in the data modeling language is with regard to the nature of associations, i.e. whether they are *composite aggregations* or not. That is, does the referring entity *own* the objects at the other end of the association or not? Since both scenarios may apply, we cannot fix a choice for all applications, but need to let the developer define it for each association. Thus, we refine properties to be either value type (e.g. `title :: String`), composite (e.g. `address <> Address`), or reference (e.g. `authors -> List<Person>`) associations. Figure 7 illustrates the use of special value types, collections, and composite and reference associations.

<pre>entity Publication {   title    :: String   authors  -&gt; List&lt;Person&gt;   year     :: Int   abstract :: Text   projects -&gt; Set&lt;Project&gt;   pdf      :: File }</pre>	<pre>entity Person {   fullname :: String   email    :: Email   homepage :: URL   photo    :: Image   address  &lt;&gt; Address   user     -&gt; User }</pre>	<pre>entity Address {   street :: String   city   :: String   phone  :: String }</pre>
--	---	--

Fig. 7. Data model with composite and reference associations

Based on the association type different code can be generated. For example, the objects in a composite collection, i.e. one in which the referrer owns the objects in the collection, are deleted with their owner. In contrast, in the case of a reference collection, only the references to the objects are deleted when the referring object is deleted. Furthermore, collections of value types are treated differently than collections of entities.

**Unfolding Associations.** One particular decision that can be made based on association type is to unfold composite associations in view and edit pages. This is what is already done in Figures 3 and 4. In Figure 7 entity `Person` has a composite association with `Address`. Thus, an address is owned by a person. Therefore, when viewing or editing a person object we can just as well view/edit the address. The following rule achieves this by *unfolding* an entity reference, i.e. instead of including an input field for the entity, the edit rows for that entity are inserted:

```
row-in-edit-form(|editY) :
  |[ x_prop <> s ]| ->
```

```

%><tr><td><h:outputText value="<%=x_prop%>" /></td></td></tr>
  <%= row* ::*%><%
where <defined-java-type> s
  ; prop* := <properties> s
  ; editYX := <concat-strings>[editY, ".", x_prop]
  ; row* := <map(row-in-edit-form(|editYX))> prop*

```

As an aside, note how the EL expression passed to the recursive call of `row-in-edit-form` is constructed using string concatenation (variable `editYX`). This rather suspect style is an artifact of the XML representation for JSF; the attributes in which EL expressions are represented are just strings without structure. This can be improved upon by defining a proper syntax of JSF XML by embedding a syntax of EL expressions.

## 6 Core Language: Scrap Your Boilertemplate

In the previous sections we have developed a data model DSL with fairly sophisticated types and associations. Furthermore, we have developed a generator for a standard view/edit user interface for objects in the data model. The DSL and generator in the previous sections are based on the analysis of the programming patterns for entity classes and for view/edit pages implemented using JSF and Seam. We factored out the commonality in these programming patterns and turned them into code generation rules with the data modeling DSL as input.

The boilerplate in the generated code is considerable. For example, for the entity `Publication` in Figure 7 the table in Figure 8 contains a breakdown of the source files generated and their size.

With 8 lines of model input, the ratio of generated lines of code to source lines of code is over 100! Now the question is what that buys us. If there was a market for boring view/edit applications this would be great, but in practice we want a much richer application with fine tuned view and edit pages. If we would continue on the path taken here, we could add new sets of generator rules to generate new types of pages. For example, we might want to have pages for searching objects, pages that list all objects of some type, pages providing selections and summaries, etc. But then we would hit an interesting barrier: code duplication in the code generator. The very phenomenon that we were trying to overcome in the first place, code duplication in application code, shows up again, but now in the form of target code fragments

file	LOC
Publication.java	121
EditPublicationBeanInterface.java	56
EditPublicationBean.java	214
ViewPublicationBeanInterface.java	28
ViewPublicationBean.java	117
editPublication.xhtml	181
viewPublication.xhtml	153
total	870

Fig. 8. LOCs generated for `Publication`

that appear in more than one rule (in slightly different forms), sets of generator rule that are very similar, but generate code for a different type of page, etc. In other words, this smells like boilerplate templates, or *boilertemplates*, for short.

The boilerplate smell is characterized by similar target coding patterns used in different templates, only large chunks of target code (a complete page type) considered as a reusable programming pattern, and limited expressivity, since adding a slightly different pattern (type of page) already requires extending the generator.

High time for some generator refactoring. The refactoring we are going to use here is called **find an intermediate language** also known as *scrap your boilerplate*. In order to gain expressivity we need to better cover the variability in the application domain. While implementing the data model DSL, we have explored the capabilities of the target platform, so by now we have a better idea how to implement variations on the view/edit theme by combining the basics of JSF and EJB in different ways. What we now need is a language that sits in between the high-level data modeling language and the low-level details of JSF/Seam and allows us to provide more variability to application developers while still maintaining an advantage over direct programming.

Frameworks such as JSF provide a large number of features (components) for composing user interfaces. It would be tempting to expose all these components to the DSL programmer to allow for maximal expressivity. However, this is not a good idea for productivity. Rather we would like to provide a small set of basic combinators for declaring the UI, and relying on different sets of definitions for their implementation. A good analogue is the complexity of  $\text{T}_{\text{E}}\text{X}$  vs the standardization of  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ .  $\text{T}_{\text{E}}\text{X}$  provides low-level expressivity for typesetting [66]. With it one can do amazingly complex things. However, for common writing of articles, this complexity is not necessary.  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$  harnesses the power of  $\text{T}_{\text{E}}\text{X}$  by providing interfaces (APIs) for building documents with a standardized structure (e.g. `\section`, `\item`, etc.) [69]. Using different style files, documents using this interface can be typeset in very different formats. While one could say that HTML serves a similar goal, the customization to implement a certain style requires quite a bit of HTML coding.

Consider the data model for an entity `ResearchGroup` in Figure 9. While a standard edit page is sufficient for this model, we want to create custom presentation pages that highlight different elements. We will use this example to design a basic language for page flow and presentation. Then we develop a generator that translates page definitions to JSF pages and supporting Seam session beans.

```
entity ResearchGroup {
  acronym    :: String (name)
  fullname   :: String
  mission    :: Text
  logo       :: Image
  members    -> Set<Person>
  projects   -> Set<ResearchProject>
  colloquia  -> Set<Colloquium>
  news       -> List<News>
}
```

Fig. 9. Entity `ResearchGroup`

## 6.1 Page Flow

The pages in Section 4 had URLs of the form `/researchGroup.seam?g=x` with `x` the identity of the object to be presented. Thus, a page has a name and arguments, so analogously to function definitions, a natural syntax for page definitions is:

```
define page researchGroup(g : ResearchGroup) {
  <presentation>
}
```

The parameter is a variable local to the page definition. The URL to request a page uses object identities. Within a page definition the parameter variable can be treated as referring to the corresponding object. Of course, a page definition can have any number of parameters, including zero.

If a page definition is similar to a function definition, page *navigation* should be similar to a function call. Thus, if `pers.group` refers to a `ResearchGroup` object, then `researchGroup(pers.group)` refers to the `researchGroup` page for that object. However, a link in a web page not only requires the destination of the link, but also a name to display it with. The `navigate` form

```
navigate(researchGroup(pers.group)){text(pers.group.acronym)}
```

combines a page reference with a name for the link. The first argument is a ‘call’ to the appropriate page definition. The second argument is a specification of the text for the anchor, which can be a literal string, or a string value obtained from some data object.

## 6.2 Content Markup and Layout

Next we are concerned with presenting the data of objects on a page. For instance, a starting page for a research group might be presented as in Figure 10(a). The layout of such a page is defined using a presentation markup language that can access the data objects passed as arguments to a page. The elements for composition of a presentation are well known from document definition languages such as L<sup>A</sup>T<sub>E</sub>X, HTML, and DocBook and do not require much imagination. We need things such as sections with headers, paragraphs, lists, tables, and text blocks. Figure 10(b) shows the top-level markup for the view in Figure 10(a). It has sections with headers, nested sections, lists, and a text block obtained by taking the `Text` from `group.mission`. The intention of these markup constructs is that they do not allow any configuration for visual formatting. That is, `section` does not have parameters or attributes for declaring the font-size, text color, or text alignment mode. The markup is purely intended to indicate the *structure* of the document. Visual formatting can be realized using cascading style sheets [106], or some higher level styling language.

While the presentation elements above are appropriate for text documents, web pages often have a more two-dimensional layout. That is, in addition to the

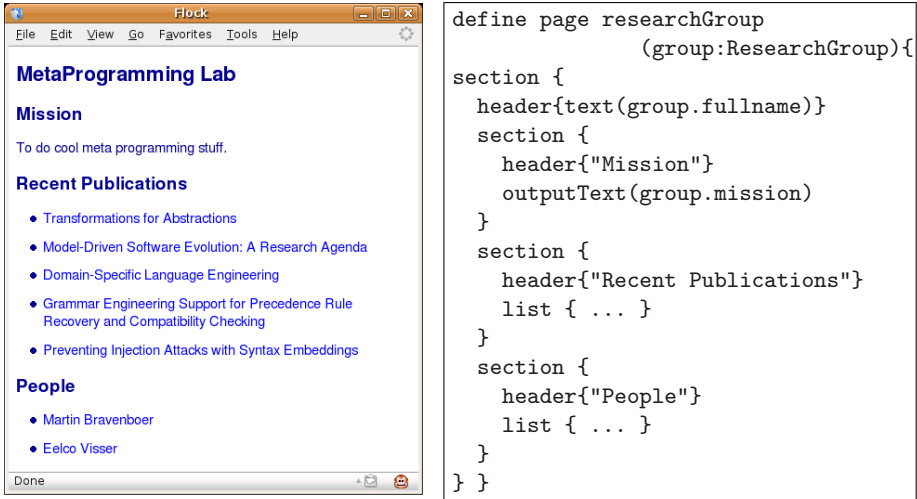


Fig. 10. View of ResearchGroup object: (a) screenshot, (b) markup

body, which is laid out as a text document, a web page often contains elements such as a toolbar with drop-down menus, a sidebar with (contextual) links, a logo, etc. Figure 11 illustrates this by an extension of the ResearchGroup view page of Figure 10 with a sidebar, menubar with drop-down menus and a logo.

WebDSL takes a simple view at the problem of two-dimensional layout. A page can be composed of **blocks**, which can be nested, and which have a name as in the right-hand side page definition in Figure 11. This definition states that a page is composed of two main blocks, **outersidebar** and **outerbody**, which form the left and right column in Figure 11. These blocks are further subdivided into logo and sidebar, and menubar and body, respectively. By mapping **blocks** to **divs** in HTML with the block name as CSS class, the layout can be determined again using CSS.

Other layout problems can be solved in a similar way using CSS. For example, the sidebar in Figure 11 is simply structured as a list:

```
block("sidebar"){
  list {
    listitem { navigate(researchGroup(group)){text(group.acronym)} }
    listitem { navigate(groupMembers(group)){"People"} }
    listitem { navigate(groupPublications(group)){"Publications"} }
    listitem { navigate(groupProjects(group)){"Projects"} list{ ... } }
  }
}
```

Using CSS the default indented and bulleted list item layout can be redefined to the form of Figure 11 (no indentation, block icon for sub lists, etc.).

Drop-down menus can be defined using a combination of CSS and some javascript, which can be generated from a declarative description of the menus.

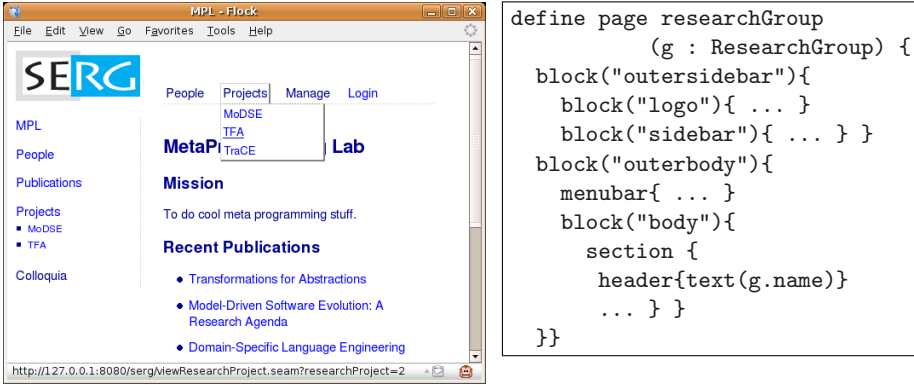


Fig. 11. Two-dimensional layout with logos, sidebars, drop-down menus

For example, the drop-down menus of Figure 11 are defined using elements such as `menu` and `menuitem`:

```

menubar{
  menu{ menuheader{"People"} menuitem{...} ...}
  menu{ menuheader{"Projects"} menuitem{...} ...}
  ...
}

```

Thus, using simple structural markup elements without visual configuration, a good separation of the definition of the structure of a page and its visual layout using cascading style sheets can be achieved. This approach can be easily extended to more fancy user interface elements by targeting java-script in addition to pure HTML. There again the aim should be to keep WebDSL specifications free of visual layout.

### 6.3 Language Constructs

We have now developed a basic idea for a page presentation language with concepts such as sections, lists, and blocks. The next question is how to define a language in which we can write these structures. The approach that novice language designers tend to take is to define a syntactic production for each markup element. Experience shows that such language definitions become rather unwieldy and make the language difficult to extend. To add a new markup construct, the syntax needs to be extended, and thus all operations that operate on the abstract syntax tree. Lets be clear that a rich syntax is a good idea, but only where it concerns constructs that are really different. Thus, rather than introducing a syntactic language construct for each possible markup element, we use the generic *template call* syntactic construct (why it is called *template call* will become clear later).



**Template Call.** A template call has the following form:

$$f(e_1, \dots, e_m) \{elem_1 \dots elem_n\}$$

That is, a template call has a **name**  $f$ , a list of *expressions*  $e_1, \dots, e_m$  and a list of *template elements*  $elem_1 \dots elem_n$ . Both the expression and element argument lists are optional.

The name of the call determines the type of markup and is mapped by the back-end to some appropriate implementation in a target markup language.

The element arguments of a call are nested presentation elements. For example, a **section** has as arguments, among others, headers and paragraphs

```
section{ header{ ... } par{ ... } par{ ... } }
```

a **list** has as elements **listitem**s

```
list { listitem { ... } ... }
```

and a **table** has **rows**

```
table { row{ ... } row{ ... } }
```

The expression arguments of a call can be simple strings, such as the name of a block:

```
block("menu") { list { ... } }
```

However, mostly expressions provide the mechanism to access data from entity objects. For example, the **text** element takes a reference to a string value and displays it:

```
text(group.name)
```

Similarly, the **navigate** element takes page invocation as expression argument and nested presentation elements to make up the text of the link.

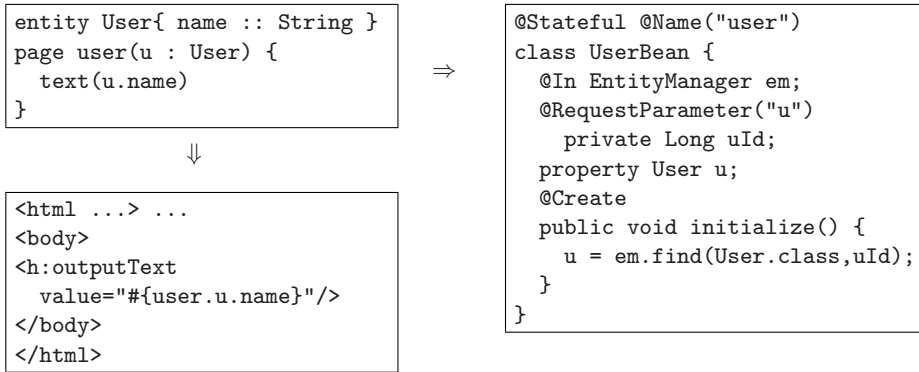
```
navigate(publication(pub)){text(pub.name)}
```

**Iteration.** While the template call element is fairly versatile, it is not sufficient for everything we need to express. In particular, we need a mechanism for iterating over collections of objects or values. This is the role of the **for** iterator element, which has the following concrete syntax:

$$\text{for}( x : s \text{ in } e ) \{elem*\}$$

The reason that this construct cannot be expressed using the syntax of a template call is the variable which is bound locally in the body of the iterator. The iterator is typically used to list objects in a collection. For example, the following fragment of a page involving **g** of type **ResearchGroup**, which has a collection of **projects**, presents a list of links to the projects in **g**.

```
list {
  for(p : ResearchProject in g.projects) {
    listitem { navigate(researchProject(p)){text(p.acronym)} }
  } }
```



**Fig. 12.** Mapping from page definition (upper left) to session bean (right) and JSF (lower left)

## 6.4 Mapping Pages to JSF+Seam

In Section 4 we saw how to generate a web application for viewing and editing objects in a data model using a row-based interface targeting the JSF and Seam frameworks. We can now use the knowledge of that implementation approach to define a mapping from the new fine grained presentation elements to JSF+Seam. Figure 12 illustrates the mapping for a tiny page definition. The mapping from a page definition to JSF involves creating an XML JSF document with as body the body of the page definition, mapping presentation elements to JSF components and HTML, and object access expressions to JSF EL expressions. The mapping from a page definition to a Seam session bean involves creating the usual boilerplate, `@RequestParameters` with corresponding properties (using `property` as an abbreviation to indicate a private field with a getter and a setter), and appropriate statements in the initialization method. In the rest of this section we consider some of the translation rules.

## 6.5 Generating JSF

The mapping from page elements to JSF is a fairly straightforward set of recursive rules that translate individual elements to corresponding JSF components. Note that while the syntax of template calls is generic, the mapping is *not* generic. First, while the syntax allows to use arbitrary identifiers as template names, only a (small) subset is actually supported. Second, there are separate generation rules to define the semantics of different template calls. The essence of the domain-specific language is in these code generation rules. They store the knowledge about the target domain that we reuse by writing DSL models. We consider some representative examples of the mapping to JSF.

**Text.** The rule for `text` is a base case of the mapping. A `text(e)` element displays the string value of the `e` expression using the `outputText` JSF component.

```
elem-to-xhtml :
  |[ text(e) ]| -> %> <h:outputText value="<%=e1%"/> <%
  where e1 := <arg-to-value-string> e
```

The `arg-to-value-string` rules translate an expression to a JSF EL expression.

**Block.** The rule for `block` is an example of a recursive rule definition. Note the application of the rule `elems-to-xhtml` in the antiquotation.

```
elem-to-xhtml :
  |[ block(str){elem*} ]| ->
  %><div class="<%= str %>">
    <%= <elems-to-xhtml> elem* ::*%>
  </div><%
```

The auxiliary `elems-to-xhtml` strategy is a map over the elements in a list:

```
elems-to-xhtml = map(elem-to-xhtml)
```

**Iteration.** While iteration might seem one of the complicated constructs of WebDSL, its implementation turns out to be very simple. An iteration such as the following

```
list{ for ( project : ResearchProject in group.projectsList ) {
  listitem { text(group.project.acronym) }
}}
```

is translated to the JSF `ui:repeat` component, which iterates over the elements of the collection that is produced by the expression in the `value` attribute, using the variable named in the `var` attribute as index in the collection.

```
<ul> <ui:repeat var="project"
  value="#{researchGroup.group.projectsList}">
  <li> <h:outputText value="#{project.acronym}" </li>
</ui:repeat> </ul>
```

This mapping is defined in the following rule:

```
elem-to-xhtml :
  |[ for(x : s in e) { elem1* } ]| ->
  %><ui:repeat var="<%= x %>" value="<%= e1 %>">
    <%= elem2* ::*%>
  </ui:repeat><%
  where e1 := <arg-to-value-string> e
        ; elem2* := <elems-to-xhtml> elem1*
```

**Navigation.** The translation of a navigation element is slightly more complicated, since it involves context-sensitive information. As example, consider the following `navigate` element:

```
navigate(viewPerson(prs)){text(prs.name)}
```

Such a navigation should be translated to the following JSF code:

```
<s:link view="/person.xhtml">
  <f:param name="p" value="#{prs.id}" />
  <h:outputText value="#{prs.name}" />
</s:link>
```

While most of this is straightforward, the complication comes from the parameter. The `f:param` component defines for a URL parameter the name and value. However, the name of the parameter (`p` in the example) is not provided in the call (`person`). The following rule solves this by means of the dynamic rule `TemplateArguments`:

```
elem-to-xhtml :
| [ navigate(p(e*)) {elem1*} ] | ->
%><s:link view = "/<%= p %>.xhtml">
  <%= <conc>(param*,elem2*) ::*%>
  </s:link><%
where <IsPage> p
  ; farg* := <TemplateArguments> p
  ; param* := <zip(bind-param)> (farg*, e*)
  ; elem2* := <elems-to-xhtml> elem1*
```

In a similar way as `declare-entity` in Section 3 declares the mapping of declared entities to Java types, for each page definition, dynamic rules are defined that (1) record the fact that a page with name `p` is defined (`IsPage`), and (2) map the page name to the list of formal parameters of the page (`TemplateArguments`). Then, creating the list of `f:params` is just a matter of zipping together the list of formal parameters and actual parameters using the following `bind-param` rule:

```
bind-param :
(| [ x : $X ] |, e) ->
%><f:param name="<%= x %>" value="<%= el %>" /><%
where <defined-java-type> $X
  ; el := <arg-to-value-string> | [ e.id ] |
```

The rule combines a formal parameter `x` and an actual parameter expression `e` into an `f:param` element with as name the name of the formal parameter, and as value the EL expression corresponding to `e`.

**Sections.** A final example is that of nested sections. Contrary to the custom of using fixed section header levels, WebDSL assigns header levels according to the section nesting level. Thus, a fragment such as

```
section { header{"Foo"} ... section { header{"Bar"} ... } }
```

should be mapped to HTML as follows:

```
<h1>Foo</h1> ... <h2>Bar</h2> ...
```

This is again an example of context-sensitive information, which is solved using a dynamic rule. The rules for `section` just maps its argument elements. But before making the recursive call, the `SectionDepth` counter is incremented.

```
elem-to-xhtml :
  [| section() { elem1* } ]| -> %> elem2* <%
  where {| SectionDepth
        : rules( SectionDepth := <(SectionDepth <+ !0); inc> )
        ; elem2* := <elems-to-xhtml> elem1*
        |}
```

The dynamic rule scope `{| SectionDepth : ... |}` ensures that the variable is restored to its original value after translating all elements of the section.

The rule for the `header` element uses the `SectionDepth` variable to generate an HTML header with the correct level.

```
elem-to-xhtml :
  [| header(){ elem* } ]| ->
  %><~n:tag><%= <elems-to-xhtml> elems :.*%></~n:tag><%
  where n := <SectionDepth <+ !1>
        ; tag := <concat-strings>["h", <int-to-string> n]
```

Interesting about this example is that the dynamic rules mechanism makes it possible to propagate values during translation without the need to store these values in parameters of the translation rules and strategies.

## 6.6 Generating Seam Session Beans

The mapping from page definitions to Seam is less interesting than the mapping to JSF. At this point there are only two aspects to the mapping. First, a page definition gives rise to a compilation unit defining a stateful session bean using the name of the page as Seam component name, and the usual boilerplate for session beans.

```
page-to-java :
  [| define page x_page(farg*) { elem1* } ]| ->
  [| @Stateful @Name("~x_page")
    public class x_Page#Bean implements x_Page#BeanInterface {
      @In private EntityManager em;
      @Create public void initialize() { bstm* }
      @Destroy @Remove public void destroy() {}
      cbd*
    }|]
  where x_Page := <capitalize-string> x_page
        ; cbd* := <map(argument-to-bean-property)> farg*
        ; bstm* := <map(argument-to-initialization)> farg*
```

Second, for each argument of the page, a `@RequestParameter` with corresponding property is generated as discussed in Section [4](#).

```

argument-to-bean-property :
|[ x : x_Class ]| ->
|[ @RequestParam("~x") private Long x#Id;
private x_Class x;
public void set#x(x_Class x) { this.x = x; }
public x_Class get#x() { return x; } ]|

```

Finally, code is generated for initializing the property by loading the object corresponding to the identity when the session bean is created.

```

argument-to-initialization :
|[ x : x_Class ]| ->
|[ if (x_Id == null) { x = new x_Class(); }
else { x = em.find(x_Class.class, x_Id); } ]|
where x_Id := <concat-strings>[x, "Id"]

```

## 6.7 Boilertemplate Scrapped

This concludes the generator refactoring ‘scrap your boilertemplate’. We have introduced a language that provides a much better coverage of the user interface domain, and which can be used to create a wide range of presentations. The resulting mapping now looks much more like a compiler; each language construct expresses a single concern and the translation rules are fairly small. Next we consider several extensions of the language.

## 7 Core Language: Extensions

In the first design of the core language for page definitions some aspects were ignored to keep things simple. In this section we consider several necessary extensions.

### 7.1 Type Checking

Java is a statically typed language, which ensures that many common programming errors are caught at compile-time. Surprisingly, however, this does not ensure that web applications developed with frameworks such as JSF and Seam are free of ‘type’ errors after compilation.

JSF pages are ‘compiled’ at run-time or deployment-time, which means that many causes of errors are unchecked. Typical examples are missing or non-supported tags, references to non-existing properties, and references to non-existing components. Some of these errors cause run-time exceptions, but others are silently ignored.

While this is typical of template-like data, it is interesting to observe that a framework such as Seam, which relies on annotations in Java programs for configuration, has similar problems. The main cause is that Seam component annotations are scanned and linked at deployment-time, and not checked at

compile-time for consistency. Thus, uses of components (e.g. in JSF pages) are not checked. Dependency injection enables loose coupling between components/classes, but as a result, the compiler can no longer check data flow properties, such as guaranteeing that a variable is always initialized before it is used. Another symptom of interacting frameworks is the fact that a method that is not declared in the `@Local` interface of a session bean, is silently ignored when invoked in JSF.

Finally, JPA and Hibernate queries are composed using string concatenation. Therefore, syntactic and type errors (e.g. non-existing column) become manifest only at run-time. Most of these types of errors will show up during testing, but vulnerabilities to injection attacks in queries only manifest themselves when the system is attacked, unless they are tested for.

**Type Checking WebDSL.** To avoid the kind of problems mentioned above, WebDSL programs are statically type checked to find such errors early. The types of expressions in template calls are checked against the types of definition parameters and properties of entity definitions to avoid use of non-existing properties or ill-typed expressions. The existence of pages that are navigated to is checked. For example, for the following WebDSL program

```
entity User { name :: String }
define page user(u : User) {
  text(u.fullname)
  text(us.name)
  navigate(foo()){ "bar" }
}
```

the type checker finds the following errors:

```
$ dsl-to-seam -i test.app
[error] entity 'User' has no property 'fullname'
[error] variable 'us' has no declared type
[error] link to undefined page 'foo'
```

**Type Checking Rules.** The type checker is a transformation on WebDSL programs, which checks the type correctness of expressions and annotates expressions with their type. These annotations will turn out useful when considering higher-level abstractions. The following type checking rule for the iterator construct, illustrates some aspects of the implementation of the type checker.

```
typecheck-iterator :
  |[ for(x : s in e1){elem1*} ]| -> |[ for(x : s in e2){elem2*} ]|
  where in-tc-context(id
    ; e2 := <typecheck-expression> e1
    ; <should-have-list-type> e2
    ; { | TypeOf
      : if not(<java-type> s) then
        typecheck-error(["index ", x, " has invalid type ", s])
      else
```

```

        rules( TypeOf : x -> s )
        end
        ; elems2 := <typecheck-page-elements> elems1
        |}
        | ["iterator ", x, "/" ] )

```

First, the type checker performs a *transformation*, that is, rather than just checking, constructs are transformed by adding annotations. Thus, in this rule, the iterator expression and elements in the body are replaced by the result of type checking them. Next, constraints on the construct are checked and errors reported with `typecheck-error`. The `in-tc-context` wrapper strategy is responsible for building up a context string for use in error messages. Finally, the local iterator variable `x` is bound to its type in the `TypeOf` dynamic rule [20]. The dynamic rule scope `{| TypeOf : ... |}` ensures that the binding is only visible while type checking the body of the iterator. The binding is used to annotate variables with their type, as expressed in the `typecheck-variable` rule:

```

typecheck-variable :
  Var(x) -> Var(x){Type(t)}
  where if not(t := <TypeOf> x) then
    typecheck-error(|["variable ", x, " has no declared type"])
    ; t := "Error"
  end

```

## 7.2 Data Input and Actions

The language of the previous section only dealt with *presentation* of data. Data *input* is of course an essential requirement for interactive web applications. To make edit pages, we need constructs to create input components that bind data to object fields, forms, and buttons and actions to save the data. Figure 13 shows a WebDSL page definition for a simple edit page with a single input field and a `Save` button, as well as the mapping to JSF and Java/Seam. The language constructs are straightforward. The `form` element builds a form, the `inputString(e)` element creates an input field bound to the contents of the field pointed at by `e`, and the `action` element creates a button, which executes a call to a defined action when pushed. The mapping to Seam is straightforward as well. The action definition is mapped to a method of the session bean.

**Action Language.** The statement language that can be used in action definitions is a simple imperative language with the usual constructs. Assignments such as `person.blog := Blog{title := name}`; bind a value to a variable or field. Method calls such as `publication.authors.remove(author)`; invoke an operation on an object. Currently, the language only supports a fixed set of methods, such as some standard operations on collections, and persistence operations such as `save`. The latter can be applied directly to entity objects, hiding the interaction with an entity manager from the WebDSL developer. The return statement is somewhat unusual, as it is interpreted as a page-flow directive, that



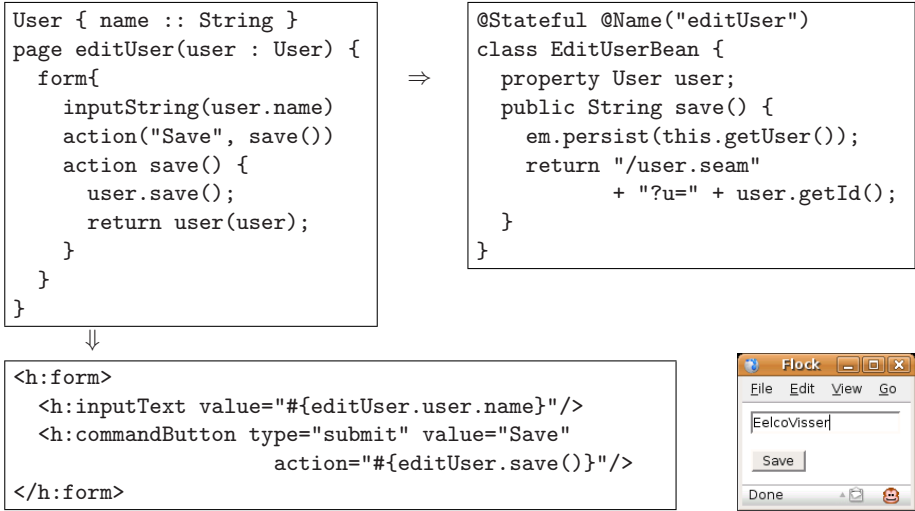


Fig. 13. Mapping form, input field, and action to JSF and Java/Seam

is, a statement `return user(u);` is interpreted as a page redirect with appropriate parameters. Conditional execution is achieved with the usual control-flow constructs.

Expressions consist of variables, constant values (e.g. strings, integers), field access, and object creation. Rather than having to assign values to fields after creating an object, this can be done with the creation expression. Thus, object creation has the form `Person{ name := e ... }`, where fields can be directly given a value. There is also special syntax for creating sets (`{e1, e2, ...}`) and lists (`[e1, e2, ...]`).

**Java Embedding.** The current design of the action language is somewhat ad hoc and should be generalized. A conventional critique of domain-specific languages is that they require the redesign of such things as statements and expressions, which is hard to get right and complete.

An alternative approach would be to directly embed the syntax of Java statements and expressions, and insert the embedded Java fragments into the generated session bean classes. This would give complete access to the full expressivity of Java. Indeed this is what is done with the Hibernate Query Language later in this section. However, Java is a large and complex language; an embedding would entail importing a type checker for Java as well. Furthermore, it would entail tying the DSL to the Java platform and preclude portability to other platforms. HQL and SQL are more portable than Java. That is, as long as we rely on a platform with a relational database, chances are that we can access the data layer through an SQL query. A more viable direction seems to keep the action language simple, but provide a foreign function interface, which gives access to functionality implemented in external libraries to be linked with the application.

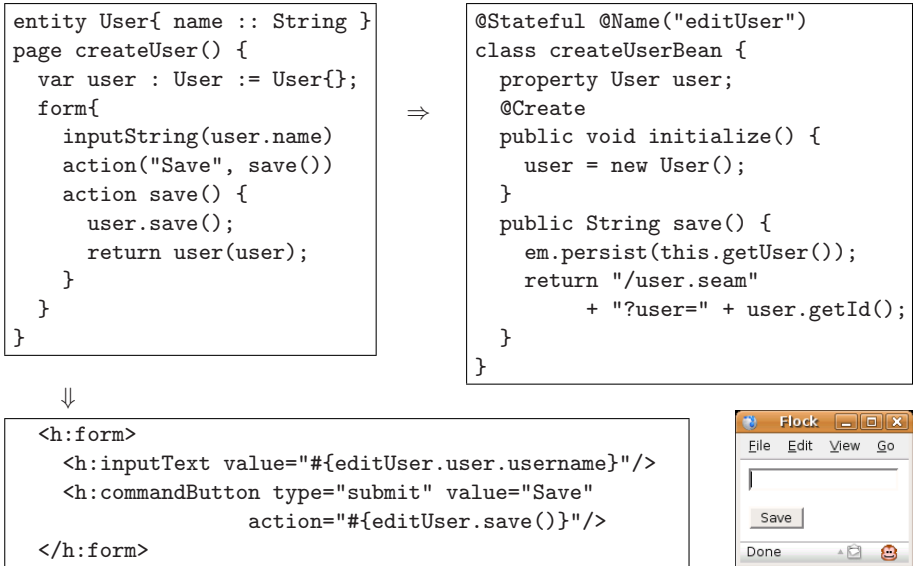


Fig. 14. Page local variables

### 7.3 Page Local Variables

So far we have considered pages that operate on objects passed as parameters. Sometimes it is necessary for a page to have local variables. For example, a page for creating a new object cannot operate on an existing object and needs to create a fresh object. Page local variables support this scenario. Figure 14 illustrates the use of a local variable in the definition of a page for creating new `User` objects, which is mostly similar to the edit page, except for the local variable.

### 7.4 Queries

The presentation language supports the access of data via (chained) field accesses. Thus, if we have an object, we can access all objects to which it has (indirect) associations. Sometimes, we may want to access objects that are not available through associations. For example, in the data model in Figure 7, a `Publication` has a list of `authors` of type `User`, but a `User` has no (inverse) association to the publications he is author of. In these situations we need a query mechanism to reconstruct the implicit association. In general, queries allow filtering of data.

There is no need to invent a DSL for querying. The Hibernate Query Language (HQL), an adaptation of the relational query language SQL to ORM, provides an excellent query language [10]. To make HQL available in WebDSL we follow the language embedding pattern described in earlier work [102]. Figure 15 illustrates the embedding and its implementation. The query retrieves the publications for

```

entity User{ name :: String }
entity Publication{ authors -> List<User> }
page user(user : User) {
  var pubs : List<Publication> :=
    select pub from Publication as pub, User as u
      where (u = ~user) and (u member of pub.authors)
      order by pub.year descending;
  for(p : Publication in pubs) { ... }
}

```

⇓

```

class UserBean {
  property List<Publication> pubs;
  @Factory("pubs") public void initPubs() {
    pubs = em.createQuery(
      "select pub from Publication as pub, User as u" +
      " where (u = :param1) and (u member of pub.authors)" +
      " order by pub.year descending"
    ).setParameter("param1", this.getUser())
    .getResultList();
  }
}

```

**Fig. 15.** Mapping embedded HQL queries to string-based query construction in Java

which the `user` is an author. An HQL query is added to the WebDSL syntax as an expression. For now we assume the result of a query is assigned to a local page variable, which can then be accessed anywhere on the page. Queries can refer to values of page objects by means of the antiquotation `~`. In Figure 15, this is used to find the user with the same identity as the `user` object of the page. The query is translated to a `@Factory` method, which uses the entity manager to create the query using string composition. Antiquoted expressions become parameters of the query.

While the use of HQL in WebDSL does not provide a dramatic decrease in code size, there are some other advantages over the use of HQL in Java. In Java programs, Hibernate queries are composed as strings and parsed at run-time. This means that syntax errors in queries are only caught at run-time, which is hopefully during testing, but maybe during production if testing is not thorough. The `getParameter` mechanism of HQL takes care of escaping special characters to avoid injection attacks. However, use of this mechanism is not enforced and developers can splice values directly into the query string, so the risk of injection attacks is high. In WebDSL, queries are not composed as strings, but integrated in the syntax of the language. Thus, syntactic errors are caught at compile-time and it is not possible to splice in strings without escaping. This embedding of HQL in WebDSL is a variant of the StringBorg approach, which provides a safe way of embedding query-like languages without the risk of injection attacks [16].

<pre>entity Blog {   title  :: String (name)   author -&gt; Person   entries &lt;&gt; List&lt;BlogEntry&gt; }</pre>	<pre>entity BlogEntry {   title  :: String (name)   created :: Date   intro  :: Text }</pre>
---	--

**Fig. 16.** Data model for blogs and blog entries

Another advantage is that the WebDSL type checker can check the consistency of queries against the data model and local variable declarations. The consistency of HQL queries in Java programs is only checked at run-time.

## 8 Abstraction Mechanisms: Templates and Modules

In the previous two sections we have extended the data modeling language with a core language for presentation, data input, and page flow. The generator now encapsulates a lot of knowledge about basic implementation patterns. The resulting language provides the required flexibility such that we can easily create different types of pages without having to extend or change the generator. However, this same flexibility entails that page definitions will consist of fragments that occur in other definitions as well. We need to balance the flexibility of the core language with abstraction mechanisms that allow developers to abstract from low-level implementation patterns. We can distinguish two forms; generative and non-generative abstraction mechanisms.

Literal code duplication can be addressed by providing a mechanism for naming and parametrizing code fragments. In this section we extend the language with *templates*, named pieces of code with parameters and hooks. Next, we add *modules*, named collections of definitions defined in a separate file, which can be imported into other modules. Modules are essential for organizing a code base and to form a library of reusable code. These mechanisms are *non-generative*, in the sense that the definitions of patterns are done by the DSL programmer and do not require an extension of the generator.

In the next section, we consider *syntactic abstractions*, extensions to the language providing higher-level abstractions, which are implemented by means of ‘model-to-model’ transformations in the generator. These abstraction mechanisms are *generative* (like the ones we saw before). Implementation in the generator allows reflection over the model and non-local transformations.

### 8.1 Reusing Page Fragments with Template Definitions

Template definitions provide a mechanism for giving a name to frequently used page fragments. A *template definition* has the form

```
define f(farg*){elem*}
```

with  $f$  the name of the template,  $farg^*$  a list of formal parameters, and  $elem^*$  a list of template elements. The use of a defined template in a template call,

leads to the replacement of the call by the body of the definition. The markup elements we introduced in Section 6 are also template calls; these are not defined by template definitions, but by the generator. To illustrate the use of template definitions, we consider pages such as the one in Figure 17. The body of the page presents entries in a blog, as represented in the data model in Figure 16, but surrounding that are elements that appear in many other pages as well. The following parameterless template definitions define the literal fragments logo, footer, and menu:

```
define logo() { navigate(home()){image("/img/serg-logo.png")} }
define footer() {
  "generated with "
  navigate(url("http://www.strategoxt.org")){"Stratego/XT"}
}
define menubar() {
  menu{ menuheader{"People"} for(p : Person){ menuitem{...} } } ...
}
```

Such fragments can be reused in many pages, as in the following page definition:

```
define page home() {
  block("menubar"){ logo() menubar() }
  section{ ... }
  footer()
}
```

Literal template definitions are of limited use. To support reuse of partial fragments, which have holes that should be filled in by the reuse context, templates can have hooks in the form of template calls that can be locally (re)defined. For example, the following main template calls logo, sidebar, menu, body, and footer.

```
define main() {
  block("outsidebar") { logo() sidebar() }
  block("outerbody") {
    block("menubar") { menubar() }
    body()
    footer()
  }
}
```

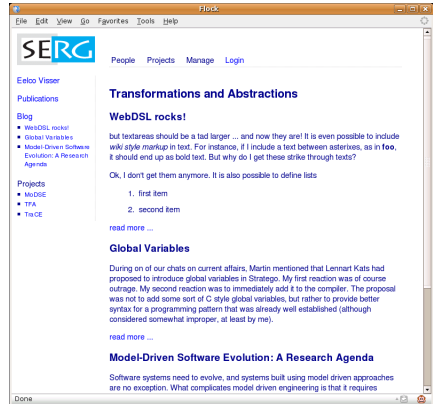


Fig. 17. Instance of blog page

Some of these templates may have a global definition, such as the ones above, but others may be defined locally in the context where `main` is called. For example, the following page definition calls the `main` template and defines `sidebar` and `body` (overriding any top-level definitions), thus instantiating the calls to these templates in the definition of `main`:

```
define page blog(b : Blog) {
  main()
  define sidebar(){ blogSidebar(b) }
  define body() {
    section{ header{ text(b.title) }
      for(entry : BlogEntry in b.entries) { ... }
    } } }
```

Templates may need to access objects. Therefore, templates can have parameters. For example, the following definition for a sidebar defines links specific to a particular `Person` object `p`.

```
define personSidebar(p : Person) {
  list {
    listitem { navigate(person(p)){text(p.name)} }
    listitem { navigate(personPublications(p)){"Publications"} }
    listitem { navigate(blog(p.blog)){"Blog"} blogEntries() }
    listitem { "Projects" listProjectAcronyms(p) }
  } }
```

This allows templates to be reused in different contexts. For example, the template above can be used to create the sidebar for the view page for a `Person`, as well as for the publications page of that person.

```
define page person(p : Person) {
  main()
  define sidebar() { personSidebar(p) } ...
}
define page personPublications(p : Person) {
  main()
  define sidebar() { personSidebar(p) } ...
}
```

Note that the template mechanism is a form of dynamic scoping; template calls may be instantiated depending on the use site of the enclosing template definition. However, the variables used in expressions are statically bound and can only refer to lexically visible variable declarations, i.e. template parameters, local variables, or global variables. The combination is similar to method overriding in object oriented languages, where variables are lexically scoped, but method invocations may be dynamically bound to different implementations. The template calls in a template definition provide a requires interface of internal variation points.

**Template Expansion.** Template expansion is a context-sensitive transformation, which again relies on dynamic rules for its implementation. For each template definition a dynamic rule `TemplateDef` is defined that maps the name of the template to its complete definition.

```
declare-template-definition =
  ?def@[ [ define mod* x(farg*){elem*} ] |
    ; rules( TemplateDef : x -> def )
```

The dynamic rule is used to retrieve the definition when encountering a template call. Subsequently, all bound variables in the definition are renamed to avoid capture of free variables.

```
expand-template-call :
  |[ x(e*){elem1*} ] | -> |[ elem2* ] |
  where <TemplateDef; rename> x => |[define mod* x(farg*){elem3*}] |
    ; { | Subst
      : <zip(bind-variable)> (farg*, <alltd(Subst)> e*)
      ; elem2* := <map(expand-element)> elem3*
      ; str := x
      | }
```

The formal parameters of the template are bound to the actual parameters of the call in the dynamic rule `Subst`:

```
bind-variable = ?(Arg(x, s), e); rules( Subst : Var(x) -> e )
```

## 8.2 Modules

A module system allows a code base to be organized into coherent and possibly reusable units, which is a requirement for building a library. Module systems come in different levels of complexity. Module systems supporting separate compilation can become quite complex, especially if the units of compilation in the DSL do not match the units of compilation of the target platform. For this version of WebDSL a very simple module system has been chosen that supports distributing functionality over files, without separate compilation. A module is a collection of domain model and template definitions and can be imported into other modules as illustrated in Figures 18 and 19. The generator first reads in all imported modules before applying other transformations. The implementation of import chasing is extremely simple:

```
import-modules =
  topdown(try(already-imported <+ import-module))

already-imported :
  Imports(name) -> Section(name, [])
  where <Imported> name
```

```

module publications
section domain definition
  Publication {
    title    :: String (name)
    year     :: Int
    authors  -> List<Person>
    abstract :: Text
  }
section presenting publications
define showPublication(pub : Publication) {
  for(author : Person in pub.authors){
    navigate(person(author)){text(author.name)} " , " }
    navigate(publication(pub)){text(pub.name)} " , "
    text(pub.year) "."
  }
}

```

Fig. 18. Module definition

```

application org.webdsl.serg
imports templates
imports people
imports blog
imports publications

```

Fig. 19. Application importing modules

```

import-module :
  Imports(name) -> mod
  where mod := <parse-webdsl-module>FILE(<concat-strings>[name, ".app"])
            ; rules( Imported : name )

```

The dynamic rule `Imported` is used to prevent importing a module more than once.

## 9 Abstraction Mechanisms: Syntactic Sugar

With the core language introduced in Sections 6 and 7 we have obtained expressivity to define a wide range of presentations. With the templates and modules from the previous section we have obtained a mechanism for avoiding code duplication. However, there are more generic patterns that are tedious to encode for which templates are not sufficient. Even if a language provides basic expressivity, it may not provide the right-level of abstraction. So if we encounter reoccurring programming patterns in our DSL, the next step is to design higher-level abstractions that capture these patterns. Since the basic expressivity is present we can express these abstractions by means of transformations from the extended



DSL to the core DSL. Such transformations are known as *desugarings*, since the high-level abstractions are known as *syntactic sugar*. In this section we discuss three abstractions and their corresponding desugarings.

## 9.1 Output Entity Links

A convention in WebDSL applications is to define for each entity type a corresponding page definition for viewing objects of that type with the name of the entity in lowercase. For example, for entity `Publication`, a page definition `publication(p : Publication)` is defined. Given an object, say `pub : Publication`, creating a link to such a page is then realized with `navigate` as follows:

```
navigate(publication(pub)){text(pub.name)}
```

While not a lot of code to write, it becomes tedious, especially if we consider that the code can be derived from the *type* of the variable. Thus, we can replace this pattern by the simple element

```
output(pub)
```

This abstraction is implemented by the following desugaring rule, which uses the *type* of the expression to determine that the expression points to an entity object:

```
DeriveOutputSimpleRefAssociation :
  |[ output(e){} ]| -> |[ navigate($y(e)){text(e.name)} ]|
  where |[ $Y ]| := <type-of> e
         ; <defined-java-type> |[ $Y ]|
         ; $y := <decapitalize-string> $Y
```

This desugaring is enabled by the type annotations on expressions produced by the type checker. Similar desugaring rules can be defined for other types, as illustrated by the following rules:

```
DeriveOutputText :
  |[ output(e){} ]| -> |[ navigate(url(e)){text(e)} ]|
  where |[ URL ]| := <type-of> e
```

```
DeriveOutputText :
  |[ output(e){} ]| -> |[ image(e){} ]|
  where |[ Image ]| := <type-of> e
```

As a consequence of this abstraction, it is sufficient to write `output(e)` to produce the default presentation of the object indicated by the expression `e`.

## 9.2 Editing Entity Collection Associations

Editing a collection of entities is not as simple as editing a string or text property. Instead of typing in the value we need to select an existing object from some

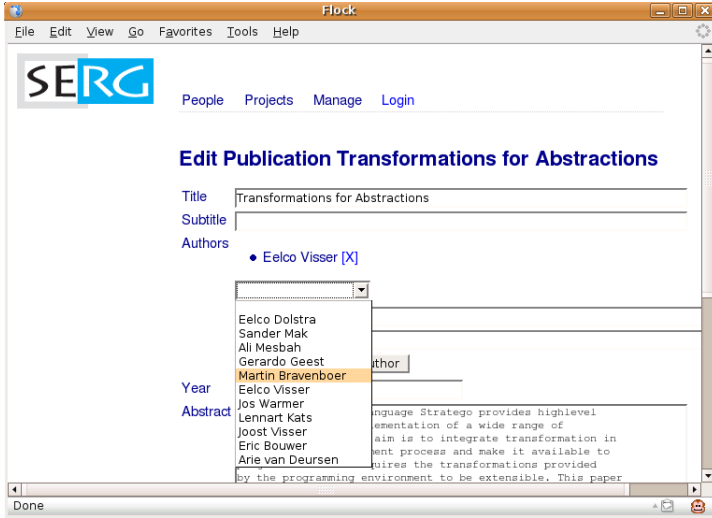


Fig. 20. Editing collection association

kind of menu. Consider the edit page for a publication in Figure 20. Editing the `authors` association requires the following ingredients: a list of names of entities already in the collection; a link `[X]` to remove the entity from the collection; a select menu to add a new (existing) entity to the collection. This is implemented by the following WebDSL pattern:

```
list { for(person : Person in publication.authors) {
  listitem{ text(person.name) " "
            actionLink("[X]", removePerson(person)) }
} }
select(person : Person, addPerson(person))
action removePerson(person : Person) {
  publication.authors.remove(person);
}
action addPerson(person : Person) {
  publication.authors.add(person);
}
```

The `select` creates a drop-down menu with (names of) objects of some type. Upon selection of an element from the list, the corresponding action (`addPerson` in this case), is executed. This fragment illustrates the flexibility of the presentation language; a complex interaction pattern can be composed using basic constructs. However, repeating this pattern for each entity association is tedious. Creating this pattern can be done automatically by considering the type of the association, which is expressed by the first desugaring rule in Figure 21. Thus, `input(pub.authors)` is now sufficient for producing the implementation of an

```

DeriveInputAssociationList :
  elem|[ input(e){} ]| ->
  elem|[ list { for(x : $X in e){
    listitem{text(x.name) " " actionLink("[X]", $removeX(x))}
  } }
  select(x : $X, $addX(x))
  action $removeX(x : $X) { e.remove(x); }
  action $addX(x : $X) { e.add(x); } ]|
  where |[ List<$X> ]| := <type-of> e
    ; x      := <decapitalize-string; newname> $X
    ; $removeX := <concat-strings; newname>["remove", $X]
    ; $addX    := <concat-strings; newname>["add", $X]

DeriveInputText :
  |[ input(e){} ]| -> |[ inputText(e){} ]|
  where SimpleSort("Text") := <type-of> e

DeriveInputSecret :
  |[ input(e){} ]| -> |[ inputSecret(e){} ]|
  where SimpleSort("Secret") := <type-of> e

```

Fig. 21. Desugaring rules for input

association editor<sup>4</sup>. Similar rules can be defined for other types, as illustrated in Figure 21. As a consequence, the `input(e)` call is now sufficient for producing the appropriate input interface.

### 9.3 Edit Page

The presentation language supports the flexible definition of custom user interfaces. Based on this language the generation of the standard view/edit interface can now be reformulated as a model-to-model transformation. Rather than directly generating Java and JSF code, a presentation model can be generated from an entity declaration. The generator for the core language then generates the implementation. We consider edit pages such as in Figure 22, which consist of an input box for each property of an entity, organized in a table, and **Save** and **Cancel** buttons. The pattern for the (body of) an edit page is:

```

form {
  table {
    row{ "Blog"      input(entry.blog) }
    row{ "Title"    input(entry.title) }
    row{ "Created"  input(entry.created) }
    row{ "Category" input(entry.category) }
    row{ "Intro"    input(entry.intro) }
  }
}

```

<sup>4</sup> At the time of producing the final version of this paper, the editing of collection associations has been replaced with a different implementation.

```

    row{ "Body"      input(entry.body) }
  }
  action("Save", save()) action("Cancel", cancel())
  action cancel() { cancel blogEntry(entry); }
  action save() { entry.save(); return blogEntry(entry); }
}

```

Generation of pages of this form is now defined by the `entity-to-edit-form` rule in Figure 23. Note that `$x` is used both as the argument of the edit page

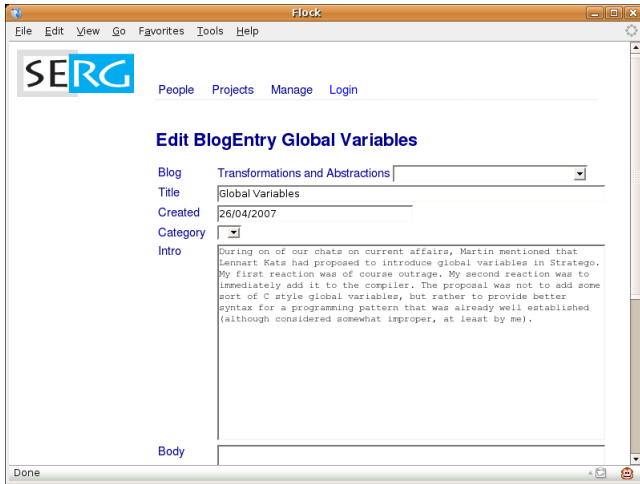


Fig. 22. Edit BlogEntry

```

entity-to-edit-form :
  |[ entity $X { prop* } ]| ->
  |[ define page $editX($x : $X) {
    form {
      table { elem* }
      action("Save", save())
      action("Cancel", cancel())
    }
    action cancel() { return $x($x); }
    action save() { $x.save(); return $x($x); }
  } ]|
  where $x      := <decapitalize-string> $X
        ; $editX := <concat-strings>["edit", $X]
        ; elem* := <map(property-to-edit-row(|$x))> prop*

property-to-edit-row(|x) :
  |[ y k s (anno*) ]| -> |[ row { str input(x.y) } ]|
  where str := <capitalize-string> y

```

Fig. 23. Derivation of edit page from entity declaration

and the name of the view page. For each property a table row with an `input` element is generated using the `property-to-edit-row` rule. Application of the previously defined desugaring rules for `input` then take care of implementing the interaction pattern corresponding to the type of the property.

## 10 Discussion: Web Engineering

The development of WebDSL in this paper touches on the development of domain-specific languages and on abstractions for web engineering. WebDSL was intended in the first place as a case study in the development of domain-specific languages. By now it has turned into a practically useful language. Since the first version of WebDSL, which is described in this paper, the language has been improved to increase coverage and has been extended with higher-level abstractions. List comprehensions support easy filtering and ordering of lists. Entity and global function definitions are useful for separating logic from presentation. Entity inheritance and extension support separation of concerns in data modeling. Recursive templates support the presentation of nested structures. Declarative access control rules regulate the access to pages and actions [52]. Furthermore, some of the implementation patterns have been replaced by others, without affecting the design of the language.

This section gives an assessment of WebDSL as a solution in the domain of web engineering. The criteria for the success of a DSL from the introduction are reiterated and the WebDSL project is evaluated with respect to these criteria. WebDSL is compared to alternative web engineering approaches, giving rise to ideas for further improvements and extensions. The next section considers other approaches and techniques for DSL engineering with respect to the criteria. Section 12 considers several challenges for language engineering.

### 10.1 DSL Engineering Evaluation Criteria

For the *process* of developing a domain-specific language we consider the following criteria:

- Productivity: What is the expected time to develop a new language? Distinguish the costs of domain analysis, language design, and language implementation.
- Difficulty: How *difficult* is it to develop a language? Can it be done by an average programmer or does it require special training? Does it require special infrastructure?
- How *systematic* and *predictable* is the process?
- Maintainable: How well does the process support language *evolution*? How difficult is it to change the language? Can languages be easily *extended* with new abstractions?

For the domain-specific language produced by a language engineering project we consider the following criteria:

- Expressivity: Do the language abstractions support *concise* expression of applications? What is the effect on the *productivity* of developing applications using the DSL compared to the traditional programming approach?
- Coverage: Are the abstractions of the language adequate for developing applications in the domain? Is it possible to express every application in the domain?
- Completeness: Does the language implementation create a complete implementation of the application or is it necessary to write additional code?
- Portability: Can the abstractions be implemented on a different platform? Does the language *encapsulate* implementation knowledge? To what extent do the abstractions leak implementation details of the target platform?
- Code quality: Is the generated code correct and efficient?
- Maintainability: How well does the language support *evolution*? What is the impact of changing a model? What is the impact of changes to the language?

In the following we evaluate the WebDSL design and development with respect to these criteria.

## 10.2 Evaluation of the WebDSL Development Process

The version of WebDSL described in this paper emerged from a project conducted by the author (non full-time) between September 2006 to June 2007. Several master’s students conducted related research activities that provided input for the project. In particular, Sander Mak developed a concurrent DSL for web applications [71] from which the idea of page definitions and navigations analogous to function definitions and calls originated.

**Productivity and Difficulty.** The effort of a language engineering project is divided into domain analysis, language design, and language implementation. In the WebDSL project, by far the most effort was spent in the first stage, i.e. becoming adequately knowledgeable in (one configuration of) the Java web programming platform. To give an indication of the effort involved, here is a brief description of the time line of the project.

In September 2006 a simple wiki application was built with MySQL, JSP, JDBC, and Java Servlets. The application included a wiki markup parser and HTML renderer. In February and March 2007 the wiki application was rewritten using Hibernate as object-relational mapping solution, greatly simplifying the implementation and improving the code quality. The reimplementations consisted of several iterations and introduced some complex features such as nested wiki pages and uploading legacy wiki content from XML data. At the end of March 2007, refactoring the code of the wiki application to try out new architectural ideas became too painful, and a start was made with building WebDSL. In April 2007, JSF, Seam, and Hibernate with annotations (instead of XML configuration) were ‘discovered’ and used as target platform in the emerging generator. Generation of a basic CRUD application (Section 4) and refinement of the data model DSL (Section 5) were realized by mid April. With this basic

generator in place it was now possible to experiment with much larger data models than the one for the wiki application. The running example was changed to the ‘research group’ application with publications, home pages, project, blogs, etc. that features in this paper. The presentation language and desugaring transformations for higher-level abstractions (Section 6) were developed in May 2007. The embedding of HQL queries, the module system, and numerous refinements and improvements were realized in June 2007.

*Language design* can be further divided into discovering the conceptual abstractions and formalizing these abstractions by means of a syntax definition. Again, most of the effort was spent in abstraction discovery; syntax definition with SDF is straightforward once the desired notation has been designed. The data model notation is not particularly original; it is basically a variation on record declarations in C or Pascal. The presentation layer language took a while to emerge. Although with hindsight it is a fairly obvious abstraction from JSF templates. In general, WebDSL liberally borrows designs from existing languages, which is a good idea since these designs will be familiar to developers.

*Language implementation* was heavily interleaved with design. The author has ample experience in language design and implementation, and is, as primary designer, intimately familiar with the Stratego/XT implementation technology. Thus, implementation of the generator required mainly the ‘encoding’ of the implementation patterns as rewrite rules and strategies using standard Stratego/XT practices. Getting to this level of language implementation productivity requires training in language design and a particular implementation technology such as Stratego/XT. A few innovations of Stratego/XT were made during the development of WebDSL. In particular, some utilities for the generation of multiple output files were developed. Furthermore, in a refactoring of the WebDSL generator several measures were taken to increase the *locality* of generation rules [54]. In particular, an extension of Java has been developed to support identifier composition, partial classes, partial methods, and interface derivation.

**Systematic.** The inductive, technology driven approach to DSL design adopted in the WebDSL project ensures a natural scope. The domain is defined by whatever is being programmed in practice. Abstractions are discovered by studying programming patterns; common codes ends up as constant code in templates, variable parts are inserted based on information in the model. This approach initially just leads to straightforward abstractions from existing programming practice. However, identification of these abstractions leads to better insight in the domain, which may give rise to reformulations not directly inspired by programming patterns. For example, the access control extension of WebDSL [52] is not based on the facilities for access control provided by the Seam framework. Rather an expressive and declarative mechanism is developed enabled by the possibility to perform desugaring transformations on the DSL itself.

Language design requires some creativity and cannot be very predictable. At first, abstractions can be formulated as enumeration of configuration data, possibly in some XML schema. However, good DSLs require a readable concrete

syntax. Language design can be inspired by existing language design patterns. For example, the design of the user interface language of WebDSL was inspired took some inspiration from L<sup>A</sup>T<sub>E</sub>X, not so much in its concrete syntax, as in concepts of separation of structure and style. A catalog of reusable language design patterns could be helpful in the design of new DSLs.

The implementation of WebDSL follows standard architectural patterns for DSL generators.

**Maintainable.** The extensibility of Stratego strategy definitions makes a generator naturally extensible to support new constructs of the same nature as existing ones. However, the extension of WebDSL with access control and the addition of new user interface components, eventually required a number of refactorings to maintain the modularity of the generator [54].

### 10.3 Evaluation of the WebDSL Language

**Expressivity.** Programming web applications in WebDSL is a breeze compared with programming in the underlying Seam architecture. Implementations are small and the data model and presentation are easily adapted when insights in the design of an application change. To objectively measure the decrease in effort (say lines of code) that is obtained by using WebDSL it is necessary to simultaneously develop the same web application in WebDSL and using some other techniques. Alternatively, we can exactly rebuild existing web applications and compare the two implementations. As an approximation we can take metrics from WebDSL projects as an indication.

For the website of `webdsl.org` we are developing a software project management application using WebDSL. The current prototype counts 2800 lines of WebDSL code and provides blog, forum, wiki, and issue tracker sub-applications. Access to the applications is controlled by a declarative access control policy (see below). The various applications support cross-linking from user-provided content via wiki-like links, which can address pages symbolically, for example `[[issue(WEBDSL-10)]]` creates a link from a blog entry to an issue in the issue tracker. The generated implementation of this application takes about 44K lines of Java code (3.6K for entity classes, the rest for beans) and some 25K lines of XHTML. Of course, this code is not necessarily as compact as it would be programmed manually. But a factor of 5 to 10 decrease in size compared to manually programmed applications appears to be a realistic (conservative) estimate.

The order of magnitude decrease in code size implies a significant increase in productivity. In particular, refactoring the design of an application can be realized much faster than is the case in the target platform, simply because less code is involved. However, the reduction of accidental complexity reduces application development to the hard part of development, i.e., requirements analysis and application design. Once it is known what the structure and functionality of an application should be, it is easy to realize that. However, WebDSL does not (yet) provide much help for coming up with a design. Further abstractions, such as for workflow, can help guide the design of certain ‘genres’ of applications.



While macro productivity is increased, micro productivity is not ideal. The time it takes to generate code, compile it, and deploy it in a JBoss application server determine the development feedback cycle. This cycle entails a penalty that is felt most when making small changes. A better model for incremental compilation and deployment should improve this factor.

**Coverage.** The WebDSL language supports the creation of a wide range of web applications with a rich data model. There are numerous ways in which the coverage of WebDSL can be extended and refined. In the rest of this section several ideas are discussed.

**Completeness.** The WebDSL generator generates complete code. There is no need to fill in or manually tune generated code skeletons. Sometimes it is necessary to add new built-in types. For instance, to represent patches for version management of the wiki application of `webdsl.org`, a patch library implemented in Java was added to the collection of libraries comprising the run-time system. Such built-in types are implemented as a separate module with rules plugging into the type checker and code generator. This extensibility should be made less intrusive by supporting the declaration of new types and operations in the language itself.

**Portability.** The portability of WebDSL to other Java web frameworks, or other implementation platforms such as PHP or C# has not yet been realized, so no hard claims about the quality of the WebDSL abstractions can be made. However, there is some evidence that the abstractions are fairly robust and target platform independent. Several of the programming patterns that gave rise to the WebDSL abstractions have been replaced by others, without changing the language constructs that they gave rise to. In Section 8 the template mechanism is implemented through expansion. This precludes the use of recursive template invocations, which would be useful for the presentation of hierarchical, nested structure such as a document with sections and subsections. Recently, we figured out how to translate separate template definitions. This required a change in the back-end of the generator, but the language itself already supported the expression of recursive template invocations.

**Code Quality.** WebDSL applications inherit properties such as performance, robustness, and safety from the target architecture. The technology driven approach underlying the design of WebDSL starts from the assumption that the target architecture is solid. However, Seam itself is new and under development. No experiments have been performed yet to establish these properties in a production setting.

**Evolution.** Complete code generation ensures that *regular* evolution of an application is a matter of reapplying the generator to obtain an implementation for a new version. Otherwise, the evolution of web applications and the version of WebDSL they are constructed with has been ignored in this paper. It is however, an important consideration in a software development process based on DSLs. Section 12 outlines (research) challenges for evolution of DSL-based software development.

## 10.4 Static Verification

WebDSL statically checks application definitions. Expressions accessing, manipulating, and creating data are checked for consistency with the declared entities and the variable declarations in scope. The existence of pages in navigations is checked, the types of actual parameters to page navigations are checked against the formal parameters of page definitions. Embedded HQL queries can also be checked against the declared entities; implementation of this feature is not yet complete. The remaining errors are logical errors in actions (e.g. accessing a property with null value), and errors in the composition of web pages. In practice, most errors that occur during development are application design errors. That is, realization during testing that pages and interactions should be organized differently. Due to code generation, the generated code correctly implements the specification. Errors normally made in boilerplate code are avoided. Any remaining errors are bugs in the generation templates, which only need to be repaired once.

Logical errors cannot be completely eliminated. Well-formedness of generated web pages could be checked statically by extending the type checker to check for valid combinations. The only error of this kind encountered in practice, is forgetting to embed form elements in a `form{...}`. The other template elements can be combined fairly liberally due to the leniency of browsers. However, checking such properties would ensure better HTML documents. This is done in systems such as `<bigwig>` [14], `JWIG` [25], `WASH` [92] and `Ocsigen` [9]. In particular, the `<bigwig>` and `JWIG` systems provide sophisticated correctness checks of document well-formedness. Templates in these systems are used to dynamically create documents, including the use of recursive definitions. Data-flow analysis is used to verify that all possible documents that can be generated by a program are valid.

## 10.5 Input Validation and Data Integrity

Properties and entities may need to satisfy more strict constraints than can be expressed using types alone. First, in some cases it is required to restrict the form of value types. For example, the syntax of an email address should be checked on submission and an error reported if not conforming. Next, constraints on combinations of objects should be checked. For example, in a conference system, the author of a paper may not be a reviewer of that same paper. Violations to this constraint should be detected when changes are made. Both types of constraints can be expressed declaratively, using regular expressions for input validation and Boolean expressions over object graphs for structural invariants. The `PowerForms` tool of the `<bigwig>` project provides a declarative language for declaring the client-side validation of form fields using regular expressions and interdependencies between form fields [13]. We plan to include support for the specification of data integrity constraints in a future version of WebDSL.

## 10.6 Access Control

A related concern is controlling the access to data and the pages that present and modify them. Access control checks can be expressed in WebDSL page

definitions by means of a conditional content construct (if condition holds, show this content). However, directly expressing access control with that mechanism would result in a tangling of concerns. We have designed an extension of WebDSL with declarative rules for user authentication and access control that supports separate specification [52].

## 10.7 Presentation

Presentations in WebDSL depend on the basic page elements defined by the generator. The elements supported currently cover the basics of HTML, abstracting from visual layout by relying on cascading stylesheets (CSS). Fancier elements can be added by extending the generator with new mappings from page elements to JSF components. It should be possible to provide such extensions as a plug-in to the generator, which requires an extensibility mechanism. Using the extensibility of strategy definitions in Stratego and an extension of Java to support partial classes, such extensibility is realized in a refactoring of the WebDSL generator [54]. A concern in the design of such extensions should be a proper separation between declaration of the structure of page content and visual formatting. Many JSF components are variations on the same theme, e.g. a list, vs a table, vs a grid, which are different visualizations of the same information.

The current design of WebDSL is page-centric, with actions and navigations leading to requests of complete new pages. The trend in web application design is towards inclusion of elements from rich (desktop) user interfaces, in which only parts of the page get updated as a reaction to user actions. An experiment with targetting the Echo2 Ajax framework [2] has shown that it might be feasible to develop rich user interfaces with the WebDSL abstractions. The central idea of the experiment was to use templates as the components to be replaced as a response to user actions. A less ambitious approximation of richer user interfaces can be obtained by targetting Ajax JSF components, which is already done to some extent.

## 10.8 Control-Flow

WebDSL provides a high-level language for implementing web applications by abstracting away from low-level details. However, in its core the language has the same page-centric model as the underlying Seam architecture. It could even be observed that WebDSL makes this architecture more explicit; where in Seam a page is defined by means of a number of separate artifacts, WebDSL unifies the elements of a page in a single definition. This architecture implies that user interactions take the shape of a series of requests and responses.

The Mawl [4] form processing language introduced a paradigm for modeling web interactions in the form of traditional console interaction. That is, web pages are considered as the input and output actions of a sequential program that control the interaction. The following Mawl example defines a session in

which first the user should provide a name (`GetName`), which is echoed in the next step (`ShowInfo`) [4]:

```
global int access_cnt = 0;
session Greet {
  local form {} -> { string id } GetName;
  local form { string id, int cnt } -> {} ShowInfo;
  local string i = GetName.put({}).id;
  ShowInfo.put({i, ++access_cnt});
}
```

Here `GetName` and `ShowInfo` are the names of separately defined HTML templates with parameters filled by the `put` operation. The statelessness of the http protocol requires the server to remember where to resume the program after the user submits a request.

In an application of Scheme to web applications, Queinnee [80] observed that capturing of the interaction state can be implemented elegantly by means of continuations, in particular the `call/cc` feature of Scheme. This approach has subsequently been adopted and refined in the PLT Scheme web server [67]. The Seaside Smalltalk web programming environment uses callbacks with closures to model control flow [40]. The OCaml web framework Ocsigen uses continuation passing style and stores continuations server-side on disk between requests [9]. The WASH [92] framework uses a monad to capture the continuation of a response. While continuations appear to be a very elegant formalization of sequential series of interactions with a single user, it is not clear that continuations can also be used to capture interactions involving (many) different users over multiple sessions as is needed for implementing workflows.

The Seam [56, 74] framework, which WebDSL targets, supports a notion of *conversations* to deal with the problem of keeping state in different threads of the same session separate. The solution here is basically to encode the continuation in a combination of data and context, i.e., the page being visited. In WebDSL it has not appeared necessary yet to build on this mechanism. First of all, the typical interaction that consists of presenting a form and receiving its inputs can be realized with a single page definition (based on the JSF facilities for forms). Next, WebDSL has session entities for storing data relevant for all interactions in a session (a feature not discussed in this paper). We have chosen to model state in sequential interactions, as well as in more complex interaction scenarios such as workflows, using regular WebDSL entities. Figure 24 illustrates this by encoding the Mawl example discussed above (including the forms for presentation). The definition introduces a `Counter` entity to keep track of the number of visits using an application global variable. The `Visitor` entity is used to store the name of a visitor obtained in the `getname` page. The object is then passed as a parameter of the `greet` page, where it is used to obtain the name. The `go()` action of the `getname` page creates the `Visitor` object and *makes it persistent*. This is the difference with the Mawl approach, where the session data is transient and restricted to the session. The advantage is that interactions become naturally persistent such that users can come back to an interaction in later sessions.

```
entity Counter { accesses :: Int }

globals { var stats : Counter := Counter { accesses := 0 }; }

entity Visitor { name :: String }

define page getname() {
  form {
    var n : String;
    "Enter your name: " input(n)
    action("Go", go())
    action go() {
      var v : Visitor := Visitor{ name := n };
      stats.accesses := stats.accesses + 1;
      v.persist();
      return greet(v); } } }

define page greet(v : Visitor) {
  "Hello, " output(v.name)
  " you are visitor number " output(stats.accesses)
}
```

**Fig. 24.** Interaction sequence using pages in WebDSL

Scenarios in which multiple stakeholders in different roles need to interact are naturally modeled in this style as well. Using an appropriate access control policy, the visibility of the objects can be restricted. While this mechanism provides flexible expressivity for implementing all kinds of control flows, we will consider adding higher-level abstractions for defining complex workflows. For short-lived conversations (e.g. filling in a multi-page form) it would still be useful to have in-memory non-persistent (transient) state, for which the Seam conversations model may be the right implementation solution.

## 10.9 Testing

An important open issue is the testing of web application developed with WebDSL. We need two types of tests. First, regression testing for the language and generator, is needed to make sure that the implementations generated by the generator are correct. For this purpose we would need to make a set of small test applications, that exercise specific constructs of the language. Secondly, WebDSL application developers need to test that their program satisfies its specification. It should not be necessary to test basic, low-level functionality, since correctness of the language construct should ensure their functionality. Thus, application tests should test application behavior. For both kinds of tests we need a DSL for expressing high-level tests.

## 10.10 Model-View-Controller

WebDSL programs combine the user interface implementation with the logic associated with user interface events. This design violates the model-view-controller pattern, which dictates that the user interface (view) should be separated from the controller [48]. There are several reasons why such a separation is desirable.

First, to distribute functionality over different nodes in the network in order to distribute the load to more than one server. Typically, the application is separated into tiers, each of which is implemented as a process on a different server. This goal is not precluded by the WebDSL approach. Even while an application *definition* integrates UI and logic, in the *implementation* these are separated into JSF pages and session beans, which are designed for a layered architecture.

Secondly, motivation for applying the MVC pattern is to be able to use different views with the same logic and/or to let developers with different skills work on view and controller separately. This requires not so much that logic and view should be separated (as a policy), but rather requires mechanisms that allows them to be separated when that is necessary. The template mechanism of WebDSL allows views and actions, performed in those views, to be implemented separately, where the view calls an abstract template, defined by the controller, as illustrated in the following example:

```
define view(field1 : String, field2 : String) {
  form{ input(field1) input(field2) submit(field1, field2) }
}
define control(m : Model) {
  view(o.field1, o.field2)
  define submit(field1 : String, field2 : String) {
    action("Submit", submit())
    action submit(){ m.field1 := field1; m.field2 := field2; }
  }
}
```

Here the `view` template definition can be an elaborate structure definition, which only takes basic data types as input values. Invoking an action is delegated to an abstract `submit` template. The `control` uses the `view` to display the data, and defines a concrete `submit` to implement the action.

## 11 Discussion: Language Engineering Paradigms

An application domain is a collection of concepts. The description of an application in a domain is a collection of statements involving those concepts using the ‘language of the domain’. For example, ‘make a page that displays the properties of this object’ is a sentence in the domain of web applications. A conceptual domain language can be implemented in many different forms, even as a library in a ‘conventional’ general-purpose programming language. *Language engineering* is concerned with the design and implementation of languages in all their different forms. This section provides a brief survey of existing language engineering

paradigms and their impact on the language development process. A complete and in-depth survey of language engineering is out of the scope of this paper. There are many surveys on domain-specific languages and their development from different perspectives, including [32, 33, 60, 73, 85, 86, 87, 99].

**Approaches.** The discussion is organized by considering the distance of the approach to the implementation platform. *Application frameworks* are based on the concept of ‘a library as a language’ (Section 11.1). *Domain-specific embedded languages* encode a language using the syntactic facilities of the host language (Section 11.2). *Interpreted DSLs* are separate languages, which are passed to an interpreter library (Section 11.3). *Domain-specific language extensions* add new syntax to a general purpose language (Section 11.4). *Compiled domain-specific languages* are defined completely separately from the implementation platform, and can in principle be translated to more than one platform (Section 11.5).

**Technologies.** These approaches entail fundamentally different architectures for capturing domain-specific knowledge with implications for development and usage. Somewhat orthogonal to these basic approaches are specific *technologies* for realizing them. Technological frameworks are typically designed for use with a particular approach, but their use may be stretched to other approaches as well. Section 11.6 outlines the main ingredients of language implementations, and gives an overview of some typical tool sets.

**Criteria.** In the previous section we applied the set of evaluation criteria to WebDSL. In this section we use these criteria to compare the properties of different approaches. Of course, it is not possible to make generic statements about all products of a particular approach. For example, the quality of generated code is not magically guaranteed by using a particular generator technology, but will depend the efforts of the generator developer performing meticulous research into the properties of the target platform. However, certain approaches may facilitate better results in some area than others.

## 11.1 Application Frameworks

The most accessible approach to encapsulating domain knowledge is by means of a library or (object-oriented) framework. The language defined by a library is the *application programmer’s interface (API)*. That is, a library provides data structures (objects) with operations (methods). The basic elements of the language are the calls to operations. The composition mechanism is generic, that is, not specific for the domain. For example, an object-oriented programming language provides object creation, method calls, subclassing, and inversion of control [46].

**Developing Frameworks.** An application framework is directly implemented in a third-generation general-purpose programming language such as Java. Thus, framework development can directly use all the productivity advantages provided by modern programming languages and their interactive development environments. While frameworks are developed in a basic programming language,

designing a good framework is not easy and requires well trained software developers. However, there is a rich literature with design patterns [48] for developing object-oriented software and frameworks. Maintenance of frameworks is tricky when many client applications exist. Changing the interface breaks the build of client code, but only changing the implementation may not be safe either, since client code may depend on implementation details of the framework.

**Developing with Frameworks.** The primary advantage of a framework compared to other approaches discussed later, is that they integrate well with other programming tasks. However, the implementation technology does not support domain-specific verification. Only constraints that can be encoded in the host type system can be checked at compile time. Frameworks are expected to cover a complete (technical) domain, which tends to make them large and complex. The expressivity of a framework is low, as the notation based on the generic composition mechanisms of the host language are typically not tuned to the application domain. Modern frameworks such as Hibernate [10] and Seam [56, 74] are fairly high-level due to the use of *annotations* and dependency injection, which are targeted by run-time or deployment-time compilation and instrumentation. Software developed with a particular framework is not portable to a different platform. The framework ties client code to its host language. To support all possible functionality, frameworks may provide multiple layers of abstractions which are not removed by the compiler. The internal structure is not completely encapsulated and are for example manifest in stack traces produced by exceptions. Frameworks use mechanisms such as inheritance and annotation processing to allow client code to specialize the generic functionality it provides. This form of extensibility is built into the infrastructure.

## 11.2 Domain-Specific Embedded Languages

While one could view the API provided by a framework as a language, this is not typically the perspective of programmers. The idea behind *domain-specific embedded languages (DSELS)* is to build DSLs in the form of libraries in a general-purpose language. Hudak argues that combinator libraries in higher-order functional languages such as Haskell are especially suited for building domain-specific languages [55]. In essence, DSELS are the same as frameworks, but the differences in abstraction mechanisms between object-oriented and functional languages, give them a different flavor.

**Developing Combinator Libraries.** The core advantage of DSELS is the reuse of abstraction mechanisms in the host language. It is not necessary to design and implement a mechanism for functional or modular abstraction. Also control-flow constructs are easily defined in a *lazy* functional language such as Haskell. Infix operators get a long way to approach domain-specific notation. Thus, the developer can concentrate on the truly domain-specific aspects of the language. Furthermore, there is no need to write code generators; language ‘constructs’ are combinators, which are defined by means of function definition.



**Developing with Combinator Libraries.** DSELs share with frameworks the good integration with the host language, the lack of portability, and the lack of domain-specific verification, syntax, optimization, and error messages. However, domain-specific type checking can be achieved to some extent using phantom types [70].

### 11.3 Interpreted Domain-Specific Languages

Interpreted DSLs are proper languages with their own syntax and semantics separately defined from a host language by means of an interpreter, which executes sentences.

**Developing Interpreters.** Developing an interpreted language requires development of a syntax (with corresponding parser) and the interpreter itself. The problem of building an interpreter can be mitigated by organizing an interpreter as a factory that creates instantiations of a class hierarchy. After initialization of the objects, it functions as a ‘normal’ program. Thus, an existing framework can be given domain-specific notation through an interpreter.

**Developing with Interpreters.** When the interpreter is built into a library, it can be invoked from a general-purpose program and may fit in a software development approach otherwise based on a general-purpose language. For example, SQL and XSLT can be used in this fashion. Models can be executed on any platform with an interpreter, which entails that the interpreter is needed at run-time. It is typically not easy to support interaction between interpreted code and code in a GPL. However, a combination of the factory approach mentioned above and reflection may support some form of interaction, e.g. a foreign function interface that supports calling (host) library functions from the DSL program. Usually, interpretation incurs overhead compared to compiled code, since the interpreter must parse and inspect the (abstract) representation of the model. Extension of the language may not be easy, as it requires extension of the interpreter.

### 11.4 Domain-Specific Language Extension

The idea of *domain-specific language extension* is to extend a general-purpose *host* language with domain-specific *guest* notation. In contrast to domain-specific embedded languages, the syntax of the host language is actually extended to truly accommodate the domain-specific notation. An *assimilation* transformation maps extension back to base language [21]. This can be implemented as a pre-processor of the base language or by a proper extension of the host language compiler. Dmitriev advocates this approach with the name *language oriented programming* [36].

**Developing Language Extensions.** Developing a good language extension implementation is difficult, since it requires extension or reimplementing of a considerable part of the host language infrastructure. First, a complete syntax

definition of the host language is needed, which can be extended with the domain-specific notation. This requires some form of syntactic extensibility. Second, the extension needs to be implemented either by extending the host compiler or by means of a translation down to the base language. (This is basically similar to DSL compilation, discussed below.) Third, the type checker of the host language needs to be extended. There are a number of approaches for realizing this scenario.

*Extensible languages* are languages that are prepared (to some extent) for extension with new syntactic constructs. The prototypical example of an extensible language is Scheme, which provides macros for introducing new ‘syntactic forms’ [26]. Macro definitions define a translation from the new language construct to more basic language constructs. Macros are applied by the interpreter. Thus, programs can introduce and use extensions. Other incarnations of this approach are Template Haskell [83], which supports compile-time generation of program fragments (but no syntactic extensions), and Converge [95], which provides compile-time meta-programming support for the definition of new embedded languages and their assimilation. Language workbenches [47] are IDEs supporting the creation of macro-like language extensions.

*Pre-processing* is another popular approach to realize language extension. The advantage over extensible languages is that a pre-processor can be built for any base language, also those not designed with macro-like facilities. An example of a pre-processor based language extension approach is MetaBorg [21], which relies on the modularity of SDF to create the syntactic extension of a language and on Stratego for expressing assimilation rules. MetaBorg extends the framework approach to DSL implementation with proper syntax, thus providing a domain-specific notation for the abstract syntax defined by an API. A particular instance of MetaBorg is StringBorg [16], a technique for providing proper syntax checking for interpreted DSLs such as SQL. Instead of encoding queries in string literals, which makes applications vulnerable to injection attacks, queries are defined in an embedded DSL, which is syntactically checked. Under the hood a string representation of the query is eventually constructed, but without the risks of malicious injections. The disadvantage of pre-processors is that they do usually not provide proper integration with the semantic checking of the host language, since that requires re-implementation of those parts of the compiler in the pre-processor.

*Extensible compilers* avoid the incompleteness of pre-processors by exposing the internal structure of the compiler to extensions. Thus, the implementation of an extension can extend the type checker to guarantee that only statically correct programs are compiled, and that error messages are phrased in terms of the source program, not the assimilated one. Examples of extensible compilers for Java are Polyglot [75], Silver [109] and JastAddJ [45]. The latter two are based on extensible attribute grammars formalisms, which supports declarative and compositional specification of the type system of a language [44, 109].

The disadvantage of an extensible compiler is that an extension is based on white box reuse of the base compiler, rather than a semantic description of the language. This requires intimate knowledge of the implementation of the

compiler and exposes extensions to changes in the implementation. The approach of *compilation by normalization* [58] avoids this problem by providing a mixed source and byte code language as target for a pre-processor. By means of tracing information, type and run-time errors can be reported in terms of the original source code. By exposing the target language as part of the source language, pre-processors can produce low-level implementations where needed without invasive extension of a compiler.

While extending compilers to support extended languages is understood to some extent, modern languages require rich interactive development environments. Exploration of the design and implementation of such IDEs for embedded languages is only recently started [59].

**Developing with Language Extensions.** Provided that also the IDE is extended, a general purpose language with domain-specific extensions can provide a very expressive programming environment that allows to use a DSL where needed, and the general-purpose language for ‘normal’ programming. As is the case with frameworks and combinator libraries, models in an embedded languages are tied to their host language and cannot be used with a different platform. It is important that assimilations do not leak, that is, expose the developer to the result of translating embedded models to host code, for example in the form of error messages at compilation or run-time.

## 11.5 Compiled Domain-Specific Languages

WebDSL falls in the category of *compiled domain-specific languages*, that is, a language dedicated to a particular application domain, not embedded in a particular host language or implementation platform. Models in such languages are implemented by *compilation* or *code generation*, i.e. translation to a program in some target language.

The main disadvantage of the approach is that implementation of a DSL compiler can be a significant undertaking. Unlike DSELs, there is no linguistic reuse of abstraction facilities of a host language, implying that all the basic constructs that a language requires, need to be implemented in addition to the actual domain-specific elements. For example, WebDSL has an action language, which is a subset of imperative language with object-oriented elements.

The main advantage is that the language can be designed to be independent of the target platform, and that models in the language can thus be implemented on more than one platform. To achieve portability one should guard against leakage of implementation details from the target platform. While abstractions cannot be borrowed from a host language, the gain is that there are no constraints imposed on the design of abstractions. Furthermore, the compiler can provide domain-specific error checking and optimization.

There are many variant approaches including generative programming [32, 33] and model-driven engineering [82] and technologies for realizing them. However, the essential architecture is the same in all approaches. In addition, to proper DSL compilers there are less complete variations, *scaffolding* and *light weight languages*.

**Scaffolding.** The term ‘code generation’ is understood in some contexts as the generation of incomplete code skeletons from configuration data, e.g. a UML model. For example, from a class diagram a set of Java classes is generated with the attributes and operations as specified in the diagram, but the implementation of the methods needs to be filled in. Another example is Ruby on Rails [93], a framework for web application implementation based on the Ruby programming language, which generates boilerplate code from a database schema.

The advantage of a scaffolding generator is that it is relatively easy to build. There is no need not design and implement abstractions for areas where the developer is expected to do heavy customization. The big disadvantage is that it requires maintenance at the level of the generated code. This requires round-trip engineering or carefully marking in the generated code which parts were generated and which parts customized, such that only generated parts can be re-generated. However, this will remain fragile and prone to inconsistencies between model and code. Often, re-generation is not supported as it carries a substantially higher implementation cost than the scaffolding generator itself. More importantly, the approach exposes the developer to the implementation, which breaks encapsulation of the generator and limits its scalability.

**Lightweight Languages.** Another category of DSL implementations is that of *lightweight languages* [86]. These are languages with a very restricted scope, possibly used in a single software project. Such languages are economically viable because they are implemented cheaply, for instance using regular expressions in Perl. The translation consists of simple local translations and does not include static error checking, placing the burden of creating a correct model on the programmer. This approach does not scale to languages that need to be used in many projects and/or by many developers.

**Heavyweight Languages.** A proper domain-specific language is constructed according to well established architectural patterns for compilers [3]. A generator consists of a front-end that parses the model from a concrete syntax representation (be it a visual or textual) to an abstract representation. This representation is subsequently checked against the static semantic constraints. After optionally applying a number of transformations to the model itself, it is translated to code in some target language. There is a long tradition of tool kits with DSLs for reducing the effort of building compilers, e.g. [5, 49, 57, 62, 63, 81]. Stratego/XT fits in this tradition and so do the various MDE tool sets introduced recently. Within these architectural boundaries there are different styles for implementing the various aspects of a generator.

## 11.6 Language Engineering Tools

For the development of a framework or combinator library only an appropriate host language is required. For the other approaches discussed above, i.e. interpreted DSLs, language extensions, and compiled DSLs, tool infrastructure for language engineering is required. A language implementation requires parsing,

analysis, transformation, generation, and/or interpretation as discussed in Section 3.4. As with any domain, these tasks can be expressed in general purpose programming languages. However, by its nature this domain is a fertile breeding ground for tools and domain-specific languages. The rest of this section gives a brief summary of the main variation points and illustrates how some existing tool sets bind these variation points.

**Parsing.** The definition of a textual DSL requires a parser that turns the text of a model into a structured representation, which can be used for further processing. Most parser generators are based on deterministic subsets of the set of all context-free grammars, such as LL (recursive descent) implemented by ANTLR [77] or LR [64] as implemented by YACC [57]. While these subsets guarantee unambiguous syntax definitions and (near) linear time parsing, the restrictions can require awkward encodings of linguistic constructs. Generalized parsing algorithms such as Earley [41], GLR [94], or SGLR [101] do not suffer these limitations. However, the support for error messages and error recovery is typically not as good as with deterministic parsers.

**Model Representation.** The abstract representation of a model is the data structure that analysis, transformation, and generation operate on. The properties of a representation determine how costly (in terms of time and space) it is to perform certain operations. Unfortunately there is no single representation that makes all operations equally cheap [104].

With a functional representation such as the Annotated Terms (ATerms) used in Stratego [96], or the algebraic data types in (pure) functional languages such as Haskell [79], performing transformations is cheap since copying of sub-trees constitutes of copying references, instead of cloning. Also, a functional representation is *persistent* in the sense that a transformation does not destroy the old representation. However, the directed acyclic graph (DAG) structure does not admit extending the tree with references to other parts of the tree. Hence, context information needs to be stored in symbol tables or similar data structures.

In contrast, graph structures (including object graphs in object-oriented languages) allow extension of nodes with arbitrary cross references in the graph, which can be used to make context information into local information. For example, add a reference from a variable to its declaration. This makes the result of analyses much easier to express. The downside is that transformations on graphs are not persistent, i.e. require a destructive update, or copying of the entire graph structure. Meta models in modeling frameworks such as EMF [23] define graph structures, and thus require graph transformation solutions. Of course, EMF can be used to model more restricted representations, including functional representations.

**Analysis and Transformation.** Analysis and transformations of models are used to prepare the model for code generation, for example by enriching it with type annotations (Section 7.1) or by desugaring high-level constructs as lower-level ones (Section 9). In principle, analyses and transformations can be

expressed in any functional, imperative, or logical programming language. However, specialized transformation languages may allow more declarative and/or more concise expression of transformations. As discussed above, the representation of models has consequences for the applicable transformation paradigms.

Term rewriting [6] is a useful paradigm for transformation of a functional representation. Rewrite rules are declarative specifications of one step transformations. Exhaustive application of rewrite rules is performed by an implied rewriting strategy. Rewriting is useful for repeated, cascading transformations such as desugaring, where model elements are rewritten to combinations of other model elements, which can subsequently again be rewritten. This approach requires an easy way to construct large patterns of model elements. Concrete object syntax [102] enables the natural construction of model fragments of hundreds of nodes, which is extremely tedious using abstract object construction techniques. In pure term rewriting, rewrite rules are applied exhaustively to the entire term. Because of non-confluence and non-termination more control over the application of rules may be necessary. Various approaches for controlling rules have been developed [104], among which the programmable rewriting strategies of Stratego.

Analysis typically requires non-local information, e.g. the declaration of a variable and its use. While rewriting approaches can express context-sensitive analyses and transformations, e.g. the type checker in Section 7.11, a more declarative approach to expressing analyses is provided by *attribute grammars* [65], which are supported by systems such as JastAdd [44] and Silver [109]. An attribute grammar assigns values to attributes of tree nodes. Attribute values are defined by means of attribute equations in terms of other attributes. The scheduling of attribute value computations is left to the attribute grammar compiler. The value of an attribute may depend on the entire tree. Applying just a single local transformation in principle invalidates all attribute values in the tree, and requires re-computing all attribute values. Therefore, attribute grammars are useful for performing analyses of static trees, while rewriting approaches are more suitable for performing transformations. It is a research challenge to find a combination of the formalisms such that analysis and transformation can be mixed.

There are numerous approaches to transformation of graph representations as occur in modeling approaches. Czarnecki and Helsen [34] give an extensive survey of features of model transformation approaches.

**Generation.** Many tool sets provide a *template engine* such as Velocity [89], StringTemplate [78], or Xpand [43] for translation of models to program text. A template is a quotation of a static piece of code. Variability in the code is realized by means of anti-quotation expressions that allow insertion of names, expressions, or sub-templates specialized for the input model. Templates are an improvement over the practice of printing string literals in a regular programming language, which require escaping of special characters and often do not support multi-line fragments. Textual templates do not check the syntax of the quoted code fragments. This makes the technique easily adaptable to any target language. However, it may result in syntactically incorrect code being generated. More importantly, the generator does not have access to the structure of the

generated code. This makes it impossible to apply transformations, e.g. instrumentation, to the generated code.

The approach used in this paper can be characterized as ‘code generation by model transformation’ [54]. The generator produces a model representation of the target program, which is amenable to further transformation. Producing large fragments of target models is often inconvenient using the abstract syntax notation. Concrete object syntax combines the surface syntax used in a template engine with the underlying model representation of the generated code. Implementation of concrete syntax requires a grammar formalism that supports the modular composition of the context-free and lexical syntax of languages [22, 102]. Eventually, the model representation needs to be rendered as text. This is a straightforward one-to-one rendering of each node also known as pretty-printing.

**Tool Sets.** A tool set for language engineering provides a particular combination of support choosing some point in the design space sketched above. In addition, this configuration is realized on a particular programming platform, which may be a specific operating system and usually a particular programming language. Thus, while in principle the architectures of the tool sets is comparable, in practice the choice for a particular tool set may be based on other factors than just the techniques supported. Furthermore, for branding purposes, tool producers, be it industrial, open source, or academic, tend to emphasize the differences between tools, rather than their commonalities. The following list of tool sets gives an impression of the variability in the domain, without pretending to be complete.

Rewriting languages

- ASF+SDF [97] is a compiled language based on first-order term rewriting with traversal functions, providing concrete syntax for patterns in rules.
- TXL [31] is an interpreted, rule-based functional language with concrete syntax patterns, and a form of deep application of rules.
- Stratego/XT [17] is a compiled transformation language based on rewriting with programmable rewriting strategies; rules can use abstract or concrete syntax.
- Strafanski [68] is a combinator library for strategic programming (in the sense of Stratego) in Haskell.

Attribute grammar formalisms

- Eli [50] is a composition of language processing tools including statically scheduled attribute grammars.
- JastAdd [44] is a compiled language based on rewriteable reference attributed grammars.
- Silver [109] is a compiled attribute grammar formalism with forwarding and dynamic scheduling of attribute evaluation.



## Modelware

- Open ArchitectureWare [43] is an Eclipse-based tool set for textual DSL definition and code generation. It uses EMF [23] for the representation of models. The xText grammar formalism, which is based on ANTLR, is used to define textual syntax of DSLs and the generation of an Eclipse editor plugin. The xTend ‘functional’ language is used for model transformation, and the xPanda textual template language is used for model to text transformation.
- MetaCase [60] supports the creation of visual domain-specific modeling languages.
- Visual Studio DSL Tools [30] is a meta-modeling framework for visual modeling languages. Code generation is achieved using a textual template engine.

## 12 Discussion: Language Engineering Challenges

A discussion of some challenges for research in language engineering.

**DSL Interaction.** WebDSL is a composition of several languages, that is, a data model language, a presentation language, a query language (HQL), and an expression and action language. The language is a good basis for further abstractions, such as ones for access control and workflow. Template definitions and modules support the creation of reusable components. While these different languages support different aspects of web applications, they are integrated into a composite language to ensure smooth interaction between the different aspects; as opposed to the heterogeneous architecture of web applications implemented in Java. Although inspired by similar features in other languages, the language was designed and implemented from scratch. It would be useful to have language design and implementation patterns to be reused when creating new languages, if possible supported by tools or reusable libraries of language components.

A particular issue that arises in domain-specific language engineering is the design of language interaction. Software development typically requires the interaction between several technical and application domains. How can programs in different languages refer to each other? Can modules be compiled, or even type checked separately? Warmer [107] has developed a collection of DSLs for web applications using the Microsoft DSL Tools. In that work the assumption is that separate models are compiled to separate target files. Interaction of models is achieved using a registry that records interface information (key, value pairs). This approach precludes weaving of code from different models. Mak [71] has explored the separation and interaction of languages in a variant of WebDSL. Basically, the separation was into a data model language and presentation language, which map to separate target code components.

**Development Environment.** Software developers, especially those developing in Java or C# are accustomed to sophisticated development environments (IDEs), which help the programmer by means of syntax highlighting, cross-referencing, access to documentation, and code completion. When developing



a new DSL, the barrier to being used can be lowered considerably, if such interactive support would be available as well. The challenge here is to generate from the definition of a language an IDE, for example by creating an Eclipse plug-in supporting syntax highlighting, syntax checking, typechecking, refactoring, code completion, and cross-referencing. Despite research projects such as the Synthesizer Generator [81] and the ASF+SDF MetaEnvironment [62], the creation of an IDE for a new language remains a laborious process. The Eclipse IDE Meta-tooling Platform [1] may reduce the effort to develop IDEs for new languages. A first step on the path to the integration of the language definition techniques used in this paper (Stratego and SDF), is the generation of Eclipse plug-ins based on the IMP framework from SDF definitions [59].

**Deployment.** A DSL generator only automates one step in the development process of a software system. While the generator encapsulates knowledge about developing applications in the domain, more knowledge is required for successfully deploying an application. Therefore, a good DSL should also hide irrelevant deployment details. Ideally, the DSL programming environment offers a virtual machine for operating DSL programs, which completely hides its run-time system. Thus, in the domain of web applications such a virtual machine would appear to run WebDSL applications directly, and behind the scenes generate the Java/XML implementation code, compile it, and activate the application server to run the application. The Nix software deployment system [37, 39] provides a suitable infrastructure for realizing this scenario. Using a functional language, deployment configurations from source builds to service activation can be described [38]. Using this approach a first experimental setup has been created for deploying WebDSL applications, which is being used to deploy the `webdsl.org` website.

**Extensibility.** A language should be designed for growth [88] in order to accommodate future requirements. Therefore, the implementation of a language should be easily extensible with new basic types, new constructs, new abstractions, and new sub-languages. Systems such as Silver [109], JastAdd [44], and Stratego/XT [103] (used in this paper), provide source level extensibility. That is, a language definition can be separated into modules and new features can be implemented by providing new modules. However, the new combination needs to be compiled from source *as a whole*. True extensibility would entail that *users* of the language can combine extensions provided by different producers for a particular application without recompiling the generator. This requires separate binary extensibility of language definitions and generators.

**Evolution.** The introduction of domain-specific languages can greatly improve the evolution of software by drastically reducing the amount of source code needed for systems. Paradoxically, reliance on DSLs also introduces a new software evolution problem. The number of languages in which software is written increases, requiring developers with knowledge of multiple languages [91]. Furthermore, while software applications may become easier to maintain, the implementations of the languages need to be maintained as well [98]. A problem that

is seen as one of the factors for the failure of fourth generation languages. The next paragraphs discuss a number of challenges for evolution of domain-specific languages.

*Data Migration.* Evolving applications based on DSLs should become easier. The size of an application is an order of magnitude smaller than before, which should make understanding and modifying programs easier. Complete code generation ensures that a complete system can be generated after modifying the DSL program. However, the data models that are implemented as the database schemas of deployed applications may have changed, requiring the database to be migrated to the new data model. To ease the evolution of applications, it is necessary to automate data migration as much as possible. At least there should be a language for specification of the migration between two data models at the level of the data model language (abstracting from implementation details of the database schema). Furthermore, the mapping between two data models could be inferred to some extent by considering the two versions.

*Model Migration.* The problem of data migration also plays a role on a level higher-up in the modeling hierarchy. Changing the definition of a DSL requires adapting existing DSL models. To increase the acceptability of DSL evolution, it is desirable to support language changes with automatic conversion tools. First of all, that requires the definition of a transformation from models in the old language to models in that new language such that the new models have the same semantics as the old models. Supporting such semantics preserving transformations, requires the new language to at least support the functionality of the old language, which imposes some constraints on evolution. As in the case of data migration it would be desirable if the migration of models can be derived from the evolution of the grammar. In practice, language designers take great care to design language changes to be backwards compatible. Better migration solutions will enable language designers to make more drastic (re)design decisions, which are sometimes needed when insight in the domain grows.

An important practical consideration in the migration of programs is the treatment of white-space and comments (layout). Developers do not appreciate the look of their programs to be drastically changed by automatic transformations. As a result, a semantics preserving transformation on the abstract syntax structure of a program is not sufficient. One solution direction is to support transformation with layout preservation. However, true layout preservation is not a solvable problem, since comments in programs do not have a formal relation to the surrounding code. Instead it would be a good idea to reduce the role of layout in languages. First, by making comments part of the syntactic structure, it can be treated like any other structures in transformations. Next, domain-specific languages should be designed to support self documenting code. After all, one of the ideas of DSLs is that they should express high-level application concerns, not implementation details. Finally, introducing enforceable coding standards (for layout) can eliminate the problem of re-formatting. (Note that these issues hold for visual (diagrammatic) languages as much as they do for textual languages.)

*Abstraction Evolution.* A particular variant of DSL evolution is the addition of new abstractions to the language. In that case it may be worthwhile to transform existing DSL models to use the new abstractions. This requires recognizing the use of the implementation patterns that the new abstraction mechanism abstracts from. Semi-automatic support for pattern recognition and subsequent transformation would be useful to support developers in migrating to the higher-level abstractions.

*Harvesting from Legacy Code.* Finally, after having developed a new DSL, it may be necessary to migrate existing legacy applications to the new DSL, which requires recognizing implementation patterns in legacy code. Even while a DSL design may be based on the abstraction of implementation patterns, these patterns may not be used exactly in an existing code base. As a concrete case, consider transforming legacy EJB applications to WebDSL programs, where JSF pages are translated to page definitions, entity classes to entity declarations, and session beans to page actions.

## 13 Conclusion

This paper has presented a case study in domain-specific language engineering. Based on this experience let's make an attempt at answering the questions 'when and how to develop a domain-specific language?'

**When to develop a DSL?** Starting to develop a DSL should only be done when there is a good understanding of the application domain and there exists a considerable code base for systems in the domain. That code base should exhibit clear signs of inadequate abstraction facilities in the form of boilerplate code in large quantities, even if best practices are being applied. Another sign is that mechanisms that have been introduced to raise the level of abstraction elude the verification facilities of the implementation language. Typical examples are XML configuration files, interpreter literal strings (SQL queries), and dependency injection annotations.

**How to develop a DSL?** Choose a high-level target technology; the DSL should not readdress problems that have already been solved by existing technology. Start with considering relatively large chunks of programs as candidate patterns. Study and understand the technology and recognize common patterns. Set up a basic generator early on. That makes it easy to experiment with alternative implementation strategies in the target architecture without having to write a lot of code. Do not overspecialize syntax. For example, a separate syntactic construct for each page element such as `section`, `header`, `list` in WebDSL, would lead to hard wiring in such constructs and a much larger implementation. Do not overgeneralize syntax either. Ending up with a completely generic syntax such as XML does not lead to readable programs. A core language that captures the essential operations of the domain is essential for achieving good coverage. But do not try to identify a core language from the start. The result may be

too close to the target target technology. For example, a modeling language that covers all EJB concepts provides 100% coverage, but is too low-level. Extend the core language with syntactic abstractions that allow concise expression. Include facilities to build a library, such as modules for organization of the code base and parametric abstraction over DSL fragments.

## Acknowledgments

In August 2006 Ralf Lämmel and Joost Visser invited me to give a tutorial at the GTTSE summer school to be held in July 2007. This invitation provided a perfect target and outlet for the rather uncertain sabbatical project that I had conceived to build a domain-specific language for web applications. Along the way I had many inspiring discussions about various aspects of this enterprise and received feedback on drafts of this paper. I would like to thank the following people for their input (in more or less chronological order of appearance): Martin Bravenboer, Jos Warmer, Sander Mak, William Cook, Anneke Kleppe, Jonathan Joubert, Rob Schellhorn, Danny Groenewegen, Zef Hemel, Paul Klint, Jan Heering, Ron Kersic, Nicolae Vintae, Charles Consel, and the GTTSE'07 reviewers. The research was supported by NWO/JACQUARD project 638.001.610, *MoDSE: Model-Driven Software Evolution*.

## References

1. Eclipse IDE Meta-tooling Platform (IMP), <http://www.eclipse.org/proposals/imp/>
2. Echo web framework (July 2007), <http://echo.nextapp.com/site/echo2>
3. Aho, A., Sethi, R., Ullman, J.: Compilers: Principles, techniques, and tools. Addison-Wesley, Reading (1986)
4. Atkins, D.L., Ball, T., Bruns, G., Cox, K.: Mawl: A domain-specific language for form-based services. *IEEE Transactions on Software Engineering* 25(3), 334–346 (1999)
5. Augusteijn, A.: Functional Programming, Program Transformations and Compiler Construction. PhD thesis, Department of Computing Science, Eindhoven University of Technology, The Netherlands (1993)
6. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
7. Backus, J.W.: Automatic programming: properties and performance of FORTRAN systems I and II. In: Proceedings of the Symposium on the Mechanisation of Thought Processes, Teddington, Middlesex, England, The National Physical Laboratory (November 1958)
8. Backus, J.W., et al.: Report on the algorithmic language ALGOL 60. *Communications of the ACM* 3(5), 299–314 (1960)
9. Balat, V.: Ocsigen: typing web interaction with objective Caml. In: Kennedy, A., Pottier, F. (eds.) Proceedings of the ACM Workshop on ML, Portland, Oregon, USA, pp. 84–94. ACM, New York (September 2006)
10. Bauer, C., King, G.: Java Persistence with Hibernate. In: Manning, Greenwich, NY, USA (2007)

11. Beck, K.: *Extreme Programming Explained*. Addison-Wesley, Reading (2000)
12. Bentley, J.L.: Programming pearls: Little languages. *Communications of the ACM* 29(8), 711–721 (1986)
13. Brabrand, C., Møller, A., Ricky, M., Schwartzbach, M.I.: PowerForms: Declarative client-side form field validation. *World Wide Web Journal* 3(4), 205–314 (2000)
14. Brabrand, C., Möller, A., Schwartzbach, M.I.: The < bigwig > project. *ACM Transactions on Internet Technology* 2(2), 79–114 (2002)
15. Bravenboer, M.: *Connecting XML processing and term rewriting with tree grammars*. Master's thesis, Utrecht University, Utrecht, The Netherlands (November 2003)
16. Bravenboer, M., Dolstra, E., Visser, E.: Preventing injection attacks with syntax embeddings. A host and guest language independent approach. In: Lawall, J. (ed.) *Generative Programming and Component Engineering (GPCE 2007)*, pp. 3–12. ACM, New York (October 2007)
17. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.16. Components for transformation systems. In: *ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM 2006)*, Charleston, South Carolina, pp. 95–99. ACM SIGPLAN, New York (January 2006)
18. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: *Stratego/XT Tutorial, Examples, and Reference Manual* (latest). Department of Information and Computing Sciences, Universiteit Utrecht, Utrecht, The Netherlands (2006), <http://www.strategoxt.org>
19. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A language and toolset for program transformation. In: *Science of Computer Programming (2008)*; Special issue on Experimental Systems and Tools
20. Bravenboer, M., van Dam, A., Olmos, K., Visser, E.: Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae* 69(1–2), 123–178 (2006)
21. Bravenboer, M., Visser, E.: Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In: Schmidt, D.C. (ed.) *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*, Vancouver, Canada, pp. 365–383. ACM Press, New York (October 2004)
22. Bravenboer, M., Visser, E.: Designing syntax embeddings and assimilations for language libraries. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) *MODELS 2007*. LNCS, vol. 4735. Springer, Heidelberg (2007)
23. Budinsky, F., Steinberg, D., Merks, E., Eilersick, R., Grose, T.J.: *Eclipse Modeling Framework*. Addison-Wesley, Reading (2004)
24. Chamberlin, D.D., Boyce, R.F.: SEQUEL: A structured english query language. In: Rustin, R. (ed.) *Proceedings of 1974 ACM-SIGMOD Workshop on Data Description, Access and Control*, Arbor, Michigan, pp. 249–264. ACM, New York (May 1974)
25. Christensen, A.S., Möller, A., Schwartzbach, M.I.: Extending Java for high-level web service construction. *ACM Transactions on Programming Languages and Systems* 25(6), 814–875 (2003)
26. Clinger, W.: Macros in scheme. *SIGPLAN Lisp Pointers* 4(4), 17–23 (1991)
27. Codd, E.F.: A relational model of data for large shared data banks. *Communications of the ACM* 13(6), 377–387 (1970)
28. Consel, C.: From a program family to a domain-specific language. In: Lengauer, C., Batory, D., Consel, C., Odersky, M. (eds.) *Domain-Specific Program Generation*. LNCS, vol. 3016, pp. 19–29. Springer, Heidelberg (2004)

29. Consel, C., Marlet, R.: Architecturing software using a methodology for language development. In: Palamidessi, C., Meinke, K., Glaser, H. (eds.) ALP 1998 and PLILP 1998. LNCS, vol. 1490, pp. 170–194. Springer, Heidelberg (1998)
30. Cook, S., Jones, G., Kent, S., Wills, A.C.: Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley, Reading (2007)
31. Cordy, J.: The TXL source transformation language. *Science of Computer Programming* 61(3), 190–210 (2006)
32. Czarnecki, K.: Overview of generative software development. In: Banâtre, J.-P., et al. (eds.) UPP 2004. LNCS, vol. 3566, pp. 313–328. Springer, Heidelberg (2005)
33. Czarnecki, K., Eisenecker, U.W.: Generative programming: methods, tools, and applications. Addison-Wesley, New York (2000)
34. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. *IBM Systems Journal* 45(3), 621–646 (2006)
35. de Jonge, M.: A pretty-printer for every occasion. In: Ferguson, I., Gray, J., Scott, L. (eds.) Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET 2000). University of Wollongong, Australia (2000)
36. Dmitriev, S.: Language Oriented Programming: The next programming paradigm (2004), <http://www.onboard.jetbrains.com/articles/04/10/lop/>
37. Dolstra, E.: The Purely Functional Software Deployment Model. PhD thesis, Utrecht University, Utrecht, The Netherlands (January 2006)
38. Dolstra, E., Bravenboer, M., Visser, E.: Service configuration management. In: James Whitehead, J.E., Dahlqvist, A.P. (eds.) 12th International Workshop on Software Configuration Management (SCM-12), Lisbon, Portugal, pp. 83–98. ACM, New York (September 2005)
39. Dolstra, E., Visser, E., de Jonge, M.: Imposing a memory management discipline on software deployment. In: Estublier, J., Rosenblum, D. (eds.) 26th International Conference on Software Engineering (ICSE 2004), Edinburgh, Scotland, pp. 583–592. IEEE Computer Society, Los Alamitos (May 2004)
40. Ducasse, S., Lienhard, A., Renggli, L.: Seaside: A flexible environment for building dynamic web applications. *IEEE Software*, pp. 56–63 (September/ October 2007)
41. Earley, J.: An Efficient Context-free Parsing Algorithm. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA (1968) (see also [42])
42. Earley, J.: An efficient context-free parsing algorithm. *Communications of the ACM* 13(2), 94–102 (1970)
43. Efftinge, S., Friese, P., Haase, A., Kadura, C., Kolb, B., Moroff, D., Thoms, K., Völter, M.: openArchitectureWare User Guide. Version 4.2 (2007), <http://www.openarchitectureware.org>
44. Ekman, T., Hedin, G.: Rewritable reference attributed grammars. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 144–169. Springer, Heidelberg (2004)
45. Ekman, T., Hedin, G.: The jastadd extensible java compiler. *SIGPLAN Notices* 42(10), 1–18 (2007)
46. Fowler, M.: Inversion of control containers and the dependency injection pattern (January 2004), <http://www.martinfowler.com/articles/injection.html>
47. Fowler, M.: Language workbenches: the killer-app for domain specific languages (2005), <http://www.martinfowler.com/articles/languageWorkbench.html>
48. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
49. Gray, R.W., Heuring, V.P., Levi, S.P., Sloane, A.M., Waite, W.M.: Eli: A complete, flexible compiler construction system. *Communications of the ACM* 35, 121–131 (1992)



50. Gray, R.W., Levi, S.P., Heuring, V.P., Sloane, A.M., Waite, W.M.: Eli: a complete, flexible compiler construction system. *Commun. ACM* 35(2), 121–130 (1992)
51. Greenfield, J., Short, K.: *Software Factories. Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, Chichester (2004)
52. Groenewegen, D., Visser, E.: Declarative access control for WebDSL: Combining language integration and separation of concerns. In: Schwabe, D., Curbera, F. (eds.) *International Conference on Web Engineering (ICWE 2008)*. IEEE CS Press, Los Alamitos (July 2008)
53. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The syntax definition formalism SDF – reference manual. *SIGPLAN Notices* 24(11), 43–75 (1989)
54. Hemel, Z., Kats, L., Visser, E.: Code generation by model transformation. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) *ICMT 2008*. LNCS, vol. 5063. Springer, Heidelberg (2008)
55. Hudak, P.: Building domain-specific embedded languages. *ACM Comput. Surv.* 28, 196 (1996)
56. Seam, J.: Seam - Contextual Components. A Framework for Java EE 5, 1.2.1.ga edition (2007), <http://www.jboss.com/products/seam>
57. Johnson, S.C.: YACC—yet another compiler-compiler. Technical Report CS-32, AT & T Bell Laboratories. Murray Hill, N.J (1975)
58. Kats, L., Bravenboer, M., Visser, E.: Mixing source and bytecode. A case for compilation by normalization. In: Kiczales, G. (ed.) *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2008)*. ACM Press, New York (2008)
59. Kats, L.C.L., Kalleberg, K.T., Visser, E.: Generating editors for embedded languages. integrating SGLR into IMP. In: Johnstone, A., Vinju, J. (eds.) *Proceedings of the Eighth Workshop on Language Descriptions, Tools, and Applications (LDTA 2008)*, Budapest, Hungary (April 2008)
60. Kelly, S., Tolvanen, J.-P.: *Domain-Specific Modeling. Enabling Full Code Generation*. John Wiley & Sons, Inc, Chichester (2008)
61. Kent, S.: Model driven engineering. In: Butler, M., Petre, L., Sere, K. (eds.) *IFM 2002*. LNCS, vol. 2335, pp. 286–298. Springer, Heidelberg (2002)
62. Klint, P.: A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology* 2(2), 176–201 (1993)
63. Knuth, D.E.: Backus Normal Form vs. Backus Naur Form. *Communications of the ACM* 7(12), 735–736 (1964)
64. Knuth, D.E.: On the translation of languages from left to right. *Information and Control* 8, 607–639 (1965)
65. Knuth, D.E.: Semantics of context-free languages. *Mathematical Systems Theory* 2(2), 127–145 (1968); Correction in: *Mathematical Systems Theory* 5(1), 95–96 (1971)
66. Knuth, D.E.: *The TEXbook. vol. A, Computers and Typesetting*. Addison-Wesley, Reading (1984)
67. Krishnamurthi, S., Hopkins, P.W., McCarthy, J.A., Graunke, P.T., Pettyjohn, G., Felleisen, M.: Implementation and use of the plt scheme web server. *Higher-Order and Symbolic Computation* 20(4), 431–460 (2007)
68. Lämmel, R., Visser, J.: Typed combinators for generic traversal. In: Krishnamurthi, S., Ramakrishnan, C.R. (eds.) *PADL 2002*. LNCS, vol. 2257, pp. 137–154. Springer, Heidelberg (2002)
69. Lamport, L.: *LaTeX: A Documentation Preparation System*. Addison-Wesley, Reading (1986)

70. Leijen, D., Meijer, E.: Domain specific embedded compilers. In: Proceedings of the 2nd conference on Domain-specific languages (DSL 1999), pp. 109–122. ACM Press, New York (1999)
71. Mak, S.: Developing interacting domain specific languages. Master's thesis, Utrecht University, Utrecht, The Netherlands, INF/SCR-07-20 (November 2007)
72. Mann, K.D.: *JavaServer Faces in Action*. Manning, Greenwich, NY, USA (2005)
73. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. *ACM Computing Surveys* 37(4), 316–344 (2005)
74. Nusairat, J.F.: *Beginning JBoss Seam*. Apress, New York (2007)
75. Nystrom, N., Clarkson, M.R., Myers, A.C.: Polyglot: An extensible compiler framework for Java. In: Hedin, G. (ed.) *CC 2003*. LNCS, vol. 2622, pp. 138–152. Springer, Heidelberg (2003)
76. OMG Architecture Board ORMSC. Model driven architecture. OMG document number ormsc/2001-07-01 (July 2001), [www.omg.org](http://www.omg.org)
77. Parr, T.: *The Definitive ANTLR Reference: Building Domain-Specific Languages* by Terence Parr. The Pragmatic Programmers (2007)
78. Parr, T.J.: Enforcing strict model-view separation in template engines. In: *WWW 2004: Proceedings of the 13th international conference on World Wide Web*, pp. 224–233. ACM, New York (2004)
79. Peyton Jones, S.L. (ed.): *Haskell98 Language and Libraries. The Revised Report*. Cambridge University Press (2003)
80. Queindec, C.: The influence of browsers on evaluators or, continuations to program web servers. In: *International Conference on Functional Programming (ICFP 2000)*, pp. 23–33. ACM, New York (2000)
81. Reps, T., Teitelbaum, T.: *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer, New York (1988)
82. Schmidt, D.C.: Model-driven engineering. *IEEE Computer* 39(2), 25–31 (2006)
83. Sheard, T., Peyton Jones, S.L.: Template metaprogramming for Haskell. In: Chakravarty, M.M.T. (ed.) *ACM SIGPLAN Haskell Workshop 2002*, pp. 1–16 (October 2002)
84. Simonyi, C., Christerson, M., Clifford, S.: Intentional software. In: *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA 2006)*, pp. 451–464. ACM, New York (2006)
85. Spinellis, D.: Notable design patterns for domain specific languages. *Journal of Systems and Software* 56(1), 91–99 (2001)
86. Spinellis, D., Guruprasad, V.: Lightweight languages as software engineering tools. In: *USENIX Conference on Domain-Specific Languages*, , pp. 67–76. USENIX Association (October 1997)
87. Stahl, T., Völter, M.: *Model-Driven Software Development*. Wiley, Chichester (2005)
88. Steele Jr, G.L.: Growing a language. *Higher-Order and Symbolic Computation* 12, 221–236 (1998); (Text of invited talk at OOPSLA 1998)
89. Sturm, T., von Voss, J., Boger, M.: Generating code from uml with velocity templates. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) *UML 2002*. LNCS, vol. 2460, pp. 150–161. Springer, Heidelberg (2002)
90. Sun Microsystems. JSR 220: Enterprise JavaBeans™, Version 3.0. Java Persistence API (May 2, 2006)
91. Tharp, A.L.: The impact of fourth generation programming languages. *SIGCSE Bull* 16(2), 37–44 (1984)



92. Thiemann, P.: WASH/CGI: Server-side web scripting with sessions and typed, compositional forms. In: Krishnamurthi, S., Ramakrishnan, C.R. (eds.) PADL 2002. LNCS, vol. 2257, pp. 192–208. Springer, Heidelberg (2002)
93. Thomas, D., Hansson, D.H.: Agile Web Development with Rails. The Pragmatic Bookshelf (2005)
94. Tomita, M.: Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems. Kluwer Academic Publishers, Dordrecht (1985)
95. Tratt, L.: Domain specific language implementation via compile-time meta-programming. ACM Transactions on Programming Languages and Systems (to appear, 2009)
96. van den Brand, M.G.J., de Jong, H., Klint, P., Olivier, P.: Efficient annotated terms. Software, Practice & Experience 30(3), 259–291 (2000)
97. van den Brand, M.G.J., Heering, J., Klint, P., Olivier, P.A.: Compiling language definitions: the ASF+SDF compiler. ACM Transactions on Programming Languages and Systems 24(4), 334–368 (2002)
98. van Deursen, A., Klint, P.: Little languages: Little maintenance? Journal of Software Maintenance 10(2), 75–92 (1998)
99. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: An annotated bibliography. SIGPLAN Notices 35(6), 26–36 (2000)
100. van Wijngaarden, J.: Code generation from a domain specific language. Designing and implementing complex program transformations. Master's thesis, Utrecht University, Utrecht, The Netherlands, INF/SCR-03-29 (July 2003)
101. Visser, E.: Syntax Definition for Language Prototyping. PhD thesis, University of Amsterdam (September 1997)
102. Visser, E.: Meta-programming with concrete object syntax. In: Batory, D., Consel, C., Taha, W. (eds.) GPCE 2002. LNCS, vol. 2487, pp. 299–315. Springer, Heidelberg (2002)
103. Visser, E.: Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In: Lengauer, C., Batory, D., Consel, C., Odersky, M. (eds.) Domain-Specific Program Generation. LNCS, vol. 3016, pp. 216–238. Springer, Heidelberg (2004)
104. Visser, E.: A survey of strategies in rule-based program transformation systems. Journal of Symbolic Computation 40(1), 831–873 (2005); Special issue on Reduction Strategies in Rewriting and Programming
105. Visser, E., Benaissa, Z.-e.-A., Tolmach, A.: Building program optimizers with rewriting strategies. In: Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP 1998), pp. 13–26. ACM Press, New York (1998)
106. W3C. Cascading Style Sheets, level 2. CSS2 Specification (May 1998), <http://www.w3.org/TR/REC-CSS2/>
107. Warmer, J.: A model driven software factory using domain specific languages. In: Akehurst, D.H., Vogel, R., Paige, R.F. (eds.) ECMDA-FA. LNCS, vol. 4530, pp. 194–203. Springer, Heidelberg (2007)
108. Wiedermann, B., Cook, W.R.: Extracting queries by static analysis of transparent persistence. In: Felleisen, M. (ed.) Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2007), pp. 199–210. ACM, New York (2007)
109. Wyk, E.V., Krishnan, L., Bodin, D., Schwerdfeger, A.: Attribute grammar-based language extensions for Java. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 575–599. Springer, Heidelberg (2007)

**Part II**  
**Short Tutorials**

# Model-Driven Engineering of Rules for Web Services

Marko Ribarić<sup>1</sup>, Dragan Gašević<sup>2</sup>, Milan Milanović<sup>3</sup>,  
Adrian Giurca<sup>4</sup>, Sergey Lukichev<sup>4</sup>, and Gerd Wagner<sup>4</sup>

<sup>1</sup> Mihailo Pupin Institute, Serbia

marko.ribaric@gmail.com

<sup>2</sup> School of Computing and Information Systems, Athabasca University, Canada

dgasevic@acm.org

<sup>3</sup> FON-School of Business Administration, University of Belgrade, Serbia

milan@milanovic.org

<sup>4</sup> Institute of Informatics, Brandenburg University of Technology at Cottbus, Germany

{lukichev,giurca,wagner}@tu-cottbus.de

**Abstract.** Web services are proposed as a way to enable loosely-coupled integration of business processes of different stakeholders. This requires effective development mechanisms that focus on modeling of business logic rather than on low-level technical details. This problem has been recognized by several researchers, and they have mainly proposed the use of process-oriented languages (e.g., UML Activity Diagrams). However, process-oriented approaches limit the definitions of Web services only to the context of concrete business processes (*where* the services are used). To overcome this limitation, in this paper, we propose a modeling approach that enables one to model Web services from the perspective of the underlying business logic regulating *how* Web services are used regardless of the context *where* they are used. This is done by modeling Web services in terms of message-exchange patterns, where each service is described by a (set of) rule(s) regulating how Web services' messages are exchanged. By leveraging the principles of model-driven engineering, we define a rule-based modeling language supporting the proposed modeling approach. More specifically, the rule-based modeling language supports reaction rules (also known as Event-Condition-Action rules) to model Web services in terms of message exchange patterns. Our approach is supported by an extension of the well-known UML tool Fujaba and by a number of model transformations for round-trip engineering between Web services and reaction rules.

## 1 Introduction

Developing interoperable and loosely coupled components is one of the most important goals for Web-based software industry with the main goal to bridge the gaps between many different stakeholders. Web services proved to be the most mature framework toward achieving that goal [32], since they are based on a set of XML-based standards for their description, publication, and invocation ([1], [2], [3]). Although Web services offer many benefits, here we name three important factors that constrain their development and use, as well as indicate the approach we propose to address these constraints.

First, developers of Web services mainly focus on platform specific and low-level implementation details (e.g., elements of Web Service Description Language – WSDL) [4]. The consequence is that they can hardly focus on the development of a system under study (e.g., a business process), and because of this, their productivity is affected. Moreover, developers have to implement many things manually, which may lead to potential execution errors, especially when trying to enrich the existing Web services with new functionalities. A promising way to solve this problem is to use a high-level modeling approach which will allow developers to focus on a problem domain rather than on an implementation technology. This is exactly why we propose using an approach based on Model Driving Engineering (MDE). By using MDE, we can first define a modeling language (i.e., metamodeling) that is suited for modeling specific problem domains (in our case business logic that should be supported by Web services). Models created by such modeling languages can later be transformed (by using model transformation languages) into different implementation platforms [8]. Although, there have been several attempts to leverage MDE principles to model Web services, they are still very low level oriented, as they again focus on technical details covered either by WSDL [4][9] or OWL-S [10][11].

Second, a higher-level approach to modeling business logic would be to use process- or workflow-oriented languages to model business process rather than just adding service constructs in existing modeling languages such as UML (i.e., creating UML Profiles) [10][11]. Examples of such languages are Business Process Modeling Notation and UML Activity Diagrams. In fact, researchers have already experimented with these languages where Web services are typically extracted from a global process-oriented model [5]. However, in that case, the definition of each specific Web service is depended on a context of its use inside the workflow [5]. The recent research on services indicates that one can hardly predict in which contexts services will be used [6]. Thus, there is a need to support workflow-independent services modeling, and yet to consider some potential patterns or conditions under which a specific service can be used. In this paper, we propose focusing the design perspective from the question *where* (or in what context) to the question *how* a service is used. To do so, our proposal is to leverage message-exchange patterns (MEPs) as an underlying perspective integrated into a Web service modeling language. This perspective has already been recognized by Web service standards [1], and our contribution is to raise this to the level of models.

Third, there are no/limited automatic mechanisms for updating Web services based on the business process changes. This is due to the fact that business systems are highly-dynamic and may change quite often. This is very important in policy driven-systems where policies are deleted, created or updated very often. On the other hand, the use of rules implies that they are defined in a declarative way, for example, by means of a language such as Jess or ILOG's JRules. That is why one, by using rules, can dynamically reflect business logic changes at run-time without the need to redesign the system. Since Web services are used for integration of business processes of various stakeholders, it is important for them to reflect changes in business logic, or policies, as good as possible. This means if we can support the definitions of services based on rules, we can also reflect changes in service-oriented processes dynamically. This has already been recognized in the service community, where Charfi and Mezini [7] demonstrate how rules can be used to make more dynamic business processes

based on service-oriented architectures. However, developers need development mechanisms that will allow them for building and later updating Web services based on such changes. Thus, we propose using rule-based approaches to modeling Web services [12] [13].

In this paper, we propose a modeling framework that integrates the three abovementioned perspectives for modeling Web services. In a nutshell, by using the MDE principles, we develop (i.e., extend) a modeling language that integrates both the rule and MEP perspectives indicated above. Besides the highlighted benefits, the MDE approach also allows us to integrate our modeling framework with languages commonly used for modeling vocabularies (e.g., UML class models) and constraints (e.g., OCL).

To explain in detail our modeling framework, in the next section, we first give a brief overview of the existing (standard) languages for Web services, rules, and rule modeling. In section 3, we describe our rule-based modeling foundation, including, UML-based Rule Language (URML) and REVERSE Rule Markup Language (R2ML) [15] with the emphasis on reaction rules, a type of rules that allows for modeling Web services from the perspective of MEPs, as described in Section 4. In Section 5, we describe process of transforming between rule-based models to the Web service standards, as a part of the tooling support that we have implemented to support our approach [16]. Before we conclude the paper, in Section 6, we compare our approach with some related works on modeling Web services.

## 2 Background

In this section, we give a brief overview of the technologies and techniques relevant to the problem under study. This includes a short description of Web services, rule language definitions based on MDE principles, and Web rule languages.

### 2.1 Web Services

A Web service is a loosely coupled component that exposes functionality to a client over the Internet (or an intranet) by using web standards such as HTTP, XML, SOAP, WSDL, and UDDI [10].

SOAP is an XML-based protocol for exchanging information in a decentralized, distributed environment. SOAP builds on XML and common Web protocols (HTTP, FTP, and SMTP) [17]. A SOAP message is the basic unit of communication between SOAP nodes. The “*envelope*” element represents the root of a SOAP message structure. It contains a mandatory *body* construct and an optional *header* construct [3]. The header construct is where meta-information can be hosted. In a large number of service-oriented architectures, the header is an important part of the overall architecture, and although it is optional it is rarely omitted.

Web Service Description Language (WSDL) is a language for describing both the abstract functionality of a service and the concrete details of a Web service [1]. At an abstract level, WSDL describes a Web service in terms of interfaces and the operations supported by the interfaces. An operation is an interaction with the service consisting of a set of (*input*, *output*, *infault* and *outfault*) messages exchanged between

the service and the other parties involved in the interaction [1]. The messages are described independently of a specific wire format by using a type system, typically XML Schema. WSDL also describes the point of contact for a service provider, known as the *endpoint* – it provides a formal definition of the endpoint interface and also establishes the physical location (address) of the service.

Potential requestors need a way to discover Web services descriptors. It is necessary that these descriptors are collected and stored in a central registry. The key part of the Universal Description Discovery and Integration (UDDI) specification [2] presents standardizing information inside such a registry as well as specifying the way the information can be searched and updated.

Regardless of how complex tasks performed by a Web service are, almost all of them require the exchange of multiple messages (e.g., SOAP) [18]. It is important to coordinate these messages in a particular sequence, so that the individual actions performed by the message are executed properly. Message exchange patterns (MEPs) are a set of templates that provide a group of already mapped out sequences for the exchange of messages [18]. This basically means that MEPs define how services should be used. The WSDL 2.0 specification defines three MEPs: 1) *in-only pattern* – supports a standard fire-and-forget pattern (i.e., only one message is exchanged); 2) *robust in-only pattern* – presents a variation of the in-only pattern that provides an option of sending a fault message, as a result of possible errors generated while transmitting, or processing data; 3) *in-out pattern* – presents a request-response pattern where two messages (input and output) must be exchanged.

## 2.2 Rule Modeling

There are currently two different standards for business (rule) modeling that have been developed for the different audiences. *Semantics of Business Vocabulary and Rules (SBVR)* is a standardization effort for business modeling that currently is now being finalized at the OMG [21]. *Production Rule Representation (PRR)* addresses the requirement for a common production rule representation, as used in various vendors' rules engines with normative considerations and high level modeling [22].

The *SBVR* standard [21] provides a number of conceptual vocabularies for modeling a business domain in the form of a vocabulary and a set of rules, and it is not suitable for rule specification and reasoning. In *SBVR*, meaning is kept separate from expression. As a consequence, the same meaning can be expressed in different ways. The *SBVR* specification defines a structured English vocabulary for describing vocabularies and verbalizing rules called *SBVR Structured English*.

The *PRR* standard [22] represents a formal modeling specification for production rules. It represents a vendor-neutral UML-friendly rule representation, by using metamodeling principles (i.e., it is defined by a metamodel for production rules). The *PRR* consists of two levels. First, the *PRR* core that is a standard rule representation model for general use. Second, *PRR* OCL that includes the *PRR* core and the extended OCL expression languages for the use of UML. *PRR* supports widely used rules (e.g., ILOG JRules or JBoss Rules). It should be indicted that both *PRR* and *SBVR* have a weak relation to Web standards, including, standards for Web services, rules, and ontologies.

### 2.3 Web Rules

As already mentioned, Web services are technology that is primarily used for integration of different processes. This basically means that different parties, besides using a standard for publishing descriptions of Web services (WSDL), also have to be able to share (and “understand”) their business vocabularies and rules. Since our approach is rule-based, we naturally expect that our rule-based models are compliant to the ongoing efforts in the area of rule interchange. In that area, researchers are trying to define a standard for rule interchange at the W3C consortium in the scope of the Rule Interchange Format (RIF) [14] initiative. The current state of this initiative is that it defines a set of requirements and use cases for sharing rules on the Web. However, there is no official submission to this standard yet, and we will name a few most relevant initiatives.

*RuleML* is a markup language for publishing and sharing rule bases on the Web [19]. RuleML builds a hierarchy of rule sublanguages upon the XML, RDF, XSLT, and Web Ontology Language (OWL) languages. The current RuleML hierarchy consists of derivation (e.g., SWRL and FOL), integrity (e.g., OCL invariants), reaction, transformation (e.g., XSLT) and production rules (e.g., Jess). RuleML is based on Datalog and its rules are defined in the form of an implication between an antecedent and a consequent, meaning that if a logical expression in the antecedent holds, the consequent must also hold.

*Semantic Web Rule Language (SWRL)* is a rule language defined on top of the OWL language [20]. Similar to RuleML rules, a SWRL rule is also in the form of an implication and is considered another (and more advanced) type of an axiom defined on top of the intrinsic OWL axiom types. This means that SWRL rules are usually used for defining integrity constraints similar to OCL in UML, but they can also be used for defining derivation rules. Both consequent and antecedent are collections (e.g., conjunctions) of atoms.

## 3 Modeling Foundation

Both the Web rule and rule modeling approaches presented in the previous section, serve very nicely the purpose of rule interchange and representation, respectively, independent of any specific platform. However, none of these languages offers a suitable modeling foundation that can be used for modeling Web services from a more abstract perspective of business rules and MEPs as requested in the introduction. Furthermore, neither of the Web rule languages can be used for modeling Web services. SWRL does not have a reaction behavior needed, while RuleML is only used for rule interchange without any syntax suitable for human comprehension and modeling. More importantly, neither of the language is developed by using MDE principles, which basically means that there is no separation between abstract syntax (i.e., metamodel), concrete syntax (both graphical and textual), and semantics. For each modeling language, this is important, as typically a textual concrete syntax is more suitable for machine processing; graphical concrete syntax is more suitable for human comprehension; and semantic constraints defined on top of the abstract syntax (e.g.,

OCL constraints over MOF-based metamodels) allow for more advanced processing (transformations, queries, views, and consistency checks) of language constructs.

In addition to the above aspects, which have to be supported by a Web service modeling language, we also set the following requirements. First, a rule-based approach to Web service modeling should be closely related to the already existing software modeling languages such as UML and preferably defined by MDE principles. Second, the language should be closely related to existing Web standards for defining ontologies (OWL) and rules (RIF) as discussed in Section 2.3.

We decided to use REVERSE I1 Rule Markup Language (R2ML), as it fully satisfies the above requirements. The language is defined by a MOF-based metamodel which is refined by OCL constraints that precisely define relations between the language's constructs, in addition to those defined by the metamodel; it has an XML schema defined as its concrete syntax; it has a UML-based graphical concrete syntax, so called UML-based Rule Modeling Language (URML); and it has a number of transformations with other rule languages (e.g., JBoss' Drools and OCL) allowing us to translate Web service models to the rule-based languages that can then regulate the use of Web services. In the rest of the section, we introduce R2ML and URML by mainly focusing on their support for modeling vocabularies and reaction rules (given the fact that our modeling of Web services is based on the use of this type of rules as explained in Sect. 4).

### 3.1 Rule-Based Modeling Language

R2ML is a rule language that addresses all the requests defined by the W3C working group for the standard rule interchange format [14]. The R2ML language is defined by MDE principles. The R2ML language can represent different types of rule constructs, that is, it can represent different types of rules [23], including, integrity rules, derivation rules, production rule, and reaction rules. Integrity rules in R2ML, also known as (integrity) constraints, consist of a constraint assertion, which is a sentence in a logical language such as first-order predicate logic or OCL. Derivation rules in R2ML are used to derive new knowledge (conclusion) if a condition holds. Production rules in R2ML produce *actions* if the *conditions* hold, while *post-conditions* must also hold after the execution of actions. A reaction rule is a statement of programming logic [24] that specifies the execution of one or more actions in the case of a triggering event occurrence and if rule conditions are satisfied. Optionally, after the execution of the action(s), post-conditions may be made true.

R2ML also allows one to define vocabularies by using the following constructs: basic content vocabulary, functional content vocabulary, and relational content vocabulary. Here we give short description of vocabulary constructs that we use in this paper. *Vocabulary* is a concept (class) that can have one or more *VocabularyEntry* concepts. *VocabularyEntry* is abstract concept (class) that is used for representing other concepts by its specialization. For example, one of the *VocabularyEntry-s* is an R2ML *Class* concept which represents the class element similar to the notion of the UML Class. An R2ML *Class* can have attributes (class *Attribute*), reference properties (class *ReferenceProperty*) and operations (class *Operation*).

Due to the importance for our Web service modeling approach, here we only describe the details of R2ML reaction rules. Reaction rules represent a flexible way



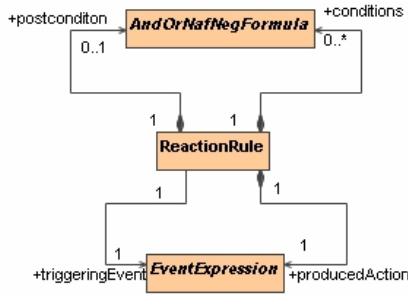


Fig. 1. The definition of reaction rules in the R2ML metamodel

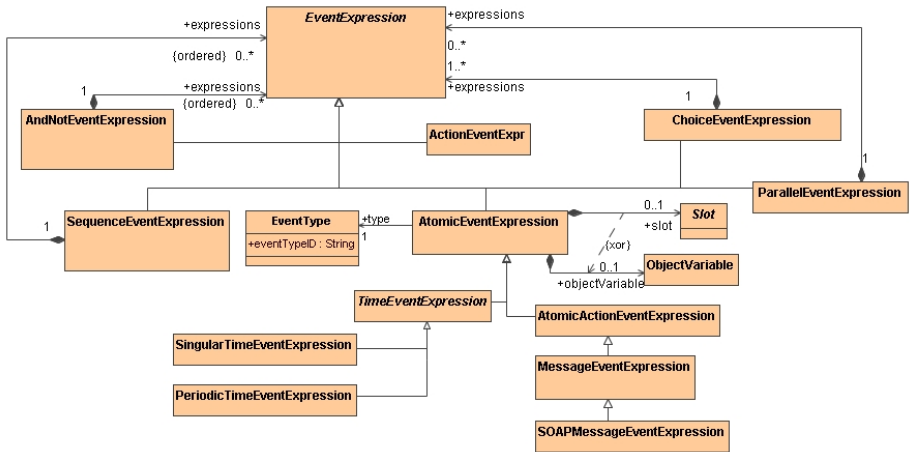


Fig. 2. Event expressions in the R2ML metamodel

for specifying control flows, as well as for integrating events/actions from a real life [24]. Reaction rules are represented in the R2ML metamodel as it is shown in Fig. 1: triggeringEvent is an R2ML EventExpression (Fig. 2); conditions are represented as a collection of quantifier free logical formulas; producedAction is an R2ML EventExpression and represents a system state change; and (optional) postcondition must hold when the system state changes.

The R2ML event metamodel defines basic concepts that are needed for dynamic rule behavior (Fig. 2). For the sake of modeling Web services, we are using *MessageEventExpression* for both triggering events and produced actions. *MessageEventExpression* is used for modeling messages that are part of the definition of Web services, including, input, output, in-fault, and out-fault messages. Each *MessageEventExpression* has its own type – *EventType*. In terms of WSDL, message types are defined by XML Schema complex types, while in R2ML, *EventType* is a subclass of *Class* (already defined as a part of the R2ML Vocabulary). This means that each *EventType* has its own attributes, associations, and all other features of R2ML classes.

### 3.2 UML-Based Rule Modeling Language

UML-Based Rule Modeling Language (URML) is a graphical concrete syntax of R2ML. URML is developed as an extension of the UML metamodel to be used for rule modeling. In URML, modeling vocabularies is done by using UML class models. Rules are defined on top of such models.

The URML reaction rules metamodel, which we use for modeling services, is shown in Fig. 3a. The figure shows components of a reaction rule: *Condition*, *Post-condition*, *RuleAction* and *EventCondition*. The figure also shows that reaction rules are contained inside the UML package which represents Web services operation. This means, that such packages have a stereotype <<operation>> in UML diagrams.

An instance of the *EventCondition* class is represented on the URML diagram as incoming arrow (e.g., see Fig. 4), from a UML class that represents either an input message or an input fault message of the Web service operations, to the circle that represents the reaction rule. The UML class that represents the input message (*inputMessage* in Fig. 3b) of the Web service operation is *MessageEventType* (a subclass of *EventType* from Fig. 2) and it is represented using the <<message event type>> stereotype on UML classes. The UML class that represents the input fault message (*inFault* in Fig. 3b) of the Web service operation is *FaultMessageEventType* in the URML metamodel. In URML diagrams, *FaultMessageEventType* is represented by the <<fault message event type>> stereotype on UML classes. *EventCondition* contains an object variable (*ObjectVariable* in Fig. 3c), which is a placeholder for an instance of the *MessageEventType* class. The object variable has a name that corresponds to the arrow annotation (*u* in Fig. 5), which represents *EventCondition*.

An instance of the *RuleAction* class is represented as an outgoing arrow on the URML diagram, from the circle that represents the reaction rule to the class that represents either an output message or an output fault message of the Web service operation. The UML class that represents the output message (*outputMessage* in Fig. 3c) of the Web service operation is *MessageEventType* and it is represented with the <<message event type>> stereotype on UML classes. The UML class that represents the output fault message (*outFault*) of the Web service operation is *FaultMessageEventType* in the URML metamodel and it is represented with the <<fault message event type>>

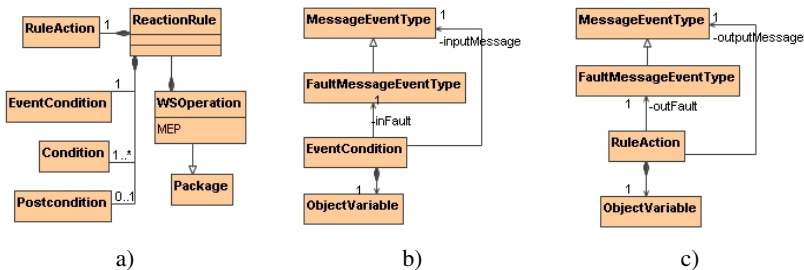


Fig. 3. a) Extension of the URML metamodel for reaction rules; b) Part of the URML metamodel for EventCondition; c) Extension of the URML metamodel for actions

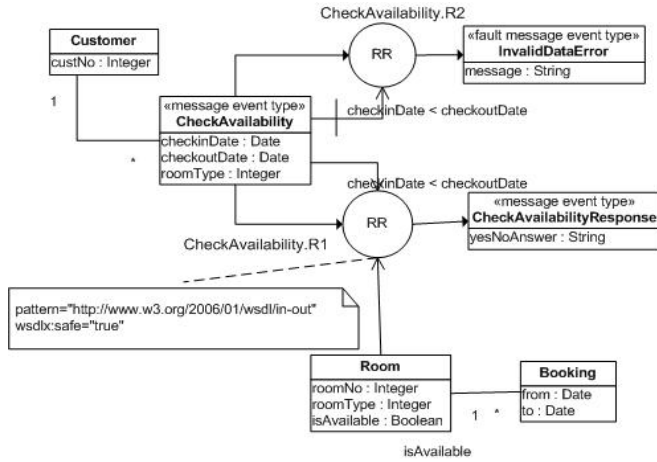


Fig. 4. PIM level of the in-out message exchange pattern with out-fault, presented in URML

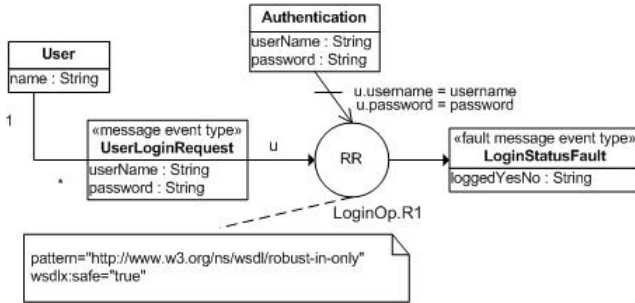


Fig. 5. The robust in-only message exchange pattern presented in URML

stereotype on UML classes. *RuleAction* contains an object variable (*ObjectVariable*), which represents an instance of the *MessageEventType* class.

### 4 Rule-Based Modeling of Services

In the previous section, we have introduced the definitions of both R2ML and URML that are used as a modeling foundation for modeling Web services. In our approach of modeling Web services, we look from the perspective of the potential patterns of the use of services. That is, we model services from the perspective of MEPs. Our modeling approach is message-oriented according to IBM’s classification of service modeling approaches [31]. It is important to point out that we first start from the definition of a business rule that corresponds to a MEP under study, but without considering the Web services that might be developed to support that rule. In this way, unlike other approaches to modeling of Web services, we are focused on the business rules describing how particular services are used, but without explicitly stating that we are

talking about Web services. This approach enables us, not only to focus on the problems under study and the underlying business logic regulating the process of the use of Web services, but also we are able to translate such Web service modeling to both Web service languages and rule-based languages that can regulate how services are used at run-time. This is the core benefit of our approach, which distinguishes it from other relevant solutions

In order to illustrate how our rule-based modeling approach is used, in the rest of the section, we describe how two MEPs are modeled in URML, including, the in-out MEP and the robust in-only MEP.

#### 4.1 In-Out Message Exchange Pattern

The *in-out* MEP consists of exactly two messages: when a service receives an input message, it has to reply with an output message. According to the WSDL specification [1], the so-called “message triggers fault propagation rule” can be applied to this pattern. This means that a fault message must be delivered to the same target node as the message it replaces, unless otherwise specified by an extension or binding extension. Here, we consider the case when the fault message replaces the output message of the service. To show how this variation of the in-out MEP can be modeled by using reaction rules, we start our discussion from an example of the in-out MEP with an out-fault message. Let us consider the following business rule:

*On a customer request for checking availability of a hotel room during some period of time, if the specified check-in date is before the specified check-out date, and if the room is available, then return to the customer a response containing the information about availability of the room, or if this is not the case return a fault message.*

The business rule is an in-out MEP and can be represented by two reaction rules, represented in semi-formal pseudo rule syntax:

```

ON CheckAvailability(checkinDate, checkoutDate)
IF checkinDate < checkoutDate AND isAvailable(Room)
THEN DO CheckAvailabilityResponse("YES")

ON CheckAvailability(checkinDate, checkoutDate)
IF NOT checkinDate < checkoutDate THEN
DO InvalidDataError("Check-in date is more than check-out date")

```

In order to have these rules represented using our modeling notation, and also to be able to relate the rules with the elements of vocabularies, we model these rules by using URML. It is important to stress, as already indicated in the beginning of Sect. 3, that all rules-based systems define business rules on top of business vocabularies. Thus, the URML graphical notation enables us to define business rules regulating the use of Web services by leveraging a human-comprehensible, and yet formally consistent representation, with the underlying business vocabularies (i.e., UML class models). These above-mentioned two reaction rules represented in URML are shown in Fig. 4.

Once we have modeled the business rule with the two reaction rules, we map the above reaction rules to Web services. In this particular case, we have the following situation: triggering event of either rule (i.e., *CheckAvailability*) maps to the input message of the Web service operation. The action of the first reaction rule (i.e., *CheckAvailabilityResponse*), which is triggered when a condition is true, maps to the

output message of the Web service operation. The action of the second reaction rule (i.e., *InvalidDataError*), triggered on a false condition, maps to the out-fault message of the Web service operation. Let us stress again that these two reaction rules are used to represent a business rule modeling the in-out MEP with the variation “message triggers fault propagation rule” (in URML, they are grouped in the <<operation>> package). Otherwise, for modeling the basic in-out MEP, only one (first) reaction rule would be used. Besides the classes that model exchanged messages, the URML diagrams also model conditions (e.g., *checkinDate < checkoutDate*). Such condition constructs are modeled by using OCL filters. OCL filters are based on a part of OCL that models logical expressions, which can be later translated to R2ML logical formulas, as parts of reaction rules. However, these OCL filters can not be later translated to Web service descriptions (e.g., WSDL), as those languages can not support such constructs. But, we can translate our URML models into rule-based languages (e.g., Jess or Drools). This means that for each Web service, we can generate a complementary rule, which fully regulates how its attributed service is used. Another benefit of this approach is that our generated Web services, and their regulating rules, are consistent, as they originate (i.e., that are generated from) the same rule-based Web service model.

## 4.2 Robust In-only Message Exchange Pattern

The *robust in-only* MEP consists of exactly one input message: service expects one message and it is not obliged to send a response back, but it considers an option of sending a fault message as a result of generated errors in the process of transmitting or processing data. An example of the robust in-only pattern that we use in this section is presented with the following business rule:

*When a customer sends a logging request, if the authentication fails send a message to the customer informing her of the failed login.*

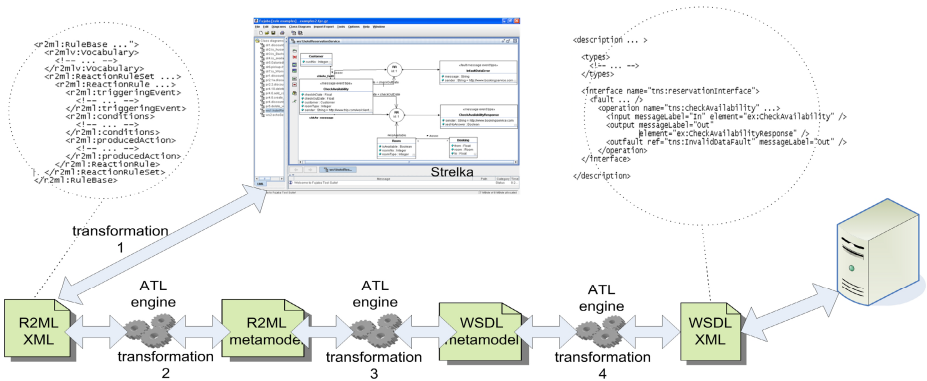
This business rule is represented with one reaction rule that has a triggering event *UserLoginRequest*, condition NOT *Authenticate*, and action *LoginFailed*:

<pre>ON UserLoginRequest(username, password) IF NOT Authenticate(username, password) DO LoginFailed ("Wrong password")</pre>
--

An URML diagram that models the corresponding Web service operation is presented in Fig. 5. The input message of the operation is the *UserLoginRequest* <<message event type>> stereotyped class, with the username and password attributes. The event arrow from the <<message event type>> stereotyped class to the rule circle is annotated with the rule variable *u*. The rule has a condition, which checks whether the username and password, provided in the event, correspond to the internal username and password. If username and password are not correct, then the out-fault *LoginFailed* is sent back to the user.

## 5 Model Transformations

In this section, we briefly describe how we automate the mappings between Web services (i.e., WSDL) and URML as described in the previous section. In Fig. 6, we



**Fig. 6.** Transformation chain for bidirectional mapping between URML and WSDL

give the tools and transformations that we have developed to support our modeling framework. For modeling with URML, we have developed a plug-in (so called Strelka) for the well-known Fujaba UML tool. This plug-in fully supports the URML modeling as described in Section 4. The native serialization of URML models in Strelka is the R2ML XML concrete syntax. To support generation of WSDL-based Web services, we need to translate R2ML XML-based models to WSDL. It is important to say that in Strelka, we also support transformation from WSDL documents to R2ML models (i.e., the reverse transformation), in order to enable reverse engineering of existing Web services, thus enabling an extraction of business rules that were already integrated into to the implementation of Web services.

We have decided to implement transformation from R2ML to WSDL at the level of metamodels by using the model transformation language ATL. An alternative could be to use the XSTL language to implement transformation between the R2ML and WSDL XML-based concrete syntax. However, as per our requirements defined in the beginning of Section 3, we consider that a definition of a modeling language such as R2ML, consists of several components, including, abstract syntax (i.e., metamodel), one or more concrete syntax, and semantics. As on top of a metamodel (i.e., abstract syntax) one can define constraints (i.e., OCL invariants) to fully support the language’s semantics, we then need to enforce such constraints during the transformation process in order to produce semantically compliant models to the target language’s semantics. This is especially important in the case of rule languages, as they use automatic reasoning (e.g., based on the RETE algorithm) as a way for determining the flow of actions to be executed. XML and its definition languages (DTD and XML Schema) do not have any constraining mechanism such as OCL, and thus it is not possible to have a guaranty that XSLT-translated models are fully semantically correct with respect to the semantics of the target language. This is particularly important for the transformation from WSDL to R2ML. Simply, XSLT can not check whether a model being transformed is complaint with its MOF-based metamodel and its additional OCL constraints (as model transformation languages can). One can only check whether the obtained XML model is compliant to the target XML Schema, but not during the transformation process. Yet, as already indicated, XML schema per se does not have constraining mechanism as OCL provides to MOF-based metamodels.

Given that this exploration significantly overcomes the scope of the discussion in this paper, we refer interested readers to the surveys such as [33] where they can find further benefits stemming from the use of model transformation languages over those specially tailored for XML (i.e., XSTL).

To support our approach, we needed to implement a number of transformations between different languages and their representation (all of them are bidirectional):

- *URML and R2ML XML concrete syntax* (transformation 1. on Fig. 6). This is the only transformation that is not implemented by using ATL [25], because Fujaba does not have explicitly defined metamodel in a metamodeling language such as MOF. We based this transformation by using JAXB<sup>1</sup>. JAXB guarantees that the R2ML XML rule sets comply with the R2ML XML schema.
- *R2ML XML-based concrete syntax and R2ML metamodel* (transformation 2. on Fig. 6). This transformation is important to bridge concrete (XML) and abstract (MOF) syntax of R2ML. This is done by using ATL and by leveraging ATL's XML injector and extractor for injecting/extracting XML models into/from the MOF-based representation of rules.
- *R2ML metamodel and WSDL metamodel* (transformation 3. on Fig. 6). This transformation is the core of our solution and presents mappings between R2ML and WSDL at the level of their abstract syntax.
- *WSDL XML-based concrete syntax and WSDL metamodel* (transformation 4. on Fig. 6). This transformation is important to bridge concrete (XML) and abstract (MOF) syntax of WSDL. This is also done by using ATL by leveraging ATL's XML injector and extractor.

Due to the size constraints for this paper, we only explain mappings between R2ML and WSDL. In Table 1, we present the conceptual mapping between the R2ML and WSDL metamodels. Even in this table, due to the same size constraint, we do not present parts related to the mapping between the XML Schema language (which WSDL uses for defining message types and vocabularies) and the R2ML vocabulary.

Fig. 7 shows a transformation rule named *Description*, presented in the QVT graphical notation [26]. This rule maps R2ML *RuleBase* (root) element to the WSDL *Description* (root) element. This rule applies that R2ML *Vocabulary* is mapped to WSDL *ElementType* and R2ML *ReactionRuleSet* is mapped to both WSDL *Service*

**Table 1.** An excerpt of the mapping between the WSDL metamodel to the R2ML metamodel

WSDL metamodel	R2ML metamodel
Description	RuleBase
ElementType	Vocabulary
Interface	ReactionRuleSet
Operation	ReactionRule
Input	MessageEventExpression
Infault	MessageEventExpression
Output	MessageEventExpression
Outfault	MessageEventExpression

<sup>1</sup> <https://jaxb.dev.java.net/>

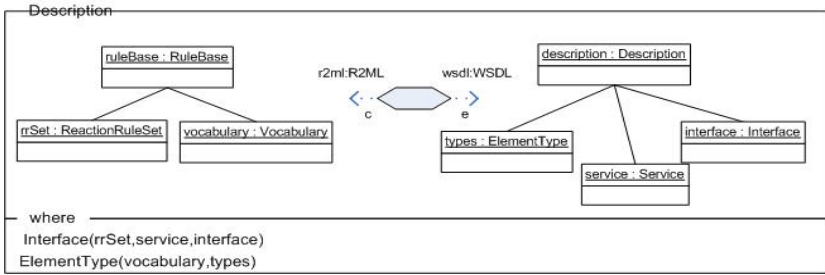


Fig. 7. Mappings between R2ML rules and WSDL service definitions

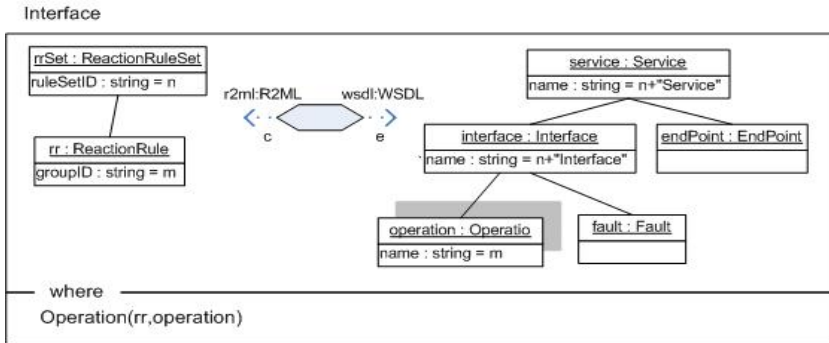


Fig. 8. Mapping between R2ML reaction rule sets and WSDL interfaces

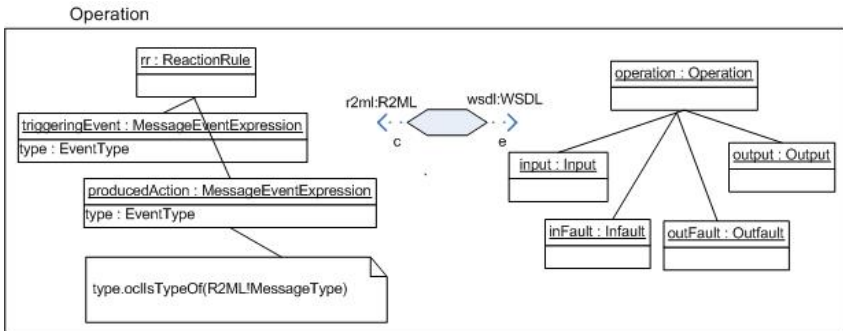


Fig. 9. Mappings between R2ML reaction rules and WSDL messages

and *Interface*. The *where* compartment indicates that every time the *Description* rule is executed, the *Interface* and *ElementType* rules must also be executed.

Fig. 8 shows a transformation rule called *Interface*. This rule transforms an R2ML *ReactionRuleSet* element to WSDL *Service* and *Interface* elements. The *where* compartment indicates that every time the *Interface* rule is executed, the *Operation* rule also has to be executed (i.e., *Operation* rule is a lazy rule). The WSDL *Interface*



element can contain multiple *Operation* elements, so for every *Operation* element an *Operation* rule will be executed.

Fig. 9 shows a transformation rule named *Operation*. This rule transforms the R2ML *ReactionRule* element, to the WSDL *Operation* element. We do not have a *where* compartment in this figure, because WSDL *Operation* does not have to contain *Input*, *Output*, *Infault* and *Outfault* elements. This figure also defines a constraint on the *producedAction* element of R2ML, which specifies that the *output* (message) element is created from the *producedAction* element of type *R2ML!MessageType*, while the *outfault* element is created from the *producedAction* element of the type *R2ML!FaultMessageType* (i.e., it is not *R2ML!MessageType*).

## 6 Related Work

In this section, we compare the proposed approach with some relevant solutions to the modeling of Web services that are based on the MDE principles.

Bezivin et al. [9] demonstrate how one can take advantage of MDE to develop e-business applications. In their approach, they start from UML and the UML profile for Enterprise Distributed Object Computing (EDOC) to define platform-independent models of e-business applications. In the next step, they translate such models into models that are based on metamodels of Java, WSDL, and JWSDP. Although this approach uses ATL, similar to what we do, it does not provide two way transformations between models and service implementation. This is unlike our solution, because we support two-way transformations, and thus we enable reverse engineering of the existing Web services. Next, the translation of regular UML models, i.e., UML class related elements, into WSDL is limited, as one does not have enough expressivity in UML diagrams to define all details of WSDL (i.e., one can not distinguish between input and output messages). This issue is addressed by using the UML profile for EDOC, but this approach is more focused on modeling distributed components rather on modeling business logics. In our approach, the use of reaction rules in URML models enables us to be closer to business processes, while OCL filter expressions can even further specify conditions under which a message can happen. This is very useful for potential integration of business process based on the use of Web services [27]. Of course, such filter expressions overcome the potentials of the Web service technology, as they presume that there should also be a rule-based engine, which is able to interpret such conditions. However, the research on semantic Web services [6] and Web rules [28] demonstrates that this issue attracts more attention in the Web service research community.

Vara et al. [4] define a metamodel for WSDL and its corresponding UML profile that is used for modeling of Web services in the MIDAS-CASE MDA-based tool for Web information system development. They also support automatic generation of the respective WSDL description for the Web services that is being modeled. Although the MIDAS framework supports platform-independent models of services and service compositions, their definitions are very incomplete and one can hardly generate complete service models automatically. In our approach, we do not strictly base models of services on workflows in which services will be used, as we wanted to focus on *how* services are used (i.e., MEPs).

Gronmo et al. [11] and Timm and Gannod [10] propose an approach to modeling semantic Web services by using UML profiles. Gronmo et al. have the goal to define platform-independent models of services, which can be translated to semantic Web services (OWL-S and WSMO). In their work, they abstract concepts from OWL-S and WSMO, and extend UML activity diagrams (i.e., they define a UML profile) accordingly, while for defining vocabularies they use the Ontology Definition Meta-model, and its corresponding UML profile defined in [29]. Since both these approaches use XSLT, they can hardly translate service pre- and post-conditions from their UML definitions (i.e., OCL) to the languages of the potential target platforms (e.g. Jess). This approach does not consider modeling usage patterns or MEPs or error handling (i.e., business logic) like we do, but instead it focuses on the details specific for service platforms.

Manolescu et al. propose a model-driven approach to designing and deploying service-enabled web applications [5]. This work is different from the ones above in the following aspects: i) it extends WebML, a well-known modeling language for developing Web applications; ii) it uses E-R models for defining data types; and iii) it focuses on the usage patterns (MEPs) of Web services in order to define WebML extensions. Their modeling approach is based on the use of MEPs, but they are considering MEPs used in WSDL 1.1, and for each MEP they define corresponding WebML primitive. These new WebML primitives can be used in hypertext models of Web applications. This approach is the most similar to ours; both approaches fully focus on modeling business processes and potential usage patterns. However, the WebML approach does not explicitly define a Web service description, but it infers it from the definitions of hypertext models in which the service is used, which makes this process context dependent. That is, services are defined inside specific WebML workflows unlike our approach where we define workflow-independent services. In addition, they do not consider the use of preconditions of Web services, which is important for the reusability of services in different contexts. Although WebML has support for exception handling [30], this is also focused on the workflow level rather than on a service level. Finally, the WebML approach does not support reverse engineering of Web services.

## 7 Conclusion

In the paper, we have demonstrated how the use of MDE principles can enable for rule-based modeling of Web services. By using the MDE principles we have been able to develop a framework for modeling Web services from the perspective of *how* services are used in terms of message-exchange patterns (MEPs). Our approach enables developers to focus on the definition of business rules, which regulate MEPs, instead of focusing on low level Web service details or on contexts where services are used (i.e., workflows). Rules in modeling services are a metaphor that is much closer to the problem domain. As such, rule-based models of services are much closer to business experts, and the process of knowledge/requirements elicitation is more reliable, as well as collaboration between service developers and business experts. Due to the declarative nature of rules, the business logic can easier be updated without

the need to change to whole system. We only have to update rules responsible for a part of logic that has changed and the update will automatically be reflected in the rest of the system. By leveraging MDE, we have defined a rule-based modeling language that can be managed by universal MDE tools (e.g., QVT). The use of model transformations allows for transforming platform independent models of business logic to specific platforms such as Web services. Moreover, for each Web service, we can also generate a rule that will fully regulate the behavior of the service (i.e., how the service is used), and thus make sure that the business logic is fully followed. In this paper, we have not explained that part of the solution, but we are going to report on that in our future papers.

Our solution has much broader potentials that overcome the pure translation between rule-based business models and Web services. Unlike the WSDL definition of Web services, our models also have an option for defining pre- and post-conditions of services. However, WSDL can not express pre- and post-conditions in the descriptions of Web services. That is, Web service tools can not automatically support conditions under which some services can be used (as defined by OCL filters in Section 4). To address this problem, our current activities are two-fold. First, we expand our approach on W3C's Semantic Annotations for WSDL (SAWSDL) recommendation. In that case, we use R2ML rules along with WSDL documents to enable for publishing pre- and post-conditions of services. Second, we are exploring how the W3C's Web Service Policy Framework (WS-Policy) recommendation can be integrated into our solution. While these activities are covering description and publishing of pre- and post-conditions of rules, we have mechanisms for regulating the use of services as already mentioned. In our current activities, we have developed transformations from our R2ML reaction rule-based models to several production rule languages (e.g., Drools, Jena2, and Jess). Our particular focus is on Drools, as the use of Drools allows us to directly enforce business rules to regulate the use of Web services deployed on JBoss' application server.

## References

- [1] Web Services Description Language (WSDL) Ver. 2.0 Part 1: Core Language. W3C Candidate Rec, <http://www.w3.org/TR/2007/REC-wsd120-20070626>
- [2] UDDI Ver. 3.0.2. OASIS v3.htm (2004), <http://uddi.org/pubs/uddi>
- [3] SOAP Ver. 1.2 Part 1: Messaging Framework. W3C Recommendation, <http://www.w3.org/TR/soap12-part1/>
- [4] Vara, J., de Castro, V., Marcos, E.: WSDL Automatic Generation from UML Models in a MDA Framework. *Int. J. of Web Services Pract.* 1(1-2), 1–12 (2005)
- [5] Manolescu, I., et al.: Model-driven design and deployment of service-enabled web applications. *ACM Trans. Inter. Tech.* 5(3), 439–479 (2005)
- [6] Sheth, A., Verma, K., Gomadam, K.: Semantics to energize the full services spectrum. *Communication of the ACM* 49, 55–61 (2006)
- [7] Charfi, A., Mezini, M.: Hybrid Web service composition: Business processes meet business rules. In: *Proc. of 2nd Int'l Conf. on Service Oriented Comp.*, pp. 30–38 (2004)
- [8] Schmidt, D.C.: Model-Driven Engineering. *Computer* 39(2), 25–31 (2006)

- [9] Bezivin, J., Hammoudi, S., Lopes, D., Jouault, F.: Applying MDA approach for Web Service Platform. In: Proc. of the 8th IEEE Int. Conf. on Enterprise Distributed Object Computing Conf., pp. 58–70 (2004)
- [10] Timm, J., Gannod, G.: A Model-Driven Approach for Specifying Semantic Web Services. In: Proc. of IEEE Int'l Conf. on Web Services, pp. 313–320 (2005)
- [11] Gronmo, R., Jaeger, M.C., Hoff, H.: Transformations between UML and OWL-S. In: Proc. of the 1st European Conf. on Model Driven Architecture - Foundations and Applications, pp. 269–283 (2005)
- [12] McClintock, C., de Sainte Marie, C.: ILOG's position on rule languages for interoperability. In: W3C Rule Languages for Interoperability (2005)
- [13] Ross, R.G.: Principles of the Business Rule Approach. Addison-Wesley, Reading (2003)
- [14] RIF Use Cases and Requirements. W3C Working Draft, <http://www.w3.org/TR/rif-ucr>
- [15] Lukichev, S., Wagner, G.: Visual rules modeling. In: Proceedings of the 6th International Conference Perspectives of Systems Informatics, pp. 467–673 (2006)
- [16] Lukichev, S., Wagner, G.: UML-based rule modeling with Fujaba. In: Proceedings of the 4th International Fujaba Days 2006, pp. 31–35 (2006)
- [17] Coyle, F.: XML, Web Services, and the Data Revolution. Addison Wesley, Reading (2002)
- [18] Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall PTR, Englewood Cliffs (2005)
- [19] Hirtle, D., et al.: Schema Specification of RuleML 0.91 (2006), <http://www.ruleml.org/spec/>
- [20] Horrocks, I., et al.: SWRL: A Semantic Web Rule Language Combining OWL and RuleML, W3C Member Sub. (2004), <http://www.w3.org/Submission/SWRL/>
- [21] Object Management Group: Semantics of Business Vocabulary and Business Rules (SBVR) – Interim Specification. OMG Document – dtc/06-03-02 (2006)
- [22] Object Management Group: Production Rule Representation (PRR) – revised submission. OMG Document - bmi/07-08-01 (2007)
- [23] Wagner, G., Giurca, A., Lukichev, S.: R2ML: A General Approach for Marking up Rules, In: Dagstuhl Seminar Proc. 05371 (2006)
- [24] Giurca, A., Lukichev, S., Wagner, G.: Modeling Web Services with URML. In: Proceedings of Workshop Semantics for Business Process Management (2006)
- [25] Atlas Transformation Language - User Manual, ver. 0.7, ATLAS group, Nantes, [http://www.eclipse.org/gmt/at1/doc/ATL\\_User\\_Manual.v0.7.pdf](http://www.eclipse.org/gmt/at1/doc/ATL_User_Manual.v0.7.pdf)
- [26] MOF 2.0 Query/View/Transformation Specification, OMG document ptc/05-11-01 (2005), <http://www.omg.org/docs/ptc/05-11-01.pdf>
- [27] Milanovic, M., Gasevic, D., Giurca, A., Wagner, G., Devedzic, V.: On interchanging between OWL/SWRL and UML/OCL. In: Proc. of the OCLApps Workshop, pp. 81–95 (2006)
- [28] Nagl, C., Rosenberg, F., Dustdar, S.: VIDRE– A distributed service oriented business rule engine based on RuleML. In: Proc. of the 10th IEEE Int'l Enterprise Distributed Object Computing Conference, pp. 35–44 (2006)
- [29] Gasevic, D., Djuric, D., Devedzic, V.: Bridging MDA and OWL ontologies. Journal of Web Engineering 4(2), 119–134 (2005)
- [30] Brambilla, M.S., et al.: Exception handling in workflow-driven web applications. In: Proc. of the 14th Int'l WWW Conference, pp. 170–179 (2005)

- [31] Brown, A.W., et al.: A Practical Perspective on the Design and Implementation of Service-Oriented Solutions. In: Proc. of the 10th ACM/IEEE 10th Int'l Conf. on Model Driven Engineering Languages and Systems, pp. 390–404 (2007)
- [32] Margaria, T.: Service Is in the Eyes of the Beholder. *Computer* 40(11), 33–37 (2007)
- [33] Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. *IBM Systems Journal* 45(3), 621–645 (2006)

# An Introduction to Context-Oriented Programming with ContextS

Robert Hirschfeld<sup>1</sup>, Pascal Costanza<sup>2</sup>, and Michael Haupt<sup>1</sup>

<sup>1</sup> Hasso-Plattner-Institut, Universität Potsdam, D-14482 Potsdam, Germany  
{robert.hirschfeld,michael.haupt}@hpi.uni-potsdam.de

<sup>2</sup> Programming Technology Lab, Vrije Universiteit Brussel, B-1050 Brussels, Belgium  
pascal.costanza@vub.ac.be

**Abstract.** Context-oriented Programming, or COP, provides programmers with dedicated abstractions and mechanisms to concisely represent behavioral variations that depend on execution context. By treating context explicitly, and by directly supporting dynamic composition, COP allows programmers to better express software entities that adapt their behavior late-bound at run-time. Our paper illustrates COP constructs, their application, and their implementation by developing a sample scenario, using ContextS in the Squeak/Smalltalk programming environment.

## 1 Introduction

Every intrinsically complex application exhibits behavior that depends on its context of use. Here, the meaning of context is broad and can range from obvious concepts such as location, time of day, or temperature over more technical properties like connectivity, bandwidth, battery level, or energy consumption to a user's subscriptions, preferences, or personalization in general.

Besides these examples of context that are often associated with the domain of ambient computing, the computational context of the program itself, for example its control flow or the sets or versions of libraries used, can be an important source of information for affecting the behavior of parts of the system.

Even though context is a central notion in a wide range of application domains, there is no direct support of context-dependent behavior from traditional programming languages and environments. Here, the expression of variations requires developers to repeatedly state conditional dependencies, resulting in scattered and tangled code.

This phenomenon, also known as crosscutting concerns, and some of the associated problems were documented by the aspect-oriented programming (AOP [16]) and the feature-oriented programming (FOP [2]) communities. The focus of AOP is mainly on the establishments of inverse one-to-many relationships [17] to achieve their vision of quantification and obliviousness [10]. FOP's main concern is the compile-time selection and combination of variations, and the necessary algebraic means to reason about such layer compositions [3].

	AOP	FOP	COP
Inverse dependencies	●		
1:n relationships	●		
Layers		●	●
Dynamic activation			●
Scoping	●		●

**Fig. 1.** Properties of AOP, FOP, and COP

Context-oriented programming (COP [6,14]) addresses the problem of dynamically composing context-dependent concerns, which are potentially crosscutting. COP takes the notion of FOP layers and provides means for their selection and composition at run-time. While FOP mechanisms are applied at compile-time—with the effect that, during program execution, layers as a distinct entity are no longer available—, COP preserves layers, adds the notion of dynamic layer activation and deactivation, and provides dynamic scoping to delimit the visibility of their composition as needed (Figure 1). With the dynamic scoping mechanisms offered by COP implementations, layered code can be associated with the units it belongs to and can be composed into or removed from the system depending on its context of use.

There are several COP extensions to popular programming languages such as ContextL for Lisp [6], ContextS for Squeak/Smalltalk [14], ContextR for Ruby, ContextPy for Python, and ContextJ\* for Java [14]. Here, we will focus on ContextS. Our paper is meant to be used mainly as a tutorial, describing ContextS in how it can be applied to the implementation of context-dependent behavioral variations.

The remainder of our paper is organized as follows: We give an overview of COP in Section 2. In Section 3 we introduce some of the COP extensions provided with ContextS which are applied to an example presented in Section 4. After some recommendations for further reading in Section 5 we conclude our paper with Section 6.

## 2 Context-Oriented Programming

COP, as introduced in [6,14], facilitates the modularization of context-dependent behavioral variations. It provides dedicated programming abstractions and mechanisms to better express software entities that need to change their behavior depending on their context of use.

Based on the implementation of several application scenarios and the development of language extensions necessary for them, we have identified behavioral variations, layers, dynamic activation, contextual information, and proper scoping mechanisms as essential properties of COP support:

**Behavioral variations.** There is a means to specify behavioral variations, typically ranging from new to modified or even removed behavior. Here, partial definitions of modules of the underlying programming model such as procedures, methods, or classes are prime candidates for being expressed as behavioral variations.

**Layers.** There needs to be a means to group related behavioral variations into layers. As first-class entities, layers can be explicitly referred to at run-time.

**Activation/deactivation.** Individual layers or combinations of them can be dynamically activated and deactivated, giving explicit control to programmers over their composition – including the point in time of their activation/deactivation as well as the desired sequence of their application.

**Context.** COP adopts a very broad definition of context: Context is everything that is computationally accessible. With that, we do not limit context to a particular concept, but encourage a wide spectrum of context representations most suitable for a specific application or system.

**Scope.** The scope of a layer activation or deactivation can be controlled explicitly so that simultaneous compositions affect each other only to the degree required by the program.

In the following, we will use message dispatch to show how COP is a continuation of previous work. While we do not require message dispatch as a base for any COP implementation, we do believe that this illustration will help to better understand how COP builds on procedural, object-oriented, and subjective programming (Figure 2).

**1D dispatch.** *Procedural programming* offers only one dimension to associate a unit of behavior with its activation [18]. Names used in procedure calls are directly mapped to procedure implementations ( $\langle m \rangle$ , Figure 2a).

**2D dispatch.** *Object-oriented programming* already uses two dimensions to associate a unit of behavior with its activation [18]. Names used at the activation site are mapped to a method implementation with the same name and the receiver it is defined in ( $\langle m, R \rangle$ , Figure 2b).

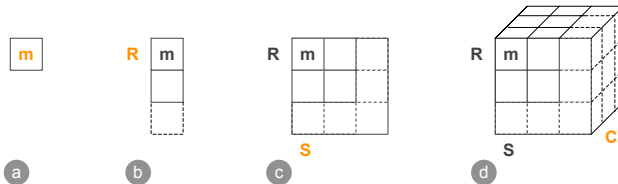


Fig. 2. Multi-dimensional Message Dispatch



**3D dispatch.** *Subjective programming* as introduced in [18] goes one step further than object-oriented programming in that it adds a third dimension to message dispatch. Here, method implementations are selected not only by their name and the receiver they are defined in, but also the sender the message send originated from ( $\langle m, R, S \rangle$ , Figure 2c).

**4D dispatch.** *COP* considers yet another dimension by dispatching not only on the name of a behavioral unit, the receiver it is defined in, and the sender the message originated from, but also on the context of this particular message send ( $\langle m, R, S, C \rangle$ , Figure 2d).

### 3 ContextS

ContextS is our COP extension to Squeak/Smalltalk to explore COP in late-bound class-based programming environments [2,15]. In Squeak/Smalltalk there are only objects, and messages exchanged between them. Since everything else is built on top of these concepts, and due to late-binding being used extensively throughout the system, the realization of ContextS was simple and straightforward. Only small changes to the language kernel needed to be made to achieve useful results. In this section we give a brief introduction to the small set of constructs provided with ContextS, leaving the illustration of their application to Section 4. We try to refrain from discussing implementation details, but will mention some alternatives we are currently investigating<sup>1</sup>.

#### 3.1 Implementation-Side Constructs

There are two main concepts to be used at the implementation side of concerns implemented in ContextS: Layers and advice-based partial method definitions.

*Layers* are simply represented as subclasses of `CsLayer`. In current versions of ContextS, layers are containers for partial method definitions. In future versions, we will move such method definitions away from the layers, into the classes they belong to. For a detailed discussion on why that should be done, please refer to our reading list, presented in Section 5.

```
CsLayer subclass: #MyLayer
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'My Category'
```

Partial *method* definitions, as shown here, still use an advice-based style as introduced in PILOT [19] and popularized by CLOS [4] and AspectJ-style language extensions [16]. The style presented here is inherited from AspectS [13], to take advantage of a set of metaobjects called *method wrappers* AspectS was built on [5].

<sup>1</sup> We use ContextS version 0.0.10 throughout the paper, available from <http://www.swa.hpi.uni-potsdam.de/cop/>.

```

MyLayer>>adviceCopLeafEvaluate
~ CsAroundVariation
  target: [MyTargetClass -> #myTargetSelector]
  aroundBlock: [:receiver :arguments :layer :client :clientMethod |
    "my layer-specific code"
    ...]

```

Here we can see that a partial method definition (`adviceCopLeafEvaluate`) belongs to a particular layer (`MyLayer`). The name of each such method definition needs to start with `advice` and has to have no arguments. These properties are exploited by the underlying framework to collect and compose all partial definitions associated with a layer. With `CsAroundVariation` we state that we will apply an *around* advice with class `MyTargetClass` and method `myTargetSelector` as its target. Layer-specific code provided by this partial method definition is stated in the *around* block and has full access to its environment, including the sender and the receiver of the message, its arguments, as well as the defining layer.

### 3.2 Activation-Side Constructs

At the client side of a concern implemented using `ContextS`, `useAsLayersFor:` is about the only construct ever used.

```
receiver useAsLayersFor: argument
```

`useAsLayersFor:` is a regular message that can be sent to collections or arrays that contain instances of `CsLayer`, or to code blocks that eventually return such collections or arrays.

All layers (instances of `CsLayer`) enumerated or computed by the receiver object are composed into the Squeak image in the order of their appearance in the list. This composition is only effective in the current process (Squeak's version of a thread) and for the dynamic extent of the execution of the block (an instance of `BlockContext` which is provided as the second argument).

## 4 Pretty-Printing as an Example

We use the task of pretty-printing an expression tree in infix, prefix, and postfix notation as well as an evaluation of the same as an example to show the differences between a regular object-oriented solution, an approach using the Visitor design pattern [11], and our context-oriented version.

The basic implementation of the nodes and leaves used to construct our expression trees is shown in Figure 3. A `CopNode` (left-hand side of Figure 3) has three instance variables for the first operand, the second operand, and the operation to combine the two. Each `CopLeaf` (right-hand side of Figure 3) has only one instance variable providing its value. Both classes provide accessor methods for their instance variables.

An expression tree can be assembled by the creation of individual instances of `CopNode` and `CopLeaf` and their combination. The example tree used throughout

```

Object subclass: #CopNode
  instanceVariableNames: 'first op second'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'ContextS-Demo Visitor'

Object subclass: #CopLeaf
  instanceVariableNames: 'value'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'ContextS-Demo Visitor'

CopNode class>>first: aFirstCopNodeOrCopLeaf
  op: aSymbol second: aSecondCopNodeOrCopLeaf
  ~ self new
  first: aFirstCopNodeOrCopLeaf;
  op: aSymbol;
  second: aSecondCopNodeOrCopLeaf

CopLeaf class>>value: anInteger
  ~ self new value: anInteger

CopNode>>first
  "^ <CopNode | CopLeaf>"
  ~ first

CopLeaf>>value
  "^ <Integer>"
  ~ value

CopNode>>first: anCopNodeOrCopLeaf
  first := anCopNodeOrCopLeaf.

CopLeaf>>value: anInteger
  value := anInteger.

CopNode>>op
  "^ <Symbol>"
  ~ op

CopNode>>op: aSymbol
  op := aSymbol.

CopNode>>second
  "^ <CopNode | CopLeaf>"
  ~ second

CopNode>>second: anCopNodeOrCopLeaf
  second := anCopNodeOrCopLeaf.

```

**Fig. 3.** Basic Node and Leaf Implementation

```

tree := CopNode
  first: (CopNode
    first: (CopNode
      first: (CopLeaf value: 1)
      op: #+
      second: (CopLeaf value: 2))
    op: #*
    second: (CopNode
      first: (CopLeaf value: 3)
      op: #-
      second: (CopLeaf value: 4)))
  op: #/
  second: (CopLeaf value: 5).

```

**Fig. 4.** Construction of an Expression

our paper is built in Figure 4. Figure 5 provides a graphical representation of the tree created in Figure 4.

## 4.1 Regular Objects

A simple and straightforward object-oriented implementation of pretty-printing is listed in Figure 6. The desired behavior is implemented *in-place* in both classes as methods `evaluate`, `printInfix`, `printPostfix`, and `printPrefix` respectively.

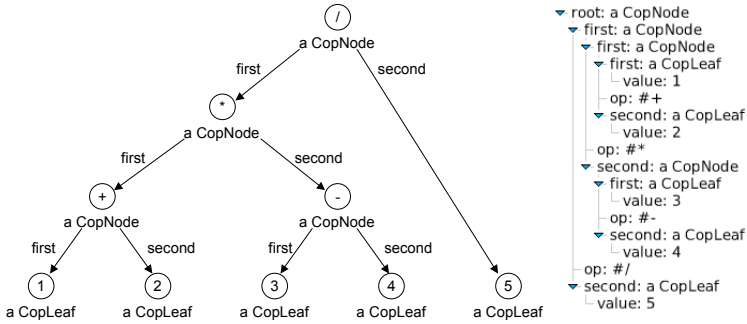


Fig. 5. Expression Tree

<pre> CopNode&gt;&gt;evaluate   ^ (self first evaluate)   perform: self op with: (self second evaluate)  CopNode&gt;&gt;printInfix   ^ '(' , self first printInfix,   self op, self second printInfix, ')'  CopNode&gt;&gt;printPostfix   ^ '(' , self first printPostfix,   self second printPostfix, self op, ')'  CopNode&gt;&gt;printPrefix   ^ '(' , self op,   self first printPrefix, self second printPrefix, ')'         </pre>	<pre> CopLeaf&gt;&gt;evaluate   ^ self value  CopLeaf&gt;&gt;printInfix   ^ self value asString  CopLeaf&gt;&gt;printPostfix   ^ self value asString  CopLeaf&gt;&gt;printPrefix   ^ self value asString         </pre>
--	---

Fig. 6. In-place Traversals

<pre> Transcript cr; show: tree printInfix. Transcript cr; show: tree printPrefix. Transcript cr; show: tree printPostfix. Transcript cr; show: tree evaluate.         </pre>	<pre> ==&gt; (((1+2)*(3-4))/5) ==&gt; ((/*(+12)(-34))5) ==&gt; (((12+)(34-)*5)/) ==&gt; (-3/5)         </pre>
---	---

Fig. 7. Use of In-place Traversals

Because these methods need to coexist side-by-side at the same time, they are named differently. And because of that, client side code needs to explicitly decide which one to use.

An application of our system so far is copied down in Figure 7, with the code executed on its left-, and the resulting print-outs on its right-hand side. In Squeak, objects are printed to the system console called Transcript by sending it the message show: with the object as argument.

## 4.2 Visitors

Our solution to the implementations of pretty-printing as presented previously is used as a motivation for the Visitor design pattern [11]. From its intent, we take that a visitor represents “an operation to be performed on the elements of an object structure” where the Visitor makes it easy to add new operations to

```

CopNode>>accept: aCopVisitor
~ aCopVisitor visitNode: self

CopLeaf>>accept: aCopVisitor
~ aCopVisitor visitLeaf: self

```

**Fig. 8.** Visitor-ready Base Objects

the entire structure. This is because all new operations can be defined outside this object structure it operates on, and so without the need to change it.

However, the resulting client code of Visitor-based systems is very hard to understand, with the manual simulation of double dispatch being one of the main reasons for that. Figure 8 lists the basic framework used for that in `CopNode` and `CopLeaf`, adding implementations of `accept:` to both classes that then call back to the argument, providing itself and the proper type information needed for further processing.

The actual implementation of two of the four visitors, `CopPrintPrefixVisitor` and `CopEvaluateVisitor`, can be seen in Figure 9. Here, our `visitLeaf:` and `visitNode:` methods control both local computation and follow-up traversals.

```

CopVisitor subclass: #CopPrintPrefixVisitor
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'ContextS-Demo Visitor'

CopPrintPrefixVisitor>>visitLeaf: aCopLeaf
~ aCopLeaf value asString

CopPrintPrefixVisitor>>visitNode: aCopNode
~ '(,
  aCopNode op,
  (aCopNode first accept: self),
  (aCopNode second accept: self),
)',

CopVisitor subclass: #CopEvaluateVisitor
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'ContextS-Demo Visitor'

CopEvaluateVisitor>>visitLeaf: aCopLeaf
~ aCopLeaf value

CopEvaluateVisitor>>visitNode: aCopNode
~ (aCopNode first accept: self)
perform: aCopNode op
with: (aCopNode second accept: self)

```

**Fig. 9.** Visitor Examples

```

Transcript cr; show: (tree accept: CopPrintInfixVisitor new).      ==> (((1+2)*(3-4))/5)
Transcript cr; show: (tree accept: CopPrintPrefixVisitor new).   ==> (/(*+12)(-34))5)
Transcript cr; show: (tree accept: CopPrintPostfixVisitor new).  ==> (((12+)(34-)*5)/)
Transcript cr; show: (tree accept: CopEvaluateVisitor new).      ==> (-3/5)

```

**Fig. 10.** Use of Visitors

An application of our Visitor-based system so far is transcribed in Figure 10, again with the code executed on its left, and the resulting print-outs on its right.

### 4.3 Layers

Now we are going to implement our expression traversal example using ContextS by applying the constructs introduced in Section 3. Here, we present two of our four layers, `CopPrintInfixLayer` and `CopEvaluateLayer`, in Figure 11. Each layer is a subclass of `CsLayer`, provides two partial method definitions with class `CopLeaf`, class `CopNode`, and their methods `printOn:` as targets. As

```

CsLayer subclass: #CopPrintPrefixLayer
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'ContextS-Demo Visitor'

CsLayer subclass: #CopEvaluateLayer
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'ContextS-Demo Visitor'

CopPrintPrefixLayer>>adviseCopLeafPrintOn
~ CsAroundVariation
target: [CopLeaf -> #printOn:]
aroundBlock: [:receiver :arguments
:composition :client :clientMethod |
receiver value printOn: arguments first]

CopEvaluateLayer>>adviseCopLeafEvaluate
~ CsAroundVariation
target: [CopLeaf -> #evaluate]
aroundBlock: [:receiver :arguments
:composition :client :clientMethod |
receiver value]

CopPrintPrefixLayer>>adviseCopNodePrintOn
| stream |
~ CsAroundVariation
target: [CopNode -> #printOn:]
aroundBlock: [:receiver :arguments
:layer :client :clientMethod |
stream := arguments first.
stream nextPut: $(.
stream nextPutAll: receiver op.
receiver first printOn: stream.
receiver second printOn: stream.
stream nextPut: $)]

CopEvaluateLayer>>adviseCopNodeEvaluate
~ CsAroundVariation
target: [CopNode -> #evaluate]
aroundBlock: [:receiver :arguments
:layer :client :clientMethod |
receiver first evaluate
perform: receiver op
with: receiver second evaluate]

```

Fig. 11. Layered Traversal Code

```

[ { CopPrintInfixLayer new } ] useAsLayersFor: [
Transcript cr; show: tree].                ==> (((1+2)*(3-4))/5)

[ { CopPrintPrefixLayer new } ] useAsLayersFor: [
Transcript cr; show: tree].                ==> (/(*(+12)(-34))5)

[ { CopPrintPostfixLayer new } ] useAsLayersFor: [
Transcript cr; show: tree].                ==> (((12+)(34-)*5)/)

[ { CopEvaluateLayer new } ] useAsLayersFor: [
Transcript cr; show: tree evaluate].        ==> (-3/5)

```

Fig. 12. Use of Layered Traversal Code

already stated previously, objects are printed to the `Transcript` by sending it the message `show:` with the object as argument. `show:` itself sends `printOn:` to the object, with a stream as its argument. This is the method we would override in a subclass to change the behavior of `show:`, and this is also the method we adapt using COP-style refinements.

In this example, our context-dependent behavior simply overrides the original behavior present before its layer activation by using an `around` construct without a `proceed`. More method combinations including `around` with `proceed`, `before`, or `after` semantics can be achieved as well, but are beyond the scope of this tutorial.

In an upcoming version of `ContextS`, the need for AOP-style inverse relationships will be reduced, since our traversal code belongs to the objects and should be defined in the scope of their classes so that programmers reading their code are aware of its impact (Figure 13).

The activation-side of our context-dependent behavior looks as simple and straightforward as promised (Figure 12): The programmer of the client code

states or computes the desired layer combination (only one layer in our example), and uses it via the `useAsLayersFor:` message. This causes the layer activation to be composed into the system and visible in the dynamic extent of the execution of the provided block argument.

Please note that the provided block arguments are the same in all four cases. The method `show:` is used all the time to print out our expression tree. Only the layer used is different from case to case. It is also important to point out that in the code block provided to `useAsLayersFor:` the message `show:` is sent to `Transcript` whereas our layers adapt `printOn:` of `CopLeaf` and `CopNode` respectively.

## 5 Further Reading

Related work of COP including AOP, FOP, or delegation have been presented and discussed in other publications. In the following we list some of them for further reading:

*Language Constructs for Context-oriented Programming – An Overview of ContextL* [6]. That paper presents ContextL, our first COP extension. It supplements the Common Lisp Object System (CLOS) with layers, layered classes, layered and special slots, layered accessors, and layered functions. In ContextL, layered classes, slots and functions can be accumulated in layers. ContextL's layers or their combinations are dynamically scoped, allowing us to associate partial behavioral variations with the classes they belong to, while, at the same time, changing their specific behavior depending on the context of their use.

*Efficient Layer Activation for Switching Context-dependent Behavior* [8]. In that paper, we illustrate how ContextL constructs can be implemented efficiently. As an interesting result, ContextL programs using repeated layer activations and deactivations are about as efficient as without, underlining the fact that apparently other things are more important regarding performance and its optimization. We also show an elegant and efficient implementation of the prominent AOSD figure editor example, even without the need to resort to cflow-style constructs in the first place.

*Reflective Layer Activation in ContextL* [7]. That paper describes a reflective approach to the expression of complex, application-specific layer dependencies, without compromising efficiency.

*Context-oriented Programming* [14]. In that contribution, we summarize our previous work and present COP as a novel approach to the modularization of context-dependent behavior. We show that, by treating context explicitly and by providing dedicated abstractions and mechanisms to represent context-dependent variations, programs can be expressed more concisely than without. Several examples are provided to illustrate COP's advantages over more traditional approaches.

```

CopNode>>printOn: aStream
  <<layer: PrintPrefix>>
  ^ '(,
    self op,
    self first printPrefix,
    self second printPrefix,
  )'

CopLeaf>>printOn: aStream
  <<layer: PrintPrefix>>
  ^ self value asString

CopNode>>printOn: aStream
  <<layer: Evaluate>>
  ^ self first evaluate
    perform: self op
    with: (self second evaluate)

CopLeaf>>printOn: aStream
  <<layer: Evaluate>>
  ^ self value

```

Fig. 13. Another Representation of Layered Traversal Code

## 6 Summary and Outlook

In our tutorial-style introduction to ContextS we have presented both activation-side and implementation-side constructs of our COP extension to Squeak/Smalltalk. We provided sample implementations illustrating three different approaches to printing out an expression tree: plain and straightforward object-oriented programming, a Visitor-based application, as well as a COP-based solution using ContextS.

Our intent was not to discuss advantages of COP-style development but rather to present some of the mechanics involved in working with ContextS. One of the issues we are working on right now is to allow for specifying context-specific code outside of layers and inside the classes it belongs to. Figure 13 presents one of our approaches to this issue. Here we can see different versions of the `printOn:` method, associated with one and the same class at the same time. The difference is the layer expressed via `<<layer: ...>>` denoting the layer the particular method belongs to.

While ContextS provides means to express heterogeneous crosscutting behavioral variations, we look into its extension to concisely implement homogeneous crosscutting concerns as well [1].

We are currently working on medium- to large-sized examples and case studies to illustrate to what extent COP helps to better express software entities that need to adapt their behavior to their context of use, and to further refine our language extensions.

## Acknowledgements

We thank Marcus Denker, Tudor Girba, Robert Krahn, Dave Thomas, and Jan Wloka for their fruitful discussions and valuable contributions, and Ralf Lämmel for his patience.

## References

1. Apel, S.: The Role of Features and Aspects in Software Development. PhD thesis, Otto-von-Guericke University Magdeburg (March 2007)
2. Batory, D.: Feature-oriented programming and the ahead tool suite. In: Proceedings of the International Conference on Software Engineering 2004 (2004)



3. Batory, D., Rauschmeyer, A.: Scaling step-wise refinement. *IEEE Transactions on Software Engineering* (June 2004)
4. Daniel, G., Bobrow, L.G., De Michiel, R.P., Gabriel, S.E.: Common lisp object system specification: 1. programmer interface concepts. *Lisp and Symbolic Computation* 1(3-4), 245–298 (1989)
5. Brant, J., Foote, B., Johnson, R.E., Roberts, D.: Wrappers to the rescue. In: Jul, E. (ed.) *ECOOP 1998*. LNCS, vol. 1445, pp. 396–417. Springer, Heidelberg (1998)
6. Costanza, P., Hirschfeld, R.: Language constructs for context-oriented programming — an overview of ContextL. In: Wuyts, R. (ed.) *Proceedings of the 2005 Dynamic Languages Symposium*, ACM Press, New York (2005)
7. Costanza, P., Hirschfeld, R.: Reflective layer activation in ContextL. In: *Proceedings of the Programming for Separation of Concerns (PSC) of the ACM Symposium on Applied Computing (SAC)*. LNCS. Springer, Heidelberg (2007)
8. Costanza, P., Hirschfeld, R., De Meuter, W.: Efficient layer activation for switching context-dependent behavior. In: Lightfoot, D.E., Szyperski, C.A. (eds.) *JMLC 2006*. LNCS, vol. 4228. Springer, Heidelberg (2006)
9. Filman, R.E., Elrad, T., Clarke, S., Akşit, M. (eds.): *Aspect-Oriented Software Development*. Addison-Wesley, Boston (2005)
10. Filman, R.E., Friedman, D.P.: Aspect-oriented programming is quantification and obliviousness. In Filman et al., [9], pp. 21–35.
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns*. Addison-Wesley, Reading (1995)
12. Goldberg, A., Robson, D.: *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston (1983)
13. Hirschfeld, R.: AspectS – aspect-oriented programming with Squeak. In: Akşit, M., Mezini, M., Unland, R. (eds.) *NODe 2002*. LNCS, vol. 2591, pp. 216–232. Springer, Heidelberg (2003)
14. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. *Journal of Object Technology (JOT)* 7(3), 125–151 (2008)
15. Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A.: Back to the future: the story of squeak, a practical smalltalk written in itself. In: *OOPSLA 1997: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 318–326. ACM Press, New York (1997)
16. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) *ECOOP 1997*. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
17. Nordberg III, M.E.: Aspect-oriented dependency management. In: Filman et al., [9], pp. 557–584
18. Smith, R.B., Ungar, D.: A simple and unifying approach to subjective objects. *TAPOS special issue on Subjectivity in Object-Oriented Systems* 2(3), 161–178 (1996)
19. Teitelman, W.: *Pilot: A step towards man-computer symbiosis*. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA (1966)

# A Landscape of Bidirectional Model Transformations

Perdita Stevens

Laboratory for Foundations of Computer Science  
School of Informatics  
University of Edinburgh  
Fax: +44 131 667 7209  
perdita@inf.ed.ac.uk

**Abstract.** Model transformations are a key element in the OMG's Model Driven Development agenda. They did not begin here: the fundamental idea of transforming, automatically, one model into another is at least as old as the computer, provided that we take a sufficiently broad view of what a model is. In many contexts, people have encountered the need for *bidirectional* transformations. In this survey paper we discuss the various notions of bidirectional transformation, and their motivation from the needs of software engineering. We discuss the state of the art in work targeted specifically at the OMG's MDD initiative, and also, briefly, related work from other communities. We point out some areas which are so far relatively under-researched, and propose research topics for the future.

**Keywords:** bidirectional model transformation, QVT, graph transformation, triple graph grammar, bidirectional programming language.

## 1 Introduction

Bidirectional model transformations are both new and old. New in that they have recently come to prominence in the context of the Object Management Group's Model Driven Architecture initiative. Old in the sense that the problems which arise in considering them turn out to be similar to problems that have been studied in other contexts for decades.

In this paper, we explore the landscape of bidirectional model transformations as it appears to the author at the time of writing, autumn 2007 to spring 2008. We attempt to summarise the perceived need for bidirectional model transformations, drawing out the various different assumptions that may be made about the environment in which the transformation operates, since this has important effects on the technology and underlying theory available. We discuss the state of the art in model transformations in the OMG sense, and then move on to survey related work in other fields. Finally, we mention some areas which have been relatively under-studied to date, and suggest some directions for further research.

## 2 Background: Why Bidirectional Model Transformations?

A model, for purposes of this paper, is any artefact relating to a software system. For example, its code or any part of it; UML models for it, or for a product-line of which it is a member; its database schema; its test set; its configuration management log; its formal specification, if any. We will be mostly concerned with formal artefacts and will therefore not usually consider, for example, the natural language document that describes the systems architecture, or the physical computer on which the system will run, to be models; however, it is surprisingly difficult to draw a hard-and-fast line between formal and informal artefacts.

Model transformations aim to make software development and maintenance more efficient by automating routine aspects of the process. For example, a common scenario is that two models have to be consistent in some sense, and that it is possible to codify the process of restoring consistency when one model is changed. In such a case, it is helpful if this notion of consistency and restoration process can be captured in a model transformation. In the simplest case, only one of the two models (the *source*) is ever modified by humans: the other (the *target*) is simply generated by a tool. The tool is deterministic, or at least, if it is possible for it to produce several different targets from the same source, the various targets are seen as having equal value, so that the difference between them is immaterial. In such a case, in practice we will never check consistency: if there is any doubt that the target was produced from the source, the transformation will be re-run. The old target model is discarded and replaced by the newly-generated version. For example, this is typically the case when a program in a high-level language is compiled. If the sets of models are  $M$  and  $N$ , a unidirectional transformation of this kind can be modelled as  $f : M \rightarrow N$ ; in the simplest case, the consistency relation will be simply that  $m$  and  $n$  are consistent iff  $n = f(m)$ .

More interesting, more difficult and arguably more typical is the situation where both models may be modified by humans. Examples include:

1. The classic MDA situation: a platform independent model (PIM) is transformed into a platform specific model (PSM).
2. The database view problem, in which a user is presented with a view of a selected subset of the data from the database, and may modify the view: at the same time, the full database may change in the usual way.
3. Many kinds of integration between systems or parts of systems, which are modelled separately but must be consistent: for example, a database schema must be kept consistent with the application that uses it.

Even if the consistency restoration process has to be manual – for example, because there may be several different ways to restore consistency, with different values, and nobody can formalise rules which capture which way is best – it may still be worthwhile to have an automatic check of whether the models are consistent.

A *bidirectional model transformation* is some way of specifying algorithmically how consistency should be restored, which will (at least under some circumstances) be able to modify either of the two models. Within this broad definition, there are many variants which we shall explore.

Even when transformation technologies are first thought of as unidirectional, it often turns out that bidirectionality is required. In [32], it is interesting to note that the ability to write bidirectional transformations appears high in the list of users' priorities, even though the original call for proposals for the OMG's language for expressing Queries, Views and Transformations, QVT [28] listed bidirectionality as an optional feature.

## 2.1 Related Work

Consistency management has as mentioned a long history, which we will not go into here. In the field of models, an interesting early paper focusing on pragmatic issues that arise in round-trip engineering – connecting a model with source code – is Sendall and Küster's position paper [29]. Czarnecki and Helsen in [8] present a survey of model transformation techniques with a particular emphasis on rule-based approaches such as those based on graph transformations. They mention directionality, but do not focus on it. Similarly Mens and Van Gorp in [24] discuss the wider field of model transformations, not necessarily bidirectional.

The present author in [30] discussed bidirectional transformations in the specific context of the QVT Relations (hereinafter QVT-R) language. The paper proposed a framework for bidirectional transformations in which a bidirectional transformation is given by three elements, flexible enough to describe any of the approaches considered here. We write  $R(m, n)$  if and only if the pair of models is deemed to be consistent. Associated with each relation  $R$  will be the two directional transformations:

$$\vec{R} : M \times N \longrightarrow N$$

$$\overleftarrow{R} : M \times N \longrightarrow M$$

The idea is that  $\vec{R}$  looks at a pair of models  $(m, n)$  and works out how to modify  $n$  so as to enforce the relation  $R$ : it returns the modified version. Similarly,  $\overleftarrow{R}$  propagates changes in the opposite direction. The paper proposed postulates to ensure that the transformation should be coherent.

## 2.2 Terminology

In a truly symmetric situation, there is an arbitrary choice about which model to call the “source” and which the “target”: nevertheless, the terminology is widely used and useful, and there is usually some asymmetry in the situation. Generally, the “source” model is the one which is created first, and it is more probable that a change to the source model will necessitate a change to the target model than the other way round.

Note that this paper does not attempt to discuss extra problems that may arise when checking or restoring consistency between more than two models – multidirectional transformations.

### 3 Important Differences in Circumstances

Discussions of bidirectional transformations often turn out to hinge on what assumptions are made about the situation: a solution which seems good with one set of assumptions can be infeasible with another. In this section, we draw out some of the most important differences in circumstances of use of a transformation. These differences are among the chosen design assumptions of the developers of a bidirectional transformation approach; looked at from the point of view of users who need to choose an appropriate approach for their circumstances, they are factors that may influence that choice. Our aim here is not to classify existing approaches: indeed, in this section we will discuss several issues where all the approaches we will later cover are in accord. Such issues are not useful for distinguishing between existing approaches, but they may be useful in pointing out lacunae in the current research. This is our main aim.

In later sections, we will discuss approaches to bidirectional transformations, trying to make clear how they fit into the framework of differences presented in this section. However, we do not claim to have exhaustively addressed every issue for every tool. Where there is a majority approach, we say so in this section, and point out exceptions later.

#### 3.1 Is There an Explicit Notion of Two Models Being Consistent?

Situations where bidirectional transformations may be applied always involve at least an informal notion of consistency: two models are consistent if it is acceptable to all their stakeholders to proceed with these models, without modifying either. Since we have said that the job of the bidirectional transformation is to restore consistency, we may alternatively say that two models are consistent if no harm would result from not running the transformation. Note that this is not quite the same as to say that running the transformation would have no effect: some transformation approaches may modify a pair of models, even if they are already consistent.

Typically, bidirectional transformation languages described as “relational”, such as QVT-R and BOTL (to be discussed), make their notion of consistency explicit: part of what they do is to define a consistency relation on two sets of models. Many other presentations of bidirectional transformations do not make their notion of consistency explicit. In such a case, any situation which may arise from a (successful) application of the transformation may be considered consistent by definition. However, it can be hard to characterise this other than by describing exactly what the transformation tool will do.

### 3.2 Does One Expression Represent Both Transformation Directions?

Superficially, the easiest way to write a bidirectional transformation is to write it (in text or in diagrams) as a pair of functions, one in either direction, maybe also giving an explicit consistency relation. This avoids the need to write special purpose languages for bidirectional transformations, enabling instead the reuse of established programming languages. However, even if we make explicit constraints on the two functions which suffice to make the different expressions coherent, we have an obvious danger: that the coherence of the different expressions will not be maintained as time passes.

Notice that the concern here is how actual transformations are written down. There is no objection to modelling a transformation as a pair of functions, if both functions are represented by one expression (textual or graphical).

Perhaps, if a framework existed in which it were possible to write the directions of a transformation separately and then check, easily, that they were coherent, we might be able to have the best of both worlds. However, no such framework exists today.

Since this paper is focused on work which aims specifically at addressing bidirectional transformations, it is unsurprising that all the approaches to be discussed involve writing a single text (or equivalently, a single set of diagrams).

### 3.3 Is the Transformation Bijective, Surjective or Neither?

This is probably the most important assumption to clarify when looking at a particular tool or technology: however, this is not always easy if the approach does not have an explicit notion of consistency (because one may have to study the syntax of the language to understand which implicit consistency relations can be defined).

Given a consistency relation on the pairs of models, we call a transformation bijective if the consistency relation is a bijection: that is, for any model (source, *rsp.* target) there exists a unique corresponding model (target, *rsp.* source) which is consistent with it. This is a very strong condition: in effect, it says that the two models contain identical information, presented differently. In particular, it is impossible to define a bijective bidirectional transformation between two sets of models which have different cardinalities.

If the reader has in mind an approach whose notion of consistency is implicit, this may not be quite obvious. Consider, for example, a transformation which defines (in any way) a function

$$f : M \longrightarrow N$$

with the implicit understanding that  $m$  and  $n$  are consistent iff  $n = f(m)$ . This consistency relation is a bijective relation if and only if  $f$  is a bijective function, and in that case there is an inverse function

$$f^{-1} : N \longrightarrow M$$

satisfying the usual identities. This situation cannot arise if, to give a trivial example, completely defining  $m \in M$  requires three boolean values while completely defining  $n \in N$  requires three integer values.<sup>1</sup>

Thus, an approach that permits only bijective transformations is in most practical situations much too restrictive. A more realistic, but still quite restrictive, condition is that one of the models should be a strict abstraction of the other: that is, one model contains a superset of the information contained in the other.

### 3.4 Must the Source Model Be Modified When the Target Changes?

This issue does not strictly concern bidirectional transformations, but it does concern transformations which do not fit into the unidirectional transformation framework ( $f : M \rightarrow N$ ) initially presented.

Suppose we have a source and target model which are consistent, perhaps because the target was produced from the source by compilation. Suppose further that changes may be made to the target model, but they are small in the sense that they will not cause the target to become inconsistent with the source. Two examples are:

1. Optimising the code produced by a compiler (using a suitably restrictive class of optimisations)
2. Taking skeleton code generated from a UML class diagram, and adding method bodies.

(Notice that the allowable changes to the target depend crucially on the notion of consistency chosen.)

In this restricted case, changes to the target model never require changes to the source model. The reason why this does not fit into the straightforward unidirectional transformation framework presented above is that it is not acceptable to discard changes to the target, when the source model changes and the target needs to be updated to reflect those changes. This kind of transformation might be represented as a function

$$f : M \times N \rightarrow N$$

assuming that a default value from  $N$ , representing “no interesting information” is available for use in the initial generation.

All the approaches discussed here assume that the source model may have to be modified when the target changes.

### 3.5 Must the Transformation Be Fully Automatic, or Interactive?

If there may be a choice about how to restore consistency, there are two options: either the transformation programmer must specify the choice to be made, or

---

<sup>1</sup> Of course, which model sets actually have the same cardinality depends crucially on whether we define our models using machine integers or mathematical integers, etc.; but either way, not all model sets have the same cardinality.

the tool must interact with a user. The ultimate interactive bidirectional transformation would consist simply of a consistency checker: the tool would report inconsistencies and it would be up to the user to restore consistency. Intermediate scenarios, in which the programmer specifies how most kinds of inconsistency can be dealt with, but exceptional cases are referred to the user, might be imagined. Another variant, [7], is discussed later.

Any approach which involves a tool behaving non-deterministically has an obvious interactive counterpart in which the user resolves the non-determinism; we will mention one such case. However, none of the work discussed here explicitly addresses interaction with the user.

### 3.6 Is the Application of the Transformation under User Control?

A slightly different question is whether the user of the transformation tool controls when consistency is restored, or whether this is done automatically without user involvement. Practically, applying the transformation automatically entails that the transformation should be fully automatic, but not vice versa. There are advantages to automatically applied transformations (compare automatic recompilation of changed source files in Eclipse and similar development environments, or the demand for “pushed” changes to data e.g. web pages): the user does not have to remember to apply the transformation, and the danger of work being wasted because it is done on an outdated model is reduced. However, the fact that users’ actions have effects which are invisible to them may also be confusing, e.g. if the transformation is not undoable in the sense of [30]. Intermediate situations such as transformations being applied every night, or on version commit, may sometimes be good compromises.

### 3.7 Is There a Shared Tool?

If the people modifying the two models use the same model transformation tool, then the tool may perhaps keep and make use of information other than the models themselves. For example, it may keep what is called “trace” information in QVT or “correspondence” information in triple graph grammars: a persistent record of which elements of one model are related to a given element in the other. This can simplify the programming of model transformations, in that provided the notion of corresponding elements can be somehow established, it can be used throughout the transformation definition. If this extra information were itself extractable in a standard format which could be read by several tools, then of course the same could be done even without a literally shared tool, at the expense of giving the user another artefact to manage.

### 3.8 Is It Permissible to Cache Extra Information in the Models?

As discussed above, there may be information in each model which is not contained in the other. One approach is to pick one of the models and modify it so that it contains all of the information from both models, in a form which is not



evident to the user of the model. For example, if we consider a transformation between a UML model and the code that implements it, the method bodies may be hidden in the model, and/or the diagrammatic layout information may be encoded in comments in the generated source files.

By this means, the bidirectional transformation can be made surjective, or even bijective, while leaving the models looking the same to their users. In effect, we are adding extra transformations: instead of a general bidirectional transformation

$$M \longleftrightarrow N$$

we have

$$M \hookrightarrow M' \longleftrightarrow N' \hookleftarrow N$$

where the outer transformations are implicit, being invisible to the user, and it can be arranged that the inner transformation is bijective.

This is a very useful and pragmatic approach widely used in, for example, generation of code from UML models. It carries the same kind of problems as any other kind of non-user-visible transformation, however. There is a possibility that the user, not understanding the extra information, may accidentally modify it, for example. Moreover, this approach only works if the transformation tool has complete control over the models. If a different tool is also being used to modify one of the models, information may be lost.

### 3.9 What Must the Scope of the Model Sets Be?

Much of the work to be discussed works with XML; some with relational database schemas; some with MOF metamodels. Some problems are more tractable if the language of models can be restricted. For example, [2] discusses round-trip engineering of models in the context of a framework-specific language. If the models are trees, e.g. XML documents, are idrefs and similar ways to turn a tree into a graph permitted? If lists occur, is order important? If the models are graphs, are attributes permitted, and with what restrictions?

## 4 The QVT Relations Language and Tools

QVT-R is the relational language which forms part of the OMG's Queries, Views and Transformations standard [28]. A single text describes the consistency relation and also the transformations in both directions. [30] discusses the use of the language for bidirectional transformations, pointing out in particular that there is some ambiguity about whether the language is supposed to be able to describe transformations which are not bijective.

Despite the heavy emphasis placed on model transformation by the OMG's model-driven development strategy, and the clear importance of bidirectionality to users, tool support for bidirectional transformations expressed in QVT-R remains limited. However, IKV++'s Medini QVT<sup>2</sup> incorporates an open-source

<sup>2</sup> <http://projects.ikv.de/qvt>, last accessed March 13th 2008.

QVT-R engine; TCS's ModelMorf tool<sup>3</sup> is also available, but is still in “pre beta” release, with no clear sign of a forthcoming true release. QVT-R is not yet supported in the Eclipse M2M project, though such support is planned.

## 5 Approaches Using MOF but Outside QVT

ATL, the ATLAS Transformation Language, is a widely used transformation language often described as “QVT-like”, which has good tool support. (The term ATL gets used both for the entire architecture and for the middle abstraction layer.). It is a hybrid language, encouraging declarative programming of transformations but with imperative features available. Its most abstract layer, the ATLAS Model Weaving language (AMW) [10] <sup>4</sup> allows the specification of a consistency relation between sets of models, which as we have discussed is an important aspect of bidirectional transformations. AMW does not, however, provide general capabilities for the programmer to choose how inconsistencies should be resolved; in ATL, a bidirectional transformation must be written as a pair of unidirectional transformations [17]. In passing, let us remark that while [10] distinguishes model weaving from model transformation, the terminological distinction of that paper does not seem to be in widespread use: both concepts, as described there, are within “model transformation” as covered in this paper, and in fact the term “model weaving” is used in other senses by other authors.

MOFLON [1] provides, among other things, the ability to specify transformations using triple graph grammars (discussed below).

BOTL [5], the Basic (or elsewhere, Bi-directional) object-oriented transformation language, builds from first principles a relational approach to transformation of models conforming to metamodels in a simple sublanguage of MOF. Although it discusses the point that relations may not be bijective, when considering transformations it only addresses bijective relations: the language does not provide the means to specify how consistency should be restored if there is a choice.

## 6 XML-Based Options

A number of approaches to bidirectional transformations between XML documents (or between an XML document and another kind of model) have arisen in the programming languages community: programming languages, possibly with sophisticated type systems, have been developed for writing single programs which can be read as forward or as backward transformations. Unlike the graph transformation approach, a program is conceived as a whole, with a defined order of execution and some degree of statically checkable correctness.

Of these, biXid [18] is the most similar to QVT-R and graph grammar work, as it adopts a relational approach to bidirectional transformations between pairs of

<sup>3</sup> See <http://www.tcs-trddc.com/ModelMorf/index.htm>, last accessed March 13th 2008.

<sup>4</sup> See also <http://www.eclipse.org/gmt/amw/>, last accessed March 13th 2008.

XML languages. It allows non-bijective relations (ambiguity, in the terminology of the paper) and discusses the implications of this for the transformation engine; however, a consistent model is chosen non-deterministically from among the possibilities. There is deliberately no way for the programmer to specify how consistency should be restored when there is a choice. This makes sense in the context addressed, which is where different formats for the same information must be synchronised: the non-bijectivity is expected to come from the existence of different but semantically equivalent orderings of the same information, not from the presence of different information which must be retained. It also makes a point of the design decision that a variable on one side of a relational rule may occur several times on the other side (non-linearity), which is important for permitting the restructuring of information.

The Harmony project [3,12] defines a language called Boomerang (building on earlier work on a language called Focal) in which bidirectional programs – termed lenses – are built up from a deliberately limited set of primitive lenses and combinators. Compositionality is thus a major concern of the approach, in contrast to the relational approaches. Transformations are always between a “concrete” set of models and a strict abstraction of this, the “abstract” set of models which contains strictly less information: hence, there is an explicit notion of consistency in which two models are consistent exactly when one is an abstraction of the other. The transformation programmer has control of how consistency should be restored where there is a choice (which only happens when the abstract model has been modified and the concrete model needs to be changed).

Hu, Mu and Takeichi [16,25,26] define a language called Inv, a language defined in “point free” style (that is, without variable names: this has obvious advantages for bidirectional languages, as also argued by [27], but can be a readability barrier) in which only injective functions can be defined. In order to capture useful behaviour, the codomain is enriched with a history part, thus making an arbitrary function injective. In model transformation terms, this approach modifies the metamodel in order to be able to define transformations. The language is targeted at the authors’ Programmable Structured Document project: [16] develops an editor for structured documents, in which the user edits a view of a full document, and these edits induce edits on the transformation between the full document and the view. This is different from the more usual bidirectional transformation scenario in which the two models are disconnected while they are being edited: that is, the work assumes a shared tool, although note that it does *not* assume that the transformation is automatically applied. A view in this sense may duplicate information, retaining dependencies between the copies – the transformation may be non-linear – but it is still the case that a document uniquely determines a view, that is, the transformation is surjective.

XRound [6] is a template-based language for bidirectional transformations. Template-based transformation languages have been widely used for unidirectional transformations such as code generation from models. In that application, the transformation gives a boiler-plate piece of code containing metavariables. The metavariables are replaced by values obtained by interrogating the input

model, to give the output code. The novelty in XRound is to use a similar technique bidirectionally by using unification between variables of the target application (e.g., from the output code) and the results of queries on the input model. Thus XRound provides a bidirectional, but asymmetric, language of transformations particularly adapted to cases where an XML document must be synchronised with a simpler structure associated with another tool. The original application was security analysis of models presented as XMI files.

Finally, XSugar [4], which, being aimed at transforming between two syntaxes for the same language, naturally insists that transformations should be bijective.

It is tempting to think that the work on transformations of XML documents should be immediately applicable to models in MOF-based languages, since these can be represented by XMI files. However, there are significant obstacles not yet overcome. Although XML files represent trees, models typically make essential use of `xmi:ids` or similar mechanisms to turn the trees into general graphs. None of the existing XML transformation languages work well in this context ([16], for example, explicitly forbids `IDRef` use, though it is not clear that it forbids the use of other information for the same purpose.)

## 7 Graph Transformations

In the context of model transformations, almost all formal work on bidirectional transformations is based on graph grammars, especially triple graph grammars (TGGs) as introduced by Schürr (see, for example, [20]). Two graphs represent the source and target models, and a third graph, the correspondence graph, records information about the matches between nodes in the source and target. Thus TGGs are an approach which relies on developers of the source and target model using a shared tool which can retain this information, or at least on its being retained somehow, possibly by modification of one of the two meta-models. Each rule has a top, or precondition, part which specifies when the rule applies: that is, what triples consisting of a subgraph of each of the three graphs should be examined. The bottom part of the rule then specifies consistency by describing what else should be true of each of the three graphs in such a case. The transformation programmer controls how consistency should be restored by marking in the bottom part of the rule which nodes should be created if no such node exists in its graph. Thus, it is possible to use TGGs to specify bidirectional transformations with non-bijective consistency relations. However, extending this to programming model transformations introduces new issues, such as how to describe how to restore any consistency relations that are specified between attributes that are not modelled as part of the graphs.

For a discussion of the use of TGGs for model transformation, see [14]. Indeed, the definition of the QVT core language was clearly influenced by TGGs. Greenyer and Kindler have given an informative comparison between QVT core and triple graph grammars in [13]. In the process of developing a translation of (a simplified version of) QVT core to a variant of TGGs that can be input into a TGG tool, they point out close similarities between the formalisms and discuss some semantic issues relating to bidirectionality.

The main tool support for TGGs is FUJABA [8], which provided the foundation for MOFLON mentioned above. FUJABA (like other tools) provides the means for rules to be given different *priorities* in order to resolve conflicts where more than one rule applies. This can improve the performance of a transformation engine, but it can also be another means for the programmer to specify how consistency is restored when there is a choice. Anecdotally, it can be hard to understand the implications of choices expressed in this way and hence to maintain the transformation.

More broadly, the field of model transformations using graph transformation is very active, with several groups working and tools implemented. We mention in particular [21,31]. Most recently, the paper [9] by Ehrig et al. addresses questions about the circumstances in which a set of TGG rules can indeed be used for forward and backward transformations which are information preserving in a certain technical sense.

## 7.1 Miscellaneous

Meertens [23] writes about bidirectional transformations – “constraint maintainers” – in a very basic and general setting of sets and relations. His maintainers are given as three components, a relation and a transformation in each direction. He discusses the conditions which relate them, with particular concern for minimising the “edit distance” involved in restoring consistency.

Finally, [7] is unusual in exploring the effects of using interactivity of a tool to loosen the constraints imposed by the metamodels. If a target model is generated from a source by a model transformation, and then manually modified, possibly in such a way that it no longer conforms to the target metamodel, how should consistency be restored between this modified target model and the original source? The authors propose using Answer Set Programming to produce a set of possible source models which, on application of the original forward transformation, give target models close to the manually modified one. The user then chooses from among these “approximations”.

## 8 Research Directions

At the time of writing, autumn 2007, there is still a wide gap between the vision of model transformations as first-class artefacts in the software engineering of systems, and the reality of the techniques and tools available. In this relatively immature field there is obviously a need for further work in many directions. In this section, we focus on some areas which seem to have been relatively under studied, but which will be important in practice.

### 8.1 Compositionality

To get beyond toy examples, we need clean mechanisms to compose model transformations in (at least) two senses. We need good semantic understanding of the

<sup>5</sup> <http://www.fujaba.de>, last accessed March 13th 2008.

consequences of composing transformations, and also good engineering understanding and tool support to make it practical to do so, even when the transformations originate from different groups. Both of these senses sound fairly unproblematic, but in fact, problems can arise: see [30] for more discussion.

*Spatial composition.* If two systems are composed of parts (such as components or subsystems) which themselves can be acted on by model transformations, we will need to be able to compose those part transformations to give a transformation of the whole systems. We will need to be able to understand the effect of the composed transformation by understanding the parts. Ideally, this would apply even if the system parts were not themselves encapsulated parts of the systems: for example, given transformations of aspects of systems, we would like to be able to construct transformations of the woven systems.

*Sequential composition.* Given unidirectional transformations  $f : A \rightarrow B$  and  $g : B \rightarrow C$ , we can obviously compose them to yield  $gf : A \rightarrow C$ . This extends to bidirectional transformations if they are bijective. In the general case, however, composition does not work because of the need to “guess” matching elements in the elided middle model space. Part of the attraction of allowing bidirectional transformations to relate models to their strict abstractions is that it permits a reasonable notion of composition: see [30], and also [23] for discussion.

## 8.2 Specification

Specifications of programs are (ideally) useful in a variety of ways. Because they focus on what the program must do, not how, they can be written before the program, and can guide the programmer in writing the program that is required. Depending on the kind of specification, we may be able to verify the program against the specification, and/or we may be able to check at runtime that the specification is not violated by a particular run of the program. If we do find a discrepancy between the specification and the program, this may as easily indicate a defect in the specification as in the program; but either way, understanding is improved. The specification can also be used by potential reusers of the program, and for various other purposes. The most useful specifications are precise, but simpler and easier to understand than the program they specify.

What kinds of specification of model transformations may prove useful? How can we write specifications of model transformations which are simpler than the transformations themselves? One pragmatic approach would be to have a precise formal specification of the intended consistency relation, together with natural language description of how to choose between several consistent models.

## 8.3 Verification, Validation and Testing

To date, most work on verification or validation of model transformations turns out to be concerned not with verification or validation in the usual software engineering sense, of ensuring that the transformation conforms to its specification

and correctly expresses the user's requirements, but in the special sense of making sure that the transformation is sane. For example, [22] focuses on ensuring that a graph transformation is terminating and confluent; that is, well-defined. This is obviously crucial, but after this is achieved, we still have to address the “are we building the right transformation?” sense of validation and the “are we building the transformation right?” sense of verification.

To validate, or to test, an ordinary function, we will normally run it on various inputs. A difficulty in importing this to model transformations is the relative difficulty of constructing or finding suitable input models, which will tend to limit the amount of testing or validation done. Model transformations are expected to be written in a wide range of business contexts. If the model transformation itself is regarded as a product having value, it may be seen as legitimate to invest effort in validating it. If, however, a model transformation is developed for use within a single organisation, perhaps initially on a small range of models, this is less likely. Compounding this problem is the complexity of the models to which transformations are likely to be applied, and the difficulty of telling at a glance when a transformation is erroneous.

Given a precise specification of a model transformation, and the definition of the transformation in a semantically well-defined form, we may expect to be able to apply the body of work on formal verification of programs, and perhaps more usefully of concurrent systems. This brief statement, though, doubtless hides a need for hundreds of papers to work out the details.

In the field of testing, too, we will have to answer again a host of theoretical and engineering questions. How can test suites be generated (semi-)automatically? What are appropriate coverage criteria, and can they be automatically checked? Can mutation testing be helpful? etc.

## 8.4 Debugging

Once a test or a verification fails, the model transformation must be debugged: that is, the user must be helped to localise, understand and fix the problem. This may be done “live”, in the tool that is applying the transformation, or “forensically”, based on records of the transformation. In the latter field, [15] is interesting early work, done in the context of Tefkat transformations with link (traceability) information available. Much more remains to be done, however. We may note that even the debugging facilities available for models themselves are not yet very sophisticated, compared with what is routinely used for programs.

## 8.5 Maintenance

To date, we lack serious experience of what issues may arise in the maintenance of model transformations. Experience with maintenance of other software artefacts suggests that easily usable specifications will be useful, but that they will, in any case, get out of date; that readability of the representation will be crucial; and that a notion of refactorings of model transformations will be needed to prevent architectural degradation of large transformations. We may expect in

the long run to have to define model transformation languages with clear modularity permitting well-defined interfaces that can hide information; this returns us to the issue of compositionality. What the relationship should be between modularity of models and modularity of model transformations also remains to be seen. A concrete fear is that pattern-based identification of areas of a model where a transformation is relevant may possibly turn out to be particularly fragile. [19] describes interesting early work in this field, focusing on structuring and packaging bidirectional model transformations, specifically TGGs.

## 8.6 Tolerating Inconsistency

Anyone using bidirectional transformations asks: to what extent can inconsistency be tolerated? If not at all, then we need frameworks in which edits are applied simultaneously to both models, even though the user doing the editing sees only part of the effect of their edit. In most of the work we have discussed, the assumption is that inconsistency can be tolerated temporarily – it is acceptable for one, or even both, models to be edited independently – but that consistency must eventually be restored by the application of a model transformation. The other end of the spectrum is to accept that the two models will be inconsistent and work on pragmatic means to tolerate the inconsistency. But is this different from using a different, weaker notion of consistency? The body of work exploring this territory includes for example [11]: it might prove fruitful to explore applications to model driven development.

## 9 Conclusions

We have surveyed the field of bidirectional model transformations, as it appears at the beginning of 2008. We have pointed out some of the assumptions between which the developers of model transformation approaches must choose, and we have discussed some of the main existing tools. Finally, we have pointed out some areas where further work is needed.

What general conclusion can we draw about the future of bidirectional model transformations? Do they, indeed, have a long-term future, or will the languages developed for writing them die without being replaced? This paper has pointed out a number of areas – support for interactivity, for debugging, for verification, validation and testing, for maintenance, among others – that are only beginning to be addressed. In the author’s opinion, realisation of the full potential of bidirectional transformations depends on progress in these areas.

*Acknowledgements.* The author would like to thank the anonymous reviewers for their comments, and Joel Greenyer, Reiko Heckel, Conrad Hughes, Gabriele Taentzer and especially Benjamin Pierce for helpful discussions. Any errors are the author’s own.



## References

1. Amelunxen, C., Königs, A., Röttschke, T., Schürr, A.: MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066. pp. 361–375. Springer, Heidelberg (2006)
2. Antkiewicz, M., Czarnecki, K.: Framework-specific modeling languages with round-trip engineering. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 692–706. Springer, Heidelberg (2006)
3. Bohannon, A., Foster, J.N., Pierce, B.C., Pilkiewicz, A., Schmitt, A.: Boomerang: Resourceful lenses for string data. In: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Francisco, California (January 2008)
4. Brabrand, C., Möller, A., Schwartzbach, M.I.: Dual syntax for XML languages. In: Bierman, G., Koch, C. (eds.) DBPL 2005. LNCS, vol. 3774. Springer, Heidelberg (2005)
5. Braun, P., Marschall, F.: Transforming object oriented models with botl. *Electronic Notes in Theoretical Computer Science*, 72(3) (2003)
6. Chivers, H., Paige, R.F.: Xround: Bidirectional transformations and unifications via a reversible template language. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 205–219. Springer, Heidelberg (2005)
7. Cicchetti, A., Di Ruscio, D., Eramo, R.: Towards propagation of changes by model approximations. In: Proceedings of the Tenth IEEE International Enterprise Distributed Object Computing Conference (EDOC 2006), p. 24. IEEE Computer Society, Los Alamitos (2006)
8. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. *IBM Systems Journal*, special issue on Model-Driven Software Development 45(3), 621–645 (2006)
9. Ehrig, H., Ehrig, K., Ermel, C., Hermann, F., Taentzer, G.: Information preserving bidirectional model transformations. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 72–86. Springer, Heidelberg (2007)
10. Del Fabro, M.D., Jouault, F.: Model transformation and weaving in the AMMA platform. In: Proceedings of GTTSE 2005 (2006)
11. Finkelstein, A., Gabbay, D., Hunter, A., Kramer, J., Nuseibeh, B.: Inconsistency handling in multi-perspective specifications. *Transactions on Software Engineering* 20(8), 569–578 (1994)
12. 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 Transactions on Programming Languages and Systems* 29(3), 17 (2007)
13. Greenyer, J., Kindler, E.: Reconciling TGGs with QVT. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 16–30. Springer, Heidelberg (2007)
14. Grunske, L., Geiger, L., Lawley, M.: A graphical specification of model transformations with triple graph grammars. In: proceedings of 2nd European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA) (November 2005)
15. Hibberd, M., Lawley, M., Raymond, K.: Forensic debugging of model transformations. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 589–604. Springer, Heidelberg (2007)

16. Hu, Z., Mu, S.-C., Takeichi, M.: A programmable editor for developing structured documents based on bidirectional transformations. In: Proceedings of the 2004 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM 2004), pp. 178–189 (2004)
17. Jouault, F., Kurtev, I.: Transforming models with atl. In: Bruel, J.-M. (ed.) *MoDELS 2005*. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
18. Kawanaka, S., Hosoya, H.: biXid: a bidirectional transformation language for XML. In: Proceedings of the International Conference on Functional Programming, ICFP 2006, pp. 201–214 (2006)
19. Klar, F., Königs, A., Schürr, A.: Model transformation in the large. In: Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007, pp. 285–294. ACM, New York (2007)
20. Königs, A., Schürr, A.: Tool Integration with Triple Graph Grammars - A Survey. In: Heckel, R. (ed.) *Proceedings of the SegraVis School on Foundations of Visual Modelling Techniques*. Electronic Notes in Theoretical Computer Science, vol. 148, pp. 113–150. Elsevier Science Publ., Amsterdam (2006)
21. Königs, A.: Model transformation with triple graph grammars. In: Proceedings of the Workshop on Model Transformations in Practice, at *MODELS 2005* (September 2005)
22. Küster, J.M.: Definition and validation of model transformations. *Software and Systems Modeling (SoSyM)* 5(3), 233–259 (2006)
23. Lambert Meertens. Designing constraint maintainers for user interaction. Unpublished manuscript (June 1998), <http://www.kestrel.edu/home/people/meertens/>
24. Mens, T., Van Gorp, P.: A taxonomy of model transformation. *Electr. Notes Theor. Comput. Sci.* 152, 125–142 (2006)
25. Mu, S.-C., Hu, Z., Takeichi, M.: An algebraic approach to bi-directional updating. In: Chin, W.-N. (ed.) *APLAS 2004*. LNCS, vol. 3302, pp. 2–20. Springer, Heidelberg (2004)
26. Mu, S.-C., Hu, Z., Takeichi, M.: An injective language for reversible computation. In: Kozen, D. (ed.) *MPC 2004*. LNCS, vol. 3125, pp. 289–313. Springer, Heidelberg (2004)
27. Oliveira, J.N.: Data transformation by calculation. In: *Informal GTTSE 2007 Proceedings*, pp. 139–198 (July 2007)
28. OMG. MOF2.0 query/view/transformation (QVT) adopted specification. OMG document ptc/05-11-01 (2005), [www.omg.org](http://www.omg.org)
29. Sendall, S., Küster, J.M.: Taming model round-trip engineering. In: Proceedings of Workshop on Best Practices for Model-Driven Software Development, Vancouver, Canada (2004)
30. Stevens, P.: Bidirectional model transformations in qvt: Semantic issues and open questions. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) *MODELS 2007*. LNCS, vol. 4735, pp. 1–15. Springer, Heidelberg (2007)
31. Taentzer, G., Ehrig, K., Guerra, E., de Lara, J., Lengyel, L., Levendovsky, T., Prange, U., Varro, D., Varro-Gyapay, S.: Model transformation by graph transformation: A comparative study. In: Proceedings of the Workshop on Model Transformations in Practice, at *MODELS 2005* (September 2005)
32. Witkop, S.: MDA users' requirements for QVT transformations. OMG document 05-02-04 (2005), [www.omg.org](http://www.omg.org)

# Evolving a DSL Implementation

Laurence Tratt

Bournemouth University, Poole, Dorset, BH12 5BB, United Kingdom  
laurie@tratt.net, <http://tratt.net/laurie/>

**Abstract.** Domain Specific Languages (DSLs) are small languages designed for use in a specific domain. DSLs typically evolve quite radically throughout their lifetime, but current DSL implementation approaches are often clumsy in the face of such evolution. In this paper I present a case study of an DSL evolving in its syntax, semantics, and robustness, implemented in the Converge language. This shows how real-world DSL implementations can evolve along with changing requirements.

## 1 Introduction

Developing complex software in a General Purpose Language (GPL) often leads to situations where problems are not naturally expressible within the chosen GPL. This forces users to find a workaround, and encode their solution in as practical a fashion as they are able. Whilst such workarounds and encodings are often trivial, they can be exceedingly complex. DSLs aim to tackle the lack of expressivity in GPLs by allowing users to use mini-languages defined for specific problem areas. [1] define DSLs as ‘languages tailored to a specific application domain. They offer substantial gains in expressiveness and ease of use compared with GPLs in their domain of application’. [2] describes the typical costs of a DSL, noting that a small extra initial investment in a DSL implementation typically leads to long term savings, in comparison to alternative routes. Exactly what identifies a particular language as being a ‘DSL’ is subjective, but intuitively I define it as a language with its own syntax and semantics, and which is smaller and less generic than a typical GPL such as Java.

Traditionally DSLs – for example the UNIX `make` program or the `yacc` parsing system – have been implemented as stand alone systems. The resulting high implementation costs, primarily due to the difficulties of practical reuse, have hindered the development of DSLs. Implementing DSLs as stand alone systems also leads to problems when DSLs evolve. DSLs tend to start out as small, declarative languages [3], but most tend to acquire new features as they are used in practise; such features tend to be directly borrowed from GPLs [2]. So while DSL implementations tend over time to resemble programming language implementations, they frequently lack the quality one might expect in such a system due to the unplanned nature of this evolution.

Recently, dedicated DSL implementation approaches such as Stratego [4], the commercial XMF [5], Converge [6], and others (e.g. [7,8,9]) have substantially reduced implementation costs through the embedding of DSLs in host languages.

As noted in [2,3], DSLs tend to start small but grow rapidly when users find them useful, and desire more power. Specifically, such evolution often takes the form of functionality influenced by that found in GPLs. Continual evolution of DSL implementations is often difficult because such evolution is generally both unplanned and unanticipated, and therefore leads to the implementation becoming increasingly difficult to maintain [2]. In this paper I present a case study of a DSL for state machines implemented within Converge. I then show how this example can be easily evolved to a substantially more powerful version without compromising the quality of the implementation, and indeed improving the user experience. The evolution in this paper is intended to show typical unplanned evolution, where an implementation is gradually edited to reflect new and changing requirements.

This paper is structured as follows. First I present a brief overview of Converge, and its DSL related features (section 2). I then outline the case study and present an initial implementation (section 3) before extending its functionality (section 4) and increasing its robustness (section 5).

## 2 Converge

This section gives a brief overview of basic Converge features that are relevant to the main subject of this paper. Whilst this is not a replacement for the language manual [10], it should allow readers familiar with a few other programming languages the opportunity to quickly come to grips with the most important areas of Converge, and to determine the areas where it differs from other languages.

### 2.1 Fundamental Features

Converge's most obvious ancestor is Python [11] resulting in an indentation based syntax, a similar range and style of datatypes, and general sense of aesthetics. The most obvious initial difference is that Converge is a slightly more static language: all namespaces (e.g. a modules' classes and functions, and all variable references) are determined statically at compile-time. Converge's scoping rules are different from many other languages, and are intentionally very simple. Essentially Converge's functions are synonymous with both closures and blocks. Converge is lexically scoped, and there is only one type of scope. Variables do not need to be declared before their use: assigning to a variable anywhere in a block makes that variable local throughout the block, and accessible to inner blocks. Variable references search in order from the innermost block outwards, ultimately resulting in a compile-time error if a suitable reference is not found. Fields within a class are not accessible via the default scoping mechanism: they must be referenced via the `self` variable which is the first argument in any *bound function* (functions declared within a class are automatically bound functions). The overall justification for these rules is to ensure that, unlike similar languages such as Python, Converge's namespaces are entirely statically calculable.

Converge programs are split into modules, which contain a series of *definitions* (imports, functions, classes and variable definitions). Each module is individually compiled into a bytecode file, which can be linked to other files to produce an executable which can be run by the Converge VM. If a module is the *main module* of a program (i.e. passed first to the linker), Converge calls its `main` function to start execution. The following module shows a caching Fibonacci generating class, and indirectly shows Converge's scoping rules (the `i` and `fib_cache` variables are local to the functions they are contained within), printing `8` when run:

```
import Sys

class Fib_Cache:
  func init():
    self.cache := [0, 1]

  func fib(x):
    i := self.cache.len()
    while i <= x:
      self.cache.append(self.cache[i - 2] + self.cache[i - 1])
      i += 1
    return self.cache[x]

func main():
  fib_cache := Fib_Cache.new()
  Sys::println(fib_cache.fib(6))
```

## 2.2 Compile-Time Meta-programming

For the purposes of this paper, compile-time meta-programming can be largely thought of as being equivalent to macros; more precisely, it allows the user of a programming language a mechanism to interact with the compiler to allow the construction of arbitrary program fragments by user code. Compile-time meta-programming allows users to e.g. add new features to a language [7] or apply application specific optimizations [9]. Converge's compile-time meta-programming facilities were inspired by those found in Template Haskell (TH) [12], and are detailed in depth in [6]. In essence Converge provides a mechanism to allow its concrete syntax to naturally create Abstract Syntax Trees (ASTs), which can then be spliced into a source file.

The following program is a simple example of compile-time meta-programming, trivially adopted from its TH cousin in [8]. `expand_power` recursively creates an expression that multiplies `x` `n` times; `mk_power` takes a parameter `n` and creates a function that takes a single argument `x` and calculates  $x^n$ ; `power3` is a specific power function which calculates  $n^3$ :

```
func expand_power(n, x):
  if n == 0:
    return [| 1 |]
  else:
    return [| ${x} * ${expand_power(n - 1, x)} |]
```

```

func mk_power(n):
  return [
    func (x):
      return ${expand_power(n, [ x ])}
  ]

power3 := ${mk_power(3)}

```

The user interface to compile-time meta-programming is inherited directly from TH. *Quasi-quoted* expressions `[| ... |]` build ASTs that represent the program code contained within them whilst ensuring that variable references respect Converge’s lexical scoping rules. Splice annotations `$(...)` evaluate the expression within at compile-time (and before VM instruction generation), replacing the splice annotation itself with the AST resulting from its evaluation. This is achieved by creating a temporary module containing the splice expression in a function, compiling the temporary module into bytecode, injecting it into the running VM, and then evaluating the function therein. Insertions `${...}` are used within quasi-quotes; they evaluate the expression within and copy the resulting AST into the AST being generated by the quasi-quote.

When the above example has been compiled into VM instructions, `power3` essentially looks as follows:

```

power3 := func (x):
  return x * x * x * 1

```

## 2.3 DSL Blocks

A DSL can be embedded into a Converge source file via a *DSL block*. Such a block is introduced by a variant on the splice syntax `$(<<expr>>)` where *expr* should evaluate to a function (the *DSL implementation function*). The DSL implementation function is called at compile-time with a string representing the DSL block, and is expected to return an AST which will replace the DSL block in the same way as a normal splice: compile-time meta-programming is thus the mechanism which facilitates embedding DSLs. Colloquially one uses the DSL implementation function to talk about the DSL block as being ‘an *expr* block’. DSL blocks make use of Converge’s indentation based syntax; when the level of indentation falls, the DSL block is finished.

An example DSL block for a railway timetable DSL is as follows:

```

func timetable(dsl_block, src_infos):
  ...

$(<<timetable>>:
  8:25 Exeter St. Davids
  10:20 Salisbury
  11:49 London Waterloo

```

As shall be seen later, DSL blocks have several useful features, particularly relating to high quality error reporting. Although in this paper I only discuss DSL blocks, Converge also supports *DSL phrases* which are essentially intra-line DSL inputs, suitable for smaller DSLs such as SQL queries.

### 3 Initial Case Study

The example used in this paper is that of a generic state machine. Although state machines are often represented graphically, they are easily represented textually. I start with a particularly simple variant of state machines which represents the basics: states and transitions with events. For example, Figure 1 shows a state machine which we wish to represent textually so that we can have a running state machine we can fire events at and examine its behaviour. In the rest of this section, I show the complete definition of a simple textual state machine DSL.

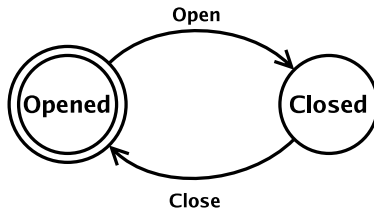


Fig. 1. Simple state machine of a door

#### 3.1 DSL Grammar

Since we use DSL blocks within Converge, much of the potential difficulty with embedding a DSL is automatically taken care of. The first action of a DSL author is therefore to define a grammar his DSL must conform to. Converge allows DSL authors to parse the text of a DSL block in any way they choose. However most DSLs can be defined in a way which allows them to make use of Converge's flexible tokenizer (sometimes called a lexer), and its built-in Earley parser. This allows the implementation work for a DSL to be minimised as Earley parsing can deal with any context free grammar, without the restrictions common to most parsing approaches [13]. Expressing a suitable grammar for simple state machines is thus simple:

```

parse := $<<DSL::mk_parser("system", ["state", "transition", "and", \
  "or", "from", "to"], [])>>:
  system    ::= element ( "NEWLINE" element )*
  element   ::= state
              | transition
  state     ::= "STATE" "ID"
  transition ::= "TRANSITION" "FROM" "ID" "TO" "ID" event
  event     ::= ":" "ID"
              |
  
```

As this code fragment suggests, grammars are themselves a DSL in Converge. The above example creates a parser which uses Converges default tokenizer, adds new keywords (`state`, `transition` etc.) and using a specified top-level rule `system`.

### 3.2 Creating a Parse Tree

The DSL implementation function is passed a DSL block string which it should parse against the DSL's grammar. Since DSL implementation functions tend to follow the same form, Converge provides a convenience function `CEI::dsl_parse` which performs parsing in one single step. The state machine DSL implementation function and an example DSL block look as follows:

```
func sm(dsl_block, src_infos):
  parse_tree := CEI::dsl_parse(dsl_block, src_infos, ["state", \
    "transition", "and", "or", "from", "to"], [], GRAMMAR, "system")
  return SM_Translator.new().generate(parse_tree)

Door := $<<sm>>:
  state Opened
  state Closed

  transition from Opened to Closed: close
  transition from Closed to Opened: open
```

The CEI (Compiler External Interface) `dsl_parse` convenience function takes a DSL block, a list of src infos, a list of extra keywords above and beyond Converge's standard keywords, a list of extra symbols, a grammar, and the name of the grammar's start rule. It returns a parse tree (that is, a tree still containing tokens). Parse trees are Converge lists, with the first element in the list representing the production name, and the resulting elements being either tokens or lists representing the production. Tokens have two slots of particular interest: `type` contains the tokens type (e.g. ID); `value` contains the particular value of the token (e.g. foo). A subset of the parse tree for the above DSL block is as follows:

```
["system", ["element", ["state", <STATE state>, <ID Opened>]],
<NEWLINE>, ["element", ["state", <STATE state>, <ID Closed>]], ... ]
```

### 3.3 Translating the Parse Tree to an AST

The second, final, and most complex action a DSL author must take is to translate the parse tree into a Converge AST, using quasi-quotes and so on. Converge provides a simple framework for this translation, where a translation class (`SM_Translator` in the above DSL implementation function) contains a function `_t_production_name` for each production in the grammar. For the simple state machine DSL, we wish to translate the parse tree into an anonymous class which can be instantiated to produce a running state machine, which can then receive and act upon events. The starting state is taken to be the first state in the DSL block. Transitions may have an event attached to them or not; if they have no event, they are unconditionally, and non-deterministically, taken. A slightly elided version of the translation is as follows:



```

1 class SM_Translator(Traverser::Strict_Traverser):
2   func _t_system(self, node):
3     sts := [all translated states]
4     tns := [all translated transitions]
5
6     return [|
7       class:
8         states := ${CEI::ilist(sts)}
9         transitions := ${CEI::ilist(tns)}
10
11       func init(self):
12         self.state := ${sts[0]}
13         while self.transition("")
14
15       func event(self, e):
16         if not self.transition(e):
17           raise Exceptions::User_Exception.new(Strings::format( \
18             "Error: No valid transition from state.")
19         while self.transition("")
20
21       func transition(self, e):
22         for tn := self.transitions.iter():
23           if tn.from == self.state & tn.event == e:
24             Sys::println("Event ", e, \
25               " causes transition to state ", tn.to)
26             self.state := tn.to
27             break
28           exhausted:
29             return fail
30     |]
31
32   func _t_element(self, node):
33     return self._preorder(node[1])
34
35   func _t_state(self, node):
36     // state ::= "STATE" "ID"
37     return CEI::istring(node[2].value)
38
39   func _t_transition(self, node):
40     // transition ::= "TRANSITION" "FROM" "ID" "TO" "ID" event
41     return [| Transition.new(${CEI::istring(node[3].value)}, \
42       ${CEI::istring(node[5].value)}, ${self._preorder(node[-1])}) |]
43
44   func _t_event(self, node):
45     // event ::= ":" "ID"
46     //           |
47     if node.len() == 1:
48       return [| "" |]
49     else:
50       return CEI::istring(node[2].value)
51
52 class Transition:
53   func init(self, from, to, event):
54     self.from := from
55     self.to := to
56     self.event := event

```

At a high level, this translation is relatively simple: states are transformed into strings; transitions are transformed into instantiations of the `Transition` class. The resulting anonymous class thus knows the valid states and transitions of the state machine, and given an event can transition to the correct state, or report errors. Certain low-level details require more explanation. The calls to `self._preorder` reference a method which, given a node in a parse tree, calls the appropriate `_t_` function. The CEI module defines functions for every Converge AST type allowing them to be created manually when quasi-quotes do not suffice. For example a call such as `CEI::istring("foo")` (e.g. lines 37) returns an AST string whose content is 'foo'. The reference to the `Transition` class in line 41 is possible since quasi-quotes can refer to top-level module definitions, as these inherently cross compile-time staging boundaries.

### 3.4 Using the DSL

Given the complete, if simplistic, definition of state machine DSL we now have, it is possible to instantiate a state machine and fire test events at it:

```
door := Door.new()
door.event("close")
door.event("open")
```

which results in the following output:

```
Event close causes transition to state Closed
Event open causes transition to state Opened
```

As this section has shown, we have been able to create a functioning DSL with its own syntax, whose complete definition is less than 75 lines of code. I assert that a corresponding implementation of this DSL as a stand-alone application would be considerably larger than this, having to deal with external parsing systems, IO, error messages and other boiler-plate aspects which are largely invisible in the Converge DSL implementation approach.

## 4 Extending the Case Study

As noted in [23], DSLs tend to start small but grow rapidly when users find them useful, and desire more power. In a traditional stand alone implementation, such changes might be difficult to integrate. In this section I show how we can easily extend the Converge DSL implementation.

### 4.1 An Extended State Machine

As an example of a more complex type of state machine, we define a state machine of a vending machine which dispenses drinks and sweets:

```
drinks := 10
sweets := 20

state Waiting
state Vend_Drink
state Vend_Sweet
state Empty
```

```

transition from Waiting to Vend_Drink: Vend_Drink \
  [ drinks > 0 ] / drinks := drinks - 1
transition from Vend_Drink to Waiting: Vended [drinks > 0 or sweets > 0]

transition from Waiting to Vend_Sweet: Vend_Sweet \
  [ sweets > 0 ] / sweets := sweets - 1
transition from Vend_Sweet to Waiting: Vended [sweets > 0 or drinks > 0]

transition from Vend_Sweet to Empty: Vended [drinks == 0 and sweets == 0]
transition from Vend_Drink to Empty: Vended [drinks == 0 and sweets == 0]

```

This state machine makes use of variables, guards, and actions. Variables can hold integers or strings, and must be assigned an initial value. Guards such as `[drinks > 0]` are additional constraints to events; they must hold in order for a transition to be taken. Actions such as `sweets := sweets - 1` are executed once a transitions constraints have been evaluated and the transition has been taken.

## 4.2 Extending the Grammar

As before, the DSL author's first action is to define – or in this case, to extend – the grammar of his DSL. An elided extension to the previous grammar is as follows:

```

element    ::= state
           | transition
           | var_def
transition ::= "TRANSITION" "FROM" "ID" "TO" "ID" event guard action
var_def    ::= "ID" ":" "=" const
guard      ::= "[" B "]"
           |
action     ::= "/" C
           ::=
B          ::= B "AND" B %precedence 5
           | B "OR" B %precedence 5
           | B "==" B %precedence 10
           | B ">" B %precedence 10
           | E
C          ::= A ( ";" A )*
A          ::= "ID" ":" "=" E
           | E
E          ::= E "+" E
           | E "-" E
           | var_lookup
           | const

```

The `%precedence` markings signify to the Earley parser which of several alternatives is to be preferred in the event of an ambiguous parse, with higher precedence values having greater priority. Essentially the extended grammar implements a syntax in a form familiar to many state machine users. Guards are conditions or expressions; actions are sequences of assignments or expressions; and expressions include standard operators.

### 4.3 Extending the Translation

The first thing to note is that the vast majority of the translation of section 3.3 can be used unchanged in our evolved DSL. The anonymous state machine class gains a `vars` slot which records all variable names and their current values, and `get_var` / `set_var` functions to read and update `vars`. Transitions gain `guard` and `action` slots which are functions. `transition` is then updated to call these functions, passing the state machine to them, so that it can read and write variables. The updated `transition` function is as follows:

```
func transition(self, e):
    for tn := self.transitions.iter():
        if tn.from == self.state & tn.event == e & tn.guard(self):
            Sys::println("Event ", e, " causes transition to state ", tn.to)
            self.state := tn.to
            tn.action(self)
            break
    exhausted:
        return fail
```

The remaining updates to the translation are purely to translate the new productions in the grammar. The full translation is less than 200 lines of code, but in the interests of brevity I show a representative subset; the translation of guards, and the translation of variable lookups and constants.

```
1  func _t_guard(self, node):
2  // guard ::= "[" B "]"
3  //      |
4  if node.len() == 1:
5  guard := [| 1 |]
6  else:
7  guard := self._preorder(node[2])
8  return [|
9  func (&sm):
10     return ${guard}
11  |]
12
13 func _t_B(self, node):
14 // B ::= B "AND" B
15 //      | B "OR" B
16 //      | B "==" B
17 //      | B ">" B
18 //      | E
19 if node.len() == 4:
20     lhs := self._preorder(node[1])
21     rhs := self._preorder(node[3])
22     ndif node[2].type == "AND":
23         return [| ${lhs} & ${rhs} |]
24     elif node[2].type == "OR":
25         return [| ${lhs} | ${rhs} |]
26     elif node[2].type == "==":
27         return [| ${lhs} == ${rhs} |]
28     elif node[2].type == ">":
29         return [| ${lhs} > ${rhs} |]
30 else:
31     return self._preorder(node[1])
32
```

```

33 func _t_const(self, node):
34     // const ::= "INT"
35     //     | "STRING"
36     ndif node[1].type == "INT":
37         return CEI::iint(Builtins::Int.new(node[1].value))
38     elif node[1].type == "STRING":
39         return CEI::istring(node[1].value)
40
41 func _t_var_lookup(self, node):
42     // var_lookup ::= "ID"
43     return [| &sm.get_var(${CEI::istring(node[1].value)}) |]

```

The majority of this translation is simple, and largely mechanical. Guards are turned into functions (lines 8–11) which take a single argument (a state machine) and return true or false. An empty guard always evaluates to 1 (line 5), which can be read as ‘true’. The translation of guards (lines 20–29) is interesting, as it shows that syntactically distinct DSLs often have a very simple translation into a Converge AST, as Converge’s expression language is unusually rich in expressive power by imperative programming language standards (including features such as backtracking which we do not use in this paper). State machine constants (strings and integers) are directly transformed into their Converge equivalents (lines 36–39).

#### 4.4 Communication between AST Fragments

One subtle aspect of the translation deserves special explanation, which are the two `&sm` variables (lines 43 and 9). These relate to the fact that the state machine which is passed by the `transition` function to the generated guard function (lines 8–11) needs to be used by the variable lookup translation (line 43). The effect we wish to achieve is that the translated guard function looks approximately as follows:

```

func (sm):
    return sm.get_var("x") < 1

```

By default, Converge’s quasi-quote scheme generates *hygienic* ASTs. The concept of hygiene is defined in [14], and is most easily explained by example. Consider the Converge functions `f` and `g`:

```

func f():
    return [| x := 4 |]

func g():
    x := 10
    $<f()>
    Sys::println(x)

```

The question to ask oneself is simple: when `g` is executed, what is printed to screen? In older macro systems, the answer would be 4 since when, during compilation, the AST from `f` was spliced into `g`, the assignment of `x` in `f` would ‘capture’ the `x` in `g`. This is a serious issue since it makes embeddings and macros ‘treacherous [, working] in all cases but one: when the user ... inadvertently picks the wrong identifier name’ [14].

Converge’s quasi-quote scheme therefore preemptively  $\alpha$ -renames variables to a *fresh name* – guaranteed by the compiler to be unique – thus ensuring that unintended variable capture can not happen. While this is generally the required behaviour, it can cause practical problems when one is building up an AST in fragments, as we are doing in our state machine translation. In normal programming, one of the most common way for local chunks of code to interact is via variables; however, hygiene effectively means that variables are invisible between different code chunks. Thus, by default, there is no easy way for the variable passed to the generated guard function (lines 8-11) to be used by the variable lookup translation (line 43).

The traditional meta-programming solution to this problem is to manually generate a fresh name, which must then be manually passed to all translation functions which need it. A sketch of a solution for Converge would be as follows:

```
func _t_guard(self, node):
    // guard ::= "[" B "]"
    //      |
    sm_var_name := CEI::fresh_name()
    if node.len() == 1:
        guard := [| 1 |]
    else:
        guard := self._preorder(node[2], sm_var_name)
    return [|
        func (${CEI::iparam(CEI::ivar(sm_var_name))}):
            return ${guard}
    |]

func _t_var_lookup(self, node, sm_var_name):
    // var_lookup ::= "ID"
    return [| ${CEI::ivar(sm_var_name)}.get_var( \
        ${CEI::istring(node[1].value)}) |]
```

This idiom, while common in other approaches, is intricate and verbose. Indeed, the quantity and spread of the required boilerplate code can often overwhelm the fundamentals of the translation.

Converge therefore provides a way to switch off hygiene in quasi-quotes; variables which are prefixed by `&` are not  $\alpha$ -renamed. Thus the `sm` variable in line 43 dynamically captures the `sm` variable defined in line 9, neatly obtaining the effect we desire. This is a very common translation idiom in Converge, and is entirely safe in this translation<sup>1</sup>.

## 5 Evolving a Robust DSL

In the previous section, I showed how a Converge DSL can easily evolve in expressive power. The DSL defined previously suffers in practice from one fundamental flaw. When used correctly, it works well; when used incorrectly, it is

<sup>1</sup> Although I do not show it in this paper, translations which integrate arbitrary user code can cause this idiom to become unsafe; Converge provides a relatively simple work around for such cases.

difficult to understand what went wrong. This is a common theme in DSLs: initial versions with limited functionality are used only by knowledgeable users; as the DSLs grow in power, they are used by increasingly less knowledgeable users. This has a dual impact: the more powerful the DSL it is, the more difficult it is to interpret errors; and the less knowledgeable the user, the less their ability to understand whether they caused the error, if so, how to fix it.

In this section, I show how Converge DSLs can easily add debugging support which makes the use of complex DSLs practical.

## 5.1 DSL Errors

Returning to the vending machine example of section [4.1](#), let us change the guard on the first transition from `drinks > 0` to `drinks > "foo"`. The vending machine state machine is contained in a file `ex.cv` and the state machine DSL definition in `SM.cv`. As we might expect, this change causes a run-time exception, as Converge does not define a size comparison between integers and strings. The inevitable run-time exception and traceback look as follows:

```
Traceback (most recent call at bottom):
 1: File "ex2.cv", line 29, column 22
 2: File "SM.cv", line 118, column 27
 3: File "SM.cv", line 124, column 57
 4: File "SM.cv", line 235, column 21
 5: (internal), in Int.>
Type_Exception: Expected arg 2 to be conformant to Number but got
instance of String.
```

This traceback gives very little clue as to where in the DSL the error occurred. Looking at line 235 of `SM.cv` merely pinpoints the `_t_B` translation function. Since we know in this case that comparing the size of a number and a string is invalid, we can rule out the translation itself being incorrect. The fundamental problem then becomes that errors are not reported in terms of the users input. Knowing that the error is related to an AST generated from the `_t_B` translation function is of limited use as several of the vending machines guards also involve the greater than comparison.

## 5.2 Src Infos

DSL implementation functions take two arguments: a string representing the DSL block and a list of src infos. A src info is a (src path, char offset) pair which records a relationship with a character offset in a source file. The Converge tokenizer associates a src info with every token; the parse tree to AST conversion carries over the relevant src infos; and the bytecode compiler associates every bytecode instruction with the appropriate src infos. As this suggests, the src info concept is used uniformly throughout the Converge parser, compiler, and VM.

From this papers perspective, an important aspect of src infos is that tokens, AST elements, and bytecode instructions can be associated with more than one src info. Converge provides a simple mechanism for augmenting the src infos that quasi-quoted code is associated with. Quasi-quotes have an extended form

[<*e*>| ... ] where *e* is an expression which must evaluate to a list of src infos. As we shall see, a standard idiom is to read src infos straight from tokens (via its `src_infos` slot) into the extended quasi-quotes form.

### 5.3 Augmenting Quasi-quoted Code

Using the extended form of quasi-quotes, we can easily augment the translation of section 4.3 to the following:

```
func _t_B(self, node):
    // B ::= B "<" B
    ...
    elif node[2].type == ">":
        return [<node[2].src_infos>| ${lhs} > ${rhs} |]
    ...
```

In other words, we augment the quasi-quoted code with src infos directly relating it to the specific location in the user's DSL input where the size comparison was made. When we re-compile and re-run the altered vending machine DSL, we get the following backtrace:

```
Traceback (most recent call at bottom):
 1: File "ex2.cv", line 29, column 22
 2: File "SM.cv", line 118, column 27
 3: File "SM.cv", line 124, column 57
 4: File "SM.cv", line 235, column 21
    File "ex2.cv", line 15, column 74
 5: (internal), in Int.>
Type_Exception: Expected arg 2 to be conformant to Number but got
instance of String.
```

The way to read this is that the fourth entry in the backtrace is related to two source locations: one is the quasi-quoted code itself (in `SM.cv`) and the other is a location within the vending machine DSL (in `ex2.cv`). This allows the user to pinpoint precisely where within their DSL input the error occurred, which will then allow them – one hopes – to easily rectify it. This is a vital practical aid, making DSL debugging feasible where it was previously extremely challenging. Because of the extended form of quasi-quotes, augmenting a DSL translation with code to record such information is generally a simple mechanical exercise.

### 5.4 Statically Detected Errors

Src infos are not only useful for aiding run-time errors. Converge also uses the same concept to allow DSL implementations to report errors during the translation of the DSL. For example, as our state machine DSL requires the up-front declaration of all variables to be used within it, we can easily detect references to undefined variables. The `CEI::error` function takes an arbitrary error message, and a list of src infos and reports an error to the user. Given that the translation class defines a slot `var_names` which is a set of known variables, we can then alter the `_t_var_lookup` function as follows:



```

func _t_var_lookup(self, node):
  // var_lookup ::= "ID"
  if not self.var_names.find(node[1].value):
    CEI::error(Strings::format("Unknown state-machine variable '%s'.", \
      node[1].value), node[1].src_infos)
  return [<node[1].src_infos>| &sm.get_var( \
    ${CEI::istring(node[1].value)}) |]

```

When an unknown variable is encountered, an error message such as the following is printed, and compilation halts:

```
Error: Line 53, column 66: Unknown state-machine variable 'amuont'.
```

## 6 Related Work

Several approaches have been suggested for DSL implementation. Hudak presented the notion of Domain Specific Embedded Languages (DSEs) [2] where DSLs are implemented using a languages normal features. The advantage of this approach is that it allows an otherwise entirely ignorant language to be used to embed DSLs, and also allows DSLs to be relatively easily combined together. The disadvantage is that the embedding is indirect, and limited to what can be easily expressed using these pre-existing components. DSLs implemented in Converge have considerably more syntactic flexibility.

TXL [15] and ASF+SDF are similar, generic source to source transformation languages [16]. Both are mature and efficient; TXL has been used to process billions of lines of code [15]. Furthermore such approaches are inherently flexible as they can be used with arbitrary source and target languages; unlike Converge, they can embed DSLs into any host language. However this flexibility means that they have little knowledge of the host language's semantics beyond the simple structure recorded in parse trees. This makes safe embeddings hard to create, whereas Converge based systems can use the compilers inherent knowledge of the host language to avoid such issues.

MetaBorg uses a combination of tools to allow language grammars to be extended in an arbitrary fashion using a rule rewriting system [4]. Although MetaBorg by default operates on parse trees in the same way as TXL, it comes with standard support for representing some of the semantics of languages such as Java. This allows transformation authors to write more sophisticated transformations, and make some extra guarantees about the safety of their transformations. Although MetaBorg is in theory capable of defining any embedding, its authors deliberately narrow their vision for MetaBorg to a 'method for promoting APIs to the language level.' This is a sensible restriction since DSLs that result from promoting a particular API to the language level will tend to shadow that API; therefore instances of the DSL will generally translate fairly directly into API calls which limits the potential for safety violations. In contrast, Converge provides coarser-grained support for implementing larger DSLs.

Macro systems have long been used to implement DSLs. Lisp was the first language with a macro system, and although its syntax is inherently flexible, it

is not possible to change it in a completely arbitrary fashion as Converge allows – Lisp DSLs are limited to what can be naturally expressed in Lisp’s syntax. Furthermore whilst this mechanism has been used to express many DSLs, its tight coupling to Lisp’s syntactic minimalism has largely prevented similar approaches being applied to other, more modern programming languages [17]. Therefore despite Lisp’s success in this area, for many years more modern systems struggled to successfully integrate similar features [6]. More recently languages such as Template Haskell [12] (which is effectively a refinement of the ideas in MetaML [18]; see [6] for a more detailed comparison of these languages with Converge) have shown how sophisticated compile-time meta-programming systems can be implemented in a modern language. However such languages still share Lisp’s inability to extend the languages syntax.

Nemerle uses its macro system to augment the compilers grammar as compilation is in progress [19]. However only relatively simple, local additions to the syntax are possible and the grammar extensions must be pre-defined; it is not intended, or suitable, for implementing complex DSLs. In comparison to Nemerle, MetaLua allows more flexible additions to the grammar being compiled but has no support for e.g. hygiene as in Converge, which makes implementing large DSLs problematic [20].

## 7 Conclusions

In this paper, I showed a case study of a state machine DSL evolving in terms of functionality and robustness. There are many other areas in which the DSL could evolve, including showing how DSL code can naturally interact with ‘normal’ Converge code. However I hope this papers’ case study gives a clear indication as to how a dedicated DSL implementation approach can make DSL evolution – in whatever form it takes – practical.

I am grateful to the anonymous referees whose comments have helped to improve this paper. Any remaining mistakes are my own.

This research was partly funded by Tata Consultancy Services.

Free implementations of Converge (under a MIT / BSD-style license) can be found at <http://convergepl.org/>, and are capable of executing all of the examples in this paper.

## References

1. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. Technical report, Centrum voor Wiskunde en Informatica (December 2003)
2. Hudak, P.: Modular domain specific languages and tools. In: Proceedings of Fifth International Conference on Software Reuse, pp. 134–142 (June 1998)
3. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: An annotated bibliography. SIGPLAN Notices, vol. 35, pp. 26–36 (June 2000)

4. Bravenboer, M., Visser, E.: Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In: Schmidt, D.C. (ed.) Proc. OOPSLA 2004, Vancouver, Canada. ACM SIGPLAN, New York (2004)
5. Clark, T., Evans, A., Sammut, P., Willans, J.: An executable metamodelling facility for domain specific language design. In: Proc. 4th OOPSLA Workshop on Domain-Specific Modeling (October 2004)
6. Tratt, L.: Compile-time meta-programming in a dynamically typed OO language. In: Proceedings Dynamic Languages Symposium, pp. 49–64 (October 2005)
7. Sheard, T., el Abidine Benaissa, Z., Pasalic, E.: DSL implementation using staging and monads. In: Proc. 2nd conference on Domain Specific Languages. SIGPLAN, vol. 35, pp. 81–94. ACM, New York (1999)
8. Czarnecki, K., O'Donnell, J., Striegnitz, J., Taha, W.: DSL implementation in MetaOCaml, Template Haskell, and C++. In: Lengauer, C., Batory, D., Consel, C., Odersky, M. (eds.) Domain-Specific Program Generation. LNCS, vol. 3016, pp. 50–71. Springer, Heidelberg (2004)
9. Seefried, S., Chakravarty, M., Keller, G.: Optimising Embedded DSLs using Template Haskell. In: Third International Conference on Generative Programming and Component Engineering, Vancouver, Canada, pp. 186–205. Springer, Heidelberg (2004)
10. Tratt, L.: Converge Reference Manual (July 2007), <http://www.convergepl.org/documentation/> (accessed August 16, 2007)
11. van Rossum, G.: Python 2.3 reference manual (2003), <http://www.python.org/doc/2.3/ref/ref.html> (accessed August 31, 2005)
12. Sheard, T., Jones, S.P.: Template meta-programming for Haskell. In: Proceedings of the Haskell workshop 2002. ACM, New York (2002)
13. Earley, J.: An efficient context-free parsing algorithm. Communications of the ACM 13(2) (February 1970)
14. Kohlbecker, E., Friedman, D.P., Felleisen, M., Duba, B.: Hygienic macro expansion. In: Symposium on Lisp and Functional Programming, pp. 151–161. ACM, New York (1986)
15. Cordy, J.R.: TXL - a language for programming language tools and applications. In: Proc. LDTA 2004, ACM 4th International Workshop on Language Descriptions, Tools and Applications (April 2004)
16. van den Brand, M.G.J., Heering, J., Klint, P., Olivier, P.A.: Compiling language definitions: the asf+sdf compiler, vol. 24, pp. 334–368. ACM Press, New York (2002)
17. Bachrach, J., Playford, K.: D-expressions: Lisp power, Dylan style (1999), <http://www.ai.mit.edu/people/jrb/Projects/dexprs.pdf> (accessed November 22, 2006)
18. Sheard, T.: Using MetaML: A staged programming language, 207–239 (September 1998)
19. Skalski, K., Moskal, M., Olszta, P.: Meta-programming in Nemerle (2004), <http://nemerle.org/metaprogramming.pdf> (accessed November 5, 2007)
20. Fleutot, F., Tratt, L.: Contrasting compile-time meta-programming in metalua and converge. In: Workshop on Dynamic Languages and Applications (July 2007)

# Adding Dimension Analysis to Java as a Composable Language Extension\*

## (Extended Abstract)

Eric Van Wyk and Yogesh Mali

Department of Computer Science and Engineering  
University of Minnesota  
Minneapolis, MN 55455, USA  
{`evw,yomali`}@cs.umn.edu

**Abstract.** In this paper we describe a language extension that adds dimension analysis to Java. Dimension analysis can be used to check that values that represent physical measurements such as length and mass are not used inconsistently. What distinguishes this work from previous work that adds dimension analysis to programming languages is that here the extension is implemented as a *composable* language extension. This means that it can easily be combined with other extensions, possibly developed by other parties, to create an extended implementation of Java with new features that address concerns from several different domains.

## 1 Introduction

Dimension analysis can be used to check that a computer program does not incorrectly use values that represent physical measurements. For example, it can ensure that a value representing a length is not added to a value representing a mass. This analysis may be extended to also take into account the units of measurements of these values and thus check, for example, that a length measurement in feet is not added to a length measurement in meters.

Modern programming languages rarely provide support for this type of analysis and it is the source of a number of highly publicized software failures. For example, in September 1999, the unsuccessful landing of the Mars Climate Orbiter on the surface of Mars was traced back to a software failure in which a measurement in English units was interpreted as a measurement in metric units by the navigation software [13].

This paper is not the first to add dimension analysis to a programming language. To mention just a few, Wand and O’Keefe [20] and Kennedy [11] add dimension inference and analysis to ML and House [9] add it to Pascal. The work presented here differs from these in that it adds dimension analysis to Java as a *composable* language extension. The others add dimension analysis by creating a new monolithic language that cannot be easily extended with some other

---

\* This work is partially funded by NSF CAREER Award #0347860, NSF CCF Award #0429640, and the McKnight Foundation.

features that may be desired. Our goal is to extend a language with dimension analysis in such a way that it is composable with other language extensions.

Our dimension analysis extension is implemented in the ableJ extensible language framework [17]. ableJ currently supports Java 1.4, but some aspects of Java 1.5 have been added. It is often the case in ableJ that the composition of the host language (Java) and several extensions can be done automatically. Thus, a programmer can direct the tools to build a customized language implementation for the host language extended with the unique set of extensions that he or she requires to handle the task at hand. This paper describes how dimension analysis can be implemented as a composable language extension. This extension follows the pattern described in previous work on ableJ and composable language extensions [17].

To get a sense of the type of dimension analysis that is supported by this extension, consider the sample program in Figure 1. The key features include the new type expressions for specifying dimensions in types. For example, the fields `len` and `wid` are defined with dimension type “`Dim<int, L>`” whose dimension expression `L` indicates that this is a length measurement and whose *representation type* specifies that this measurement value is represented as an integer. These type expressions share the syntax of Java generics but are implemented as new types; `Dim` is a new keyword, not the name of a parameterized class. Another feature to note is that arithmetic operators such as `+`, `*`, and others are overloaded for dimension types so that we can check that dimension values are added and

```
public class Sample {
    Dim<int, L> len, width, perimeter ;
    Dim<int, L^2> area ;
    public Dim<int, a b> product (Dim<int, a> x, Dim<int, b> y)
        { return x * y ; }
    public Dim<int, a> sum (Dim<int, a> x, Dim<int, a> y)
        { return x + y ; }

    void demo (int l, int w) {
        len = (Dim<int, L>) l ; // cast from primitive types
        wid = (Dim<int, L>) w ;
        perimeter = len + wid + len + wid ; // a valid sum
        Dim<int, T> t = (Dim<int, T>) 3 ;
        Dim<int, L/T^2> acc ; // an acceleration variable
        area = len * wid ; // a valid product
        len = sum(len, wid) ; // valid call to method sum
        area = product(len, wid) ; // valid call to method product
        len = sum(len, area) ; // invalid call to method sum
        len = len + area ; // a dimension error
        acc = len / (t * t) ; // an acceleration
    }
}
```

Fig. 1. A sample program using the dimension analysis extension

multiplied correctly. Assignment is similarly overloaded. An overloaded method call operator ensures that for any method call, the dimension variables (`a` and `b`) in the dimension expressions in methods `product` and `sum` are substituted for types in a consistent manner. In the second assignment to `area` in `demo` we check that the dimension variable `a` is instantiated to the same dimension expression, in this case `L`, in both of the input types and in the output type and that these instantiated dimension expressions match those in the types of the actual arguments. In processing this program, the extended language implementation will check that the dimension types are used correctly and translate the program to a pure Java program in which the dimension types are translated to their representation types. Although this example program is a bit contrived it highlights the key features of the language extension.

In Section 2 we describe the ableJ framework and the attribute grammar that defines Java. In Section 3 we describe the attribute grammar that defines the language extension that implements dimension analysis. In Section 4 we describe some related work and conclude in Section 5.

## 2 The AbleJ 1.4 Extensible Language Framework

In this section we briefly describe the ableJ extensible language framework. Java 1.4 and the dimension analysis language extension are implemented as attribute grammars written in the Silver [15] attribute grammar specification language. Many aspects of the ableJ grammar have been simplified for presentation reasons. Additional information about ableJ can be found in previous work [17].

Silver has many features beyond those originally introduced by Knuth [12]. These include higher-order attributes [19], collection attributes [3], forwarding [16], various general-purpose constructs such as pattern matching and type-safe polymorphic lists. Silver also has mechanisms for specifying the concrete syntax of languages and language extensions. It passes these specifications to Copper, our parser and scanner generator that implements context-aware scanning [18]. In this approach the scanner uses information from the LR-parser's state to disambiguate lexical syntax. The resulting scanner and parser are deterministic and also support the composition of language extensions. In this paper we will only present the abstract syntax of ableJ and the dimension extension.

Figure 2 presents a significantly simplified subset of the Silver specification of Java 1.4 which is used in our actual implementation. This grammar defines nonterminal symbols, terminal symbols, productions, and attributes. The nonterminals `Stmt`, `Expr`, and `Type` represent, respectively, Java statements, expressions, and type expressions that appear in the abstract syntax tree of a Java program. The nonterminal `TypeRep` is used in the symbol table (the attribute `env`) in bindings of names to type representations.

Synthesized attributes such as the pretty print attribute, named `pp`, are defined. The `pp` attribute has type `String` and decorates (`occurs on`) tree nodes of type `Expr`, `Stmt`, and others. An errors attribute is used to collect type errors, and later dimension errors, found in a program. Both of these attributes are

```

grammar edu:umn:cs:melt:ableJ14 ;
nonterminal Stmt, Expr, Type, TypeRep ;
terminal Id_t /[a-zA-Z][a-zA-Z0-9_]*/ ;

synthesized attribute pp::String occurs on Expr, Stmt, Type;
synthesized attribute errors::[String] occurs on Expr, Stmt, ...
synthesized attribute typerep::TypeRep occurs on Expr, Type ;
synthesized attribute hostStmt::Stmt occurs on Stmt ;
synthesized attribute hostType::Type occurs on Type ;

abstract production if_then  s::Stmt ::= cond::Expr  body::Stmt
{ s.pp = "if ( " ++ cond.pp ++ " ) {\n" ++ body.pp ++ "}\n" ;
  s.hostStmt = if_then( cond.hostExpr, body.hostStmt ) ;
  cond.env = s.env ;  body.env = s.env ;
  s.errors = case cond.typerep of
    booleanTypeRep() => [ ]
    | _ => [ "Error: condition must be boolean." ]
  end ++ cond.errors ++ body.errors ; }

abstract production add      e::Expr ::= l::Expr  r::Expr
{ e.pp = "( " ++ l.pp ++ " + " ++ r.pp ++ " )" ;
  attribute transforms :: [Expr] with ++ ;
  transforms := [ ] ;
  forwards to if length(transforms) == 1 then head(transforms)
    else if length(transforms) == 0 then exprWithErrors(
      ["Type error on addition, types not supported." ] )
    else exprWithErrors(["Internal compiler error."]) ; }

abstract production localVarDcl  s::Stmt ::= t::Type id::Id_t
{ s.pp = t.pp ++ " " ++ id.lexeme ++ "\n" ;
  s.defs = [ varBinding(id.lexeme, t.typerep ) ] ; }

abstract production boundId      e::Expr ::= id::Id_t  t::TypeRep
{ e.pp = id.lexeme ;
  attribute transforms :: [Expr] with ++ ;    transforms := [ ] ;
  forwards to if length(transforms) == 1 then head(transforms)
    else generic_boundId(id,t) ; }

abstract production booleanTypeExpr  te::TypeExpr ::= b::'Boolean'
{ te.typerep = booleanTypeRep() ; te.pp = "Boolean" ;
  te.hostType = booleanTypeExpr(b) ; }
abstract production booleanTypeRep  tr::TypeRep ::= { tr.pp="boolean"; }

aspect production add      e::Expr ::= l::Expr  r::Expr
{ transforms <- if match(intTypeRep(), l) && match(intTypeRep(), r)
  then add_int(l,r)  else [ ] ; }

abstract production add_int  e::Expr ::= l::Expr  r::Expr
{ e.typerep = intTypeRep ( ) ; e.hostExpr = add(l.hostExpr,r.hostExpr); }

```

Fig. 2. Highlights of a simplified Java host language attribute grammar

defined on the `if_then` production. The definition of the `errors` attribute uses pattern matching to check that the type (`typerep`) of the condition is Boolean. This production also passed the error messages from its children up the abstract syntax tree (AST).

The production `localVarDcl` creates a binding of the name of its identifier (`id.lexeme`) to its type representation (`t.typerep`) and passes this up the tree in the `defs` attribute. At the statement-block level (and others) this information is collected to form the symbol table and passed back down the tree in the inherited attribute `env`.

*Forwarding:* We have previously introduced *forwarding* [16] as an extension to attribute grammars that is useful in specifying languages in a highly modular way. To use forwarding, a production specifies (using the `forwards to` clause) how to build a new AST that will be queried for any attributes that are not explicitly defined by an equation on the “forwarding” production. This new AST, called the “forwarded-to tree”, can be seen as the *semantic equivalent* of the original forwarding AST. If the original tree does not explicitly define an attribute  $a$ , its value is automatically computed by copying it from the  $a$  attribute on the forwarded-to tree.

Forwarding is used by productions that define new language extensions to specify the semantically equivalent construct in the host language that they will “translate” to. The type expression “`Dim<int, L>`” used in the declaration of `len` and `wid` in the second line of Figure 1 will be represented in the program’s AST by a production that forwards to the type expression tree for `int`. As we will see below (Figure 4), the extension production will define some attributes to facilitate dimension analysis but it will not define any of the `hostNT` attributes that are used to translate the extended program to a semantically equivalent host language program. Each nonterminal  $NT$  in the host language is decorated by a synthesized attribute `hostNT` of type  $NT$  that holds a node’s translation to the host language. This attribute is defined only on host language productions and computes the translation using the host language production and the host language translations of its children (stored in their `hostNT` attribute) to compute this. This can be seen on the `if_then` production. When the dimension type expression tree for “`Dim<int, L>`” is queried for the value of its `hostType` attribute it will forward that query to the type expression tree for “`int`” which will return a copy of itself. In this manner, we can extract the translation of the extended program to the host language.

*Operator Overloading:* Operator overloading also uses forwarding and this can be seen in the production `add`. This production is used by the parser in constructing the original AST. It is a place holder that will forward to a new `Expr` tree constructed by a production specific to the types of the operands. In the simplified example in the figure, this production specifies a *collection attribute* [3] named `transforms` that is given an initial value (by the `:=` operator) of an empty



list. The *aspect production* for `add` near the bottom of the figure can remotely define values for attributes for the original `add` abstract production. In this case, it may add an `Expr` tree to the `transforms` list, using the `<-` operator, if the types of the operands `l` and `r` are both integers. The tree that it may add is constructed by the `add_int` production. This production defines the `errors` and `typerep` attributes for integer addition. When the original tree built by `add` is queried for its `typerep` attribute, for example, it forwards that query to the first tree in the `transforms` list. If `transforms` is empty, then there is a type error in the program; there is no implementation for `add` for the types of the child expressions. If there is more than one tree in `transforms` then an internal compiler error has occurred since more than one aspect production has specified an implementation for addition for the types of the child expressions. This can only occur when language extensions overload operators for host language types, which they rarely do. Extensions typically overload operators for the new types that they introduce; such overloading do not trigger this error and it is thus rare in practice. In Section 3 we will see that the dimension analysis extension overloads addition and other constructs in a similar fashion.

*Language Extensions:* In the following section we describe the attribute grammar specification of the dimension analysis language extension. This extension follows the pattern of other extensions to the ableJ framework. Extensions may introduce new language constructs, either by specifying new concrete syntax so that their abstract syntax is placed into the original AST, or by using operator overloading facilities so that a “place holder” production will forward to a production specified in the language extension. In either case, the productions defined in the extension will (and must) specify their transformation to a semantically equivalent host language construct using forwarding. In effect, the translation of the extended program to a host language program is carried out by the many local transformations that forwarding specifies. Different language extension constructs are not isolated from one another in the AST however since they can communicate when declarations add information to the symbol table that may be retrieved in other parts of the AST. The extension productions will do some semantic analysis (by explicitly defining some attributes). Forwarding is then used to specify their translation to the host language. The dimension analysis extension follows this pattern. The new productions perform the dimension analysis and then forward to pure Java constructs on which the dimension information has been translated away.

### 3 Dimension Analysis Extension

In this section we discuss some principles of dimension analysis and then show how dimension analysis can be added as a composable language extension to the ableJ specification of Java. This language extension defines new syntax for type expressions and overloads some existing host language syntax for arithmetic

operators, among others, to perform dimension analysis and ensure that measurement values are not incorrectly used.

### 3.1 Principles of Dimension Analysis

Dimensions describe a specific type of measurement. These include length, mass, temperature, among others. These are not to be confused with units that specify the unit of measurement for a particular dimension. For example, the dimension of length can be measured in units of feet or meters. Dimensions can be classified as *base dimensions* or *derived dimensions*. Traditionally, base dimensions include length, mass, temperature, time, electric current, amount of material and luminosity. Derived dimensions are specified in terms of these. For example, area is derived from the base dimension of length as length squared; acceleration is derived from length and time as length divided by time squared. Dimension expressions are generated from base dimensions and dimension variables using the operations of product and inverse. From these operations we can define division and exponentiation. A *unit* dimension is represented as 1. We represent dimension expressions in our language extension using the nonterminal `DimExpr` and the productions shown in Figure 3. A nonterminal and productions for base dimensions are also shown. The acceleration dimension expression  $L / T^2$  is represented by the tree `divide(basedim(L()), power(basedim(time()),2))`.

We will define a few functions on `DimExpr` trees for use in type checking. A `unify` function is used to unify two dimension expressions and if successful, return the set of substitutions for dimension variables that unifies them. This function will be used in checking that two expressions that have dimension types can be added or copied in an assignment. If the dimension expressions that are components of their types can be unified then it is safe to add or copy them. Note that multiplication of values of different dimensions is always allowed. The dimension of the resulting product is the product of their respective dimension expressions. If unit checking is added, then multiplication can fail if the operands use units inconsistently.

```

nonterminal DimExpr ;
abstract production product de::DimExpr ::= l::DimExpr r::DimExpr { }
abstract production divide de::DimExpr ::= l::DimExpr r::DimExpr { }
abstract production power de::DimExpr ::= b::DimExpr e::Integer { }
abstract production dimvar de::DimExpr ::= v::String { }
abstract production basedim de::DimExpr ::= bd::BaseDim { }
abstract production unit de::DimExpr ::= { }

nonterminal BaseDim ;
abstract production M bd::BaseDim ::= { } -- Mass
abstract production L bd::BaseDim ::= { } -- Length
abstract production T bd::BaseDim ::= { } -- Time

```

Fig. 3. Grammar for dimension expressions

### 3.2 Type Expressions for Dimension Analysis

*New type expressions:* We need new type expressions so that programmers can specify dimension types. Thus, an abstract production `dimTypeExpr` with the left hand side as the host language nonterminal `Type` is introduced in the extension. This production is shown in Figure 4. This production specifies that dimension types consist of a representation type `rep` (`int`, `float`, `Integer`, etc) and a dimension expression `d` that specifies the dimensionality of the values to be represented. This production defines a pretty print attribute and errors attribute as expected. The `typerep` attribute specifies the *representation* of the type. This is used by productions such as the `localVarDecl` production from Figure 2 to add information to the symbol table that binds variable names to types. The end result is that there are entries in the symbol table `env` that bind variable names to information about their type. The `dimTypeExpr` production passes the symbol table `env` down the tree to its components and also forwards to the representation type `rep`.

This extension does not check that the underlying representation types are type correct. Dimension analysis productions, such as `dimTypeExpr`, will forward to constructs on which the dimension types have been erased that do additional type checking on the translated version to ensure that the underlying representation types are used correctly. Such errors are reported to the programmer.

*Dimension Expressions:* Concrete syntax productions are used to parse dimension expressions like those shown in the `Dim` type expressions in Figure 1 to construct abstract syntax trees (ASTs) using the productions in Figure 3. These trees are the representation of the dimensions used in dimension analysis. We do not describe the specification of the concrete syntax here as it is what one would expect. A normalization function (`normalize`) converts these expressions into a normalized form that simplifies the dimension analysis. For example, `normalize` would simplify the dimension expression  $(L \ T) / (M \ T^2 \ M^{-1})$  to  $L / T$ .

*Type representations:* The host language nonterminal `TypeRep` is used for internal representations of types that are used in type checking in `ableJ`. For our dimension extension to fit into the `ableJ` framework we define productions that define type representations for dimension types. There are two such representations. The first production, `dimTypeRep_ST`, is used in the symbol table and

```

abstract production dimTypeExpr te::Type ::= rep::Type d::DimExpr
{ te.pp = "Dim<" ++ rep.pp ++ "," ++ d.pp ++ ">" ;
  te.errors := rep.errors ++ d.errors;
  te.typerep = dimTypeRep_ST(rep.typerep, normalize(d)) ;
  rep.env = te.env ;      d.env = te.env ;
  forwards to rep ;      }

abstract production dimTypeRep_ST tr::TypeRep ::= rep::TypeRep de::DimExpr
abstract production dimTypeRep_Ex tr::TypeRep ::= de::DimExpr reptree::Expr

```

Fig. 4. Type expression and representation productions

specifies the dimensionality and the representation type of variables declared as dimension types. The signature of this production is shown in Figure 4. This production is used in `dimTypeExpr` to define the type representation of the dimension type. The second, `dimTypeRep_Expr`, is used in type representations that decorate expressions. This one also specifies the dimensionality of the expression but instead of the type of the representation it specifies the tree that this expression will translate to. The type representation on this tree is the representation type. Thus, expressions (`Expr` trees) that have a dimension `typerep` will forward to this representation tree that is part of their type. This `TypeRep` production is used below in the `Expr` productions that overload arithmetic operators.

### 3.3 Overloading Existing Host Language Syntax

When introducing a new type, we often find it useful to overload the certain host language operations to provide type-specific behavior to these operations. For example, we will overload the host addition production `add` so that we can check that the dimensions of the operands are compatible on addition. This subsection describes several of the operators (productions) that are overloaded.

*Overloading variable references:* It is sometimes useful to overload the variable reference production so that variables that are bound to dimension types get their own type-specific production in the AST.

The `ableJ` infrastructure handles Java name disambiguation<sup>1</sup> and the looking up of names and binding them to their types. The abstract productions that perform this task are not relevant here. What matters is that they will forward to the production `boundId` shown in Figure 2. This production has a *collection* attribute called `transforms` that has the type `[Expr]`. Language extension will add new trees to this list if they want to overload a specific instance of a variable reference. This is similar to the way in which `add` is overloaded.

The aspect production at the top of Figure 5 from the dimension extension specifies that if the type bound to this identifier is a dimension type, then add the AST that is constructed with the bound identifier production specific to dimension types (`boundId_dims`) to the list of possible trees that the original `boundId` can transform to. If there is only one such tree, then the `boundId` production will forward to that and we effectively overload variable references with the `boundId_dims` production. This production, also in Figure 5, defines the type (`typerep` attribute) to be the dimension type that contains the dimension expression of the type (`dimexpr(t)`) and the tree that this will eventually translate to (`reptree`). The tree `reptree` is constructed with the `boundId` production but the type given to that production is the representation type of the dimension. As an example, consider the variable `len` in the example program in Figure 1. On the multiplication of `len` with `wid`, the `len` identifier in the original AST is overloaded using the production `boundId_dims` and sets its type to be a dimension type that contains a representation tree that is just the bound identifier

<sup>1</sup> This determines, for example, if “a” in “a.b.c” is a package, a class, or an object.

```

aspect production boundId  e::Expr ::= id::Id_t  t::TypeRep
{ transforms <- if match(dimTypeRep_ST(_,_), t)
                  then [ boundId_dims(id,t) ]   else [ ] ;  }

abstract production boundId_dims  e::Expr ::= id::Id_t  t::TypeRep
{ e.pp = id.lexeme ;
  e.typerep = dimTypeRep_Ex(dimexpr(t), reptree) ;
  forwards to reptree ;
  local attribute reptree :: Expr ;
  reptree = boundId(id, reptyperrep(t)) ;  }

```

**Fig. 5.** Overloading variable references

“len” that has as its type the type `int`. Thus, we essentially erase the dimension information in the translation to Java once we have verified that the dimension values are used in a correct manner.

*Overloading arithmetic operations:* In Section 2 we showed how addition can be overloaded with a type specific production. In Figure 6 are the aspect and dimension-type-specific productions that accomplish this for dimension types. The production `add_dims` will unify the dimension expressions from the type representations of the two operands `l` and `r`. If unification succeeds, this process returns a list of bindings (`bnds` of type `[UnifyBnd]`) that map dimension variables to dimension expressions that will unify the two dimension expressions and an empty list of errors. Otherwise an empty list of bindings and a list with one error message specifying that unification failed is returned. The type of `unify` is given in Figure 7. If the case of the addition in Figure 11 of `x` and `y` in method `sum` which both have the dimension expression `a`, the unification succeeds and returns no bindings since they are the same expression. In the case of the

```

aspect production add  e::Expr ::= l::Expr  r::Expr
{ transforms <- if match(dimTypeRep_Ex(_,_), l) &&
                  match(dimTypeRep_Ex(_,_), r)
                  then [ add_dims(l,r) ]   else [ ] ;  }

abstract production add_dims  e::Expr ::= l::Expr  r::Expr
{ e.pp = l.pp ++ " + " ++ r.pp ;
  local attribute bnds :: [UnifyBnd] ;
  local attribute errs :: [String];
  (bnds,errs) = unify ( get_dimexpr(l.typerep), get_dimexpr(r.typerep) ) ;
  e.typerep = dimTypeRep_Ex(apply(bnds, get_dimexpr(l.typerep)), rep_tree);
  forwards to if  null(l.errors ++ r.errors) then rep_tree
               else exprWithErrors(l.errors ++ r.errors) ;
  local attribute rep_tree :: Expr ;
  rep_tree = if  null(errs)
              then add (get_rep_tree(l.typerep), get_rep_tree(r.typerep))
              else exprWithErrors(["Dimensions incompatible on addition."]);}

```

**Fig. 6.** Overloading addition

```

function unify ([UnifyBnd],[Error]) ::= d1::DimExpr d2::DimExpr { ... }
function apply DimExpr ::= b::[UnifyBnd] de::DimExpr { ... }
function compose [UnifyBnd] ::= b1::[UnifyBnd] b2::[UnifyBnd] { ... }

```

**Fig. 7.** Function headers for unify, compose, and apply

additions that compute `perimeter` the dimension expressions are always `L` which also unify. If we were to unify dimension expressions `L` and `a` unification would succeed with the binding  $a \mapsto L$ . In these cases the bindings are applied (using the `apply` function) to the dimension expression of `l` to get the new dimension expression used in the `typerep` for the type of the sum. For the erroneous addition of `len` and `area`, the dimension expressions `L` and `L L` will not unify. In this case the dimension expression in the `typerep` is an erroneous dimension expression. Our extension uses the unification algorithm given by Kennedy in his work on extending ML with dimension analysis [11].

The tree that the `add_dims` production will forward to (`rep_tree`) is either the sum of the representation trees of `l` and `r` (constructed by the production `add`) or an erroneous tree indicating that the dimensions were incompatible. It is this tree that this production will forward to. When unification succeeds, `rep_tree` is simply the same expression in which the dimension information has been removed and the variables and expressions have the type of the underlying representation type instead of the dimension type.

Overloading multiplication is done in a similar fashion except that we need only generate the product of the dimension expressions of the operands since multiplication of any dimensions is valid.

*Overloading assignment and parameter passing:* The productions for assignment and method call can also be overloaded in a similar fashion. For many language extensions that introduce new types this is often not necessary however since, if these productions are not overloaded, they forward to trees that use the `copy` production whose signature is shown below:

```

abstract production copy e::Expr ::= s::Expr t::TypeRep { ... }

```

By overloading this production, an extension in essence overloads assignment, the copying of parameters into a method, and the copying of the return value back out. In the dimension extension we overload this production with a `copy_dims` production that unifies the dimension expressions on `s` and `t` to check that the dimensions are compatible. It is quite similar to `add_dims` and we thus do not show it here.

*Overloading method calls:* Although the `copy` production above would ensure that on method calls the dimension expressions of the formal and actual parameters unify individually, we must check that they unify in a consistent manner and that we provide consistent substitutions for dimension variables in all places. For example, the call to `sum` in Figure 1 with arguments `len` and `area` is incorrect because we must unify the dimension variable `a` to the same dimension

expression. Although copy would unify `a` to `L` and `a` to `L L` for each parameter individually this is not enough. Thus, we will overload the method call production with a type specific production for dimension types if any of the arguments have a dimension type. (In the case where only the return type is a dimension type the overloading of `copy` is sufficient.)

This new method call production calls the function `check_call` shown in Figure 8 and passes it the dimension expressions of the formal parameters and actual parameters that have dimension types. It also passes in an empty set of bindings. It will first check that for each parameter either both the formal and the actual parameter or neither have a dimension type. The `check_call` function calls `unify` on the first dimension expression in the `formals` and `actuals` lists after applying any previously discovered substitutions (`psubs`) to them. If unification succeeds, then it calls itself with the tails of the lists and the new substitutions (`new_sub`s) and the application of them to the dimension expressions in the previous bindings `psubs`.

In the case of the incorrect call to `sum` mentioned above, `check_call` will first unify `a` to `L` for the first parameter `len`. It will then pass this binding in `psubs` in the recursive call. On the second call, it first applies this substitution to `a`, the dimension expression of the formal parameter `y` to get `L`, and then applies it to `L L`, the dimension expression the actual parameter `area`, to get `L L`. It then attempts to unify `L` and `L L` and fails—thus indicating that the arguments to `sum` have incompatible dimensions.

If the call to `check_call` succeeds, the returned set of substitutions are applied to the dimension expression of the return type to get the dimension expression that is used in the type representation of the method call. In the case of the valid call to `sum` with parameters `len` and `wid`, the substitution mapping `a` to `L` is applied to the return type dimension expression `a` to yield `L`—the length dimension that is then correctly used in the assignment to `len`.

```
function check_call ([UnifyBnd], [String])
  ::= formals::[DimExpr] actuals::[DimExpr] psubs::[UnifyBnd]
{ return
  if      null(formals) && null(actuals)
  then   (psubs, [ ])
  else if null(formals) || null(actuals)
  then   (psubs, [ "incorrect number of arguments" ] )
  else if ! null(unify_errors)
  then   (psubs, [ "incompatible dimensions:" ++ head(formals).pp ++
                  " and " ++ head(actuals).pp ] )
  else   check_call( tail(formals), tail(actuals),
                    new_sub ++ compose(new_sub,psubs)) ;
local attribute new_sub :: [ UnifyBnd ] ;
local attribute unify_errs :: [ String ] ;
( new_sub, unify_errs ) = unify ( apply(psubs,head(formals)) ,
                                apply(psubs,head(actuals)) ) ; }
```

Fig. 8. The `check_call` function

### 3.4 Composing AbleJ and the Dimension Analysis Language Extension

The attribute grammar fragment presented above in this section defines the dimension analysis constructs and analysis needed to extend the ableJ attribute grammar specification of Java with dimension analysis. The process of composing these two attribute grammars is performed by the Silver tools. A component-wise union of the sets of nonterminals, terminals, productions, and attribute definitions on productions is straightforward and implements the composition of the host language and the language extension. This is easily extended to handle more than one language extension. Since this composition is performed by the Silver tools, non-experts can specify a host language and a desired set of language extensions that can be automatically composed to create a specification of a unique language that has features tailored to a particular task at hand. For more information on the composition process readers are directed to our previously published paper on this topic [17].

## 4 Related Work

*Dimension analysis:* Previous research in this area has illustrated different techniques for adding dimension and unit analysis to programming languages. House [9] implemented the extension of dimensions to Pascal. He implemented a polymorphic dimension type system in the monomorphic type system of Pascal. Wand and O’Keefe [20] designed dimensional inference in an ML-like type system. In their extension, they extended single numeric types parameterized on dimension. They assumed a fixed number of base dimensions. Delft [5] extended Java with dimension analysis in a monolithic way. Kennedy [11] has implemented dimension extension to Standard ML programming using the ML’s capabilities of polymorphism and type inference. It is his dimension unification algorithm that is used in our extension.

Allen et al. [2] provide a solution that differs from ours and the others described above. They first add meta-programming facilities (meta-classes) to an extension of Java called MixGen. They then use those to implement dimension and unit analysis. Their extension of dimension types integrates well with generics and subtypes, something not investigated in our extension.<sup>2</sup> This approach also supports composition of extensions since a programmer can use meta-classes from different libraries. It does not however allow the extension writer to create new syntactic constructs that reflect the notation of the domain. This is less critical for extensions that add new numeric types for dimension analysis, but it is critical for language extensions that, for example, extend Java with SQL to support static checking for syntax and type errors [17].

*Extensible languages:* Mechanisms for implementing programming languages have been an active research area for many years and thus there is much work

---

<sup>2</sup> ableJ supports extensions that add new types and sub-type relationships [17].



on this topic in general and, more specifically, on the topic of extending Java with different language features. We are thus necessarily very brief here. A more complete description of related work can be found in previous work [16,17].

In the attribute grammar community there have been many investigations into the modular specification of languages [1,7,8,10,14], to cite just a few. Of particular interest is the JastAdd [6] system and its implementation of Java 1.5. Where as we use forwarding to add a form of (non-destructive) transformation to attribute grammars JastAdd adds destructive rewriting. Both allow for the implicit specification of semantics (that is attributes) through some transformation technique. Destructive rewriting has the advantage of not keeping both trees in memory at the same time and thus uses less memory than forwarding. It can also be used for traditional optimizations implemented as rewrite rules. Forwarding, on the other hand, has the advantage of allowing attributes to be computed on either the original extension AST or the forwarded-to host language AST. This supports a more modular specification of languages.

JavaBorg is an extensible Java tool that uses MetaBorg [4], an embedding tool that allows one to extend a host language by adding concrete syntax for objects. It is based on term rewriting and uses conditional rewriting of the AST to process programs. Thus, one must encode semantic analysis as rewrite rules.

## 5 Conclusion

There are several features that we are in the process of adding to our extension. The first adds the notion of units so that these can also be checked. This extension is straight forward since unit specifications like `ft` (for feet) might be used instead of `L` in dimension expressions. The unification algorithm must be extended to track units and multiplication must be checked for consistent use of units as well. A second may add the automatic conversion of non-dimension types (like `int`) to appropriate dimension types. Although more convenient, this is less safe than the current implementation. In extending ableJ 1.4 with Java 1.5 features we will investigate how new extension-introduced types such as the dimension types presented here can integrate with Java generics.

The language extension presented here adds dimension analysis to Java. Although it is less complete than some previous work on dimension analysis, it is distinguished in that it is added to Java as a *composable* language extension that also introduces new syntactic constructs. Thus, several extensions may be simultaneously added to Java so that the composed language contains new language features from more than one domain.

## References

1. Adams, S.R.: Modular Grammars for Programming Language Prototyping. PhD thesis, University of Southampton, Department of Elec. and Comp. Sci., UK (1993)
2. Allen, E., Chase, D., Luchangco, V., Maessen, J.-W., Guy, J., Steele, L.: Object-oriented units of measurement. In: Proc. Conf. on Object-oriented programming, systems, languages, and applications (OOPSLA), pp. 384–403. ACM, New York (2004)

3. Boyland, J.T.: Remote attribute grammars. *J. ACM* 52(4), 627–687 (2005)
4. Bravenboer, M., Visser, E.: Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In: *Proc. ACM Conf. on Object-oriented programming, systems, languages, and applications (OOPSLA)*, pp. 365–383. ACM, New York (2004)
5. Delft, A.V.: A java extension with support for dimensions. *Software-Practice and Experience* 29(7), 605–616 (1999)
6. Ekman, T., Hedin, G.: The Jastadd extensible Java compiler. In: *Proc. Conf. on Object oriented programming systems and applications (OOPSLA)*, pp. 1–18. ACM, New York (2007)
7. Farrow, R., Marlowe, T.J., Yellin, D.M.: Composable attribute grammars. In: *19th ACM Symp. on Prin. of Programming. Languages*, pp. 223–234 (1992)
8. Ganzinger, H.: Increasing modularity and language-independency in automatically generated compilers. *Science of Computer Programming* 3(3), 223–278 (1983)
9. House, R.T.: A proposal for an extended form of type checking of expressions. *The Computer Journal* 26(4), 366–374 (1983)
10. Kastens, U., Waite, W.M.: Modularity and reusability in attribute grammars. *Acta Informatica* 31, 601–627 (1994)
11. Kennedy, A.: Dimension types. In: Sannella, D. (ed.) *ESOP 1994. LNCS*, vol. 788, pp. 348–362. Springer, Heidelberg (1994)
12. D. E. Knuth. *Semantics of context-free languages. Mathematical Systems Theory*, 2(2):127–145, 1968; Corrections in 5, 95–96 (1971)
13. NASA. Mars climate orbiter - mishap investigation report. Technical report (November 1999),  
[ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MC0\\_report.pdf](ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MC0_report.pdf)
14. Saraiva, J., Swierstra, D.: Generic Attribute Grammars. In: *2nd Workshop on Attribute Grammars and their Applications*, pp. 185–204 (1999)
15. Van Wyk, E., Bodin, D., Krishnan, L., Gao, J.: Silver: an extensible attribute grammar system. In: *Proc. of LDTA 2007, 7<sup>th</sup> Workshop on Language Descriptions, Tools, and Analysis* (2007)
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., Krishnan, L., Schwerdfeger, A., Bodin, D.: Attribute grammar-based language extensions for Java. In: *Ernst, E. (ed.) ECOOP 2007. LNCS*, vol. 4609, pp. 575–599. Springer, Heidelberg (2007)
18. Van Wyk, E., Schwerdfeger, A.: Context-aware scanning for parsing extensible languages. In: *Intl. Conf. on Generative Programming and Component Engineering (GPCE)*. ACM Press, New York (2007)
19. Vogt, H., Swierstra, S.D., Kuiper, M.F.: Higher-order attribute grammars. In: *ACM Conf. on Programming Language Design and Implementation (PLDI)*, pp. 131–145 (1990)
20. Wand, M., O’Keefe, P.: Automatic dimensional inference. In: *Computational Logic - Essays in Honor of Alan Robinson*, pp. 479–483 (1991)

**Part III**  
**Participants Contributions**

# Model Transformations for the Compilation of Multi-processor Systems-on-Chip

Éric Piel, Philippe Marquet, and Jean-Luc Dekeyser

INRIA Lille – Nord Europe & LIFL, University of Lille, France  
e.a.b.piel@tudelft.nl, {philippe.marquet,jean-luc.dekeyser}@lifl.fr

**Abstract.** With the increase of amount of transistors which can be contained on a chip and the constant expectation for more sophisticated applications, the design of Systems-on-Chip (SoC) is more and more complex. In this paper, we present the use of model transformations in the context of SoC co-design. Both the hardware part and the software part of a SoC can be represented as a model using the MARTE standard from the OMG. We introduce the use of Model-Driven Engineering in order to generate executable code from a self-contained model of SoC.

First, we detail the restrictions and extensions we have brought to the MARTE profile in order to permit the complete description of the SoC as a model.

The compilation is a sequence of small and maintainable transformations that allows to pass gradually from a high-level description into models closer in abstraction to the final model, which is then converted into code. An in-depth view of one of the several transformation chains composing our tool is given. The implementation relies on the use of our experimental Java-based transformation engine which uses a hybrid declarative-imperative language.

We later discuss why model transformations fit better the compilation of the SoCs than traditional compilers. In particular, the re-use of transformations can greatly help with the fast evolution of SoC design, allowing development time reduction. Additionally, as each rule is small and relatively self-contained, their correctness is easier to ensure, which leads to more reliable compilation and indirectly more reliable SoCs.

## 1 Introduction

At the same time as advances in technology allow to integrate more and more transistors on a single chip, the embedded system applications get always more sophisticated. Although these two evolutions fit well in term of computation power, the combination put a strong pressure on the designers' capacity to design and verify the resulting very complex systems. As the International Technology Roadmap for Semiconductors has highlighted [1], there is a strong need for enhancing the design productivity. New design methodologies have to be adopted for the development of these large and complex Systems-on-Chip (SoCs).

A SoC contains on a single chip all the components of a computer: memory, processor, interconnection network, A/D and D/A converters... With the size

increase of chips, it is possible to put more components in a SoC. Additionally, due to physical restrictions in terms of frequency and voltage, to expand the processing power it is not possible to simply increase the size of the processor. It is necessary to put *several* processors in the system. Requiring from the software developers to handle the parallel programming paradigm in addition to the traditional concerns. Typically, the embedded systems are used in areas like multimedia (such as video encoding/decoding, HDTV), detection systems (such as radars, sonars), or telecommunications (such as mobile phones, antennas). All these applications are inherently multidimensional data flow applications.

In this paper, after introducing the specificities of SoC design, we will mention different approaches proposed until now for improving the productivity. Then we will give a brief description of a possible usage of Model-Driven Engineering in this context by presenting our development environment. The description of the metamodel for the specification will be followed by a close look at several model transformations allowing the compilation of a SoC model into simulation code. Then, mainly based on the acquired experience during the development of the presented transformations, we will highlight the benefits of model transformations for this particular purpose.

### 1.1 SoC Co-Design

One of the particularities of SoCs is that they are built for one specific application. Each new application leads to the design of a new architecture and new software, both exactly fitted to the task and specifically adapted to each other. Another particularity is that the initial cost for realization on the silicon (the creation of the mask of the chip) is very high, this mostly forbids the usage of prototypes. The SoC developers have to rely on simulations to test and verify their design.

The development of a SoC usually consists in the concurrent design of both the application and the hardware architecture, as illustrated in Figure 1. Each part is handled by different people, specialized on one of these domains. Then the application is mapped on the hardware, during the phase of association. This leads to generations of simulations of the full system. These simulations of both the hardware and the software together vary depending on the level of

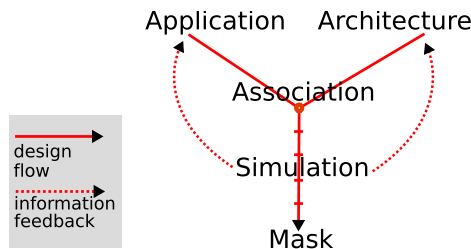


Fig. 1. Overview of the usual SoC development organization

abstraction. From the simulation results, the SoC designers can correct the SoC specification (the application, the architecture or the association) and obtain a new simulation. This is represented by the *information feedback* arrows on the figure. Gradually, the abstraction levels used for the description and the simulation are reduced in order to obtain more accurate observations.

At first, simulations at high levels of abstraction are used in order to rapidly obtain results about the system behaviour. Although there do not exist universally accepted levels of abstraction, typically those levels highly abstract the communications between the hardware components as well as their inner behavior between each cycle [2]. In some cases, the application is also abstracted such as proposed in [3] by using a special OS layer designed for the simulator or as proposed in [4] by only executing once each phase composing the application execution. The lowest abstraction level is usually the RTL (Register Transfer Level), from which the SoC can be synthesized. This journey through the representation of the same system at successive abstraction levels is typical of the SoC development. Classically, each time the abstraction level is reduced, the simulation has to be entirely re-written.

## 1.2 Related Works

In order to speed up the design flow, several methodologies have been proposed. The *system synthesis* methodology consists in transforming through successive refinements the original sequential specification into a concurrent specification defining all the implementation details. This relies on the use of high-level languages. At the beginning the system is represented only as a network of processes following a specific Model of Computation (MoC) such as KPN [5] (Kahn Process Network) or SDF [6] (Synchronous DataFlow). It is then rewritten in languages targeted toward hardware specification such as SystemC [7] or SpecC [8]. These languages allow to gradually specify the architecture part of the system with different levels of refinements down to a physical description. Such approaches have been proposed through projects such as COSMOS [9], Chinook [10], or Specsyn [11]. They have been the first approaches proposing automated transformations from a high abstraction level to a lower one. However, one of the main drawback was that the transformations were not complete, various tasks had to be done manually.

Another approach called *platform-based design* [12] aimed at reducing the work of the designers by supplying a parametrizable architecture. A specific tool allows the SoC designer to compose and configure a hardware platform in order to adapt it to the specific requirements of the application. The software is expressed using a high-level API. Tools such as VCC by Cadence [13], or N2C by Coware [14] offer such kind of platform. However, the platform provided by the tool is often specific to a particular application and targeting a different domain requires the costly introduction of a new platform into the tool by its makers.

The approach called *component-based design* [15,16] strives to provide the advantages of the two previous presented methodologies. The designers describe the whole SoC as a hierarchical network of virtual components and communication

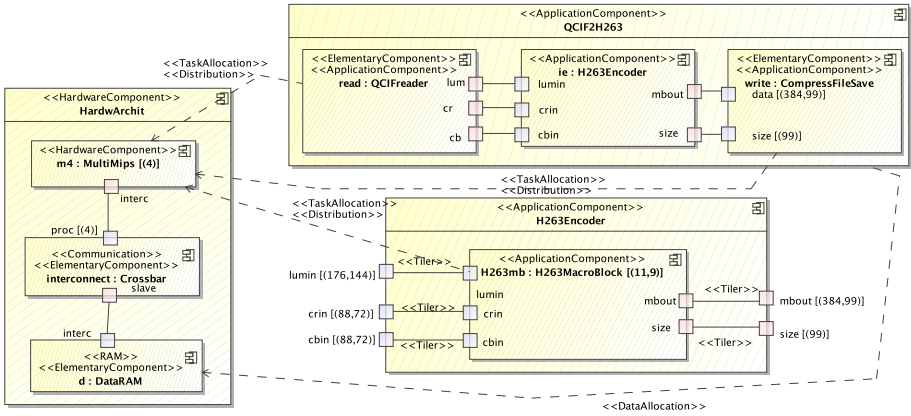
channels. Each virtual component is a primitive internally specified at a low level of abstraction, typically at the RTL level for the hardware components and C or assembly language for the software components. The interconnection of the components is done via *wrappers*. This bottom-up approach allows designers to reuse efficient custom solutions with best performances. Unfortunately, the abstraction level at which the system is initially defined is limited, requiring from the developer to have already defined which part is hardware and which one is software and to write the system as a code.

Recently some propositions have appeared suggesting the usage of *Model-Driven Engineering* [17] in the specific context of SoC design. Early propositions have focused on the *modeling* side of this approach. In particular, several UML profiles dedicated to the representation of such systems have emerged. However, either they display the same problem as UML which has too many variation points to allow a complete specification only at the model level, such as SysML [18], or they tend to be very specific to one given implementation language, such as the standardization proposal by Fujitsu [19] which is very tied to SystemC.

Some works are starting to appear on the usage of the second side of this approach: the *model transformations*. They highlight the need for model notation to have an entirely *executable* semantics [20], that is not having any semantic variation points and containing information so that each part of the system can be completely realized. Unfortunately, so far the propositions [20,21,22] have been limited as direct transformations from UML profiles to SystemC simulations. Similarly, in [23] a model transformation is used to pass from a high level model of the software to a compilable level. In our project called Gaspard [24], we went further in the usage of MDE by abstracting more the level of specification of the SoC and leveraging the model transformations to target the multiple types of outputs that might be needed during the SoC design. Moreover, each target is obtained not by the execution a one big transformation but by a chain of smaller and maintainable transformations.

## 2 Executable Models of SoC

In our project Gaspard the specification of the SoC is done exclusively via models. These models conform to the new UML profile called MARTE [25] standardized by the OMG. The MARTE profile *consists in defining foundations for model-based description of real-time and embedded systems*. It is specifically designed to permit description of both the hardware and the software parts of those systems. From the user point of view, using this profile has the benefits of using a standard representation (in other words, not being tied to a vendor specific representation) and of providing a high abstraction level, which is not directly associated to an implementation and more closely fits the generic concepts manipulated during SoC design. Additionally, one particular point of interest in our case for developing multi-processor SoCs (MPSoC) is the introduction of concepts



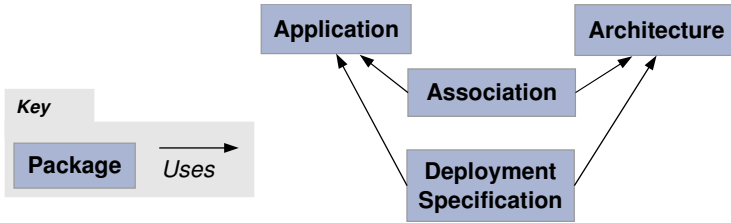
**Fig. 2.** Overview of a quadri-processor SoC with an association of an H.263 encoder application

for Repetitive Structure Modeling [26]. It allows to represent in a compact way both parallel architectures and parallel applications.

To use the MARTE profile as a means to represent a fully *executable system*, additional concepts and restrictions have been introduced as a supplementary profile. The necessity is twofold: first, the semantic variation points have to be eliminated to allow a non-ambiguous interpretation by the model transformations, and second, all the implementation details (down to the complete code of each function) have to be specified. In the Gaspard supplementary profile, the first point has been addressed by defining additional strict semantics on each package of the profile. For instance, the hardware components are defined to be represented only via the notions of components, ports and connectors, each of them having a precise meaning on the implementation. On the application side, while in MARTE the behaviour can be represented in ways nearly as broad as in UML, in Gaspard we have restrained the set of notions to a model of computation (MoC) based on ArrayOL [27]. It focuses on the expression of data flow applications with all their data and task parallelism. It is particularly well suited to the context of Gaspard that aims to design intensive signal processing applications. This is not a restriction of MDE: if the need arises to support a more generic context, another, less specific, MoC would have to be employed.

As an example, an overview of an MPSoC model using this profile is provided in Figure 2. On the left side is the main component of the architecture (four processors, a memory, and a crossbar). On the right side are the first two levels of an application dedicated to H.263 video encoding. Multiplicity (the numbers between brackets) allows to specify the repetition of a component in a compact and explicit way. In this example it is used for representing the four processors, as well as the  $11 \times 9$  repetitions of the task *H263mb* (each repetition processing a different part of the picture). Not only this allows the designer to easily modify the configuration of the system, but it also permits to express the parallelism of





**Fig. 3.** Main packages used during an MPSoC design with Gaspard

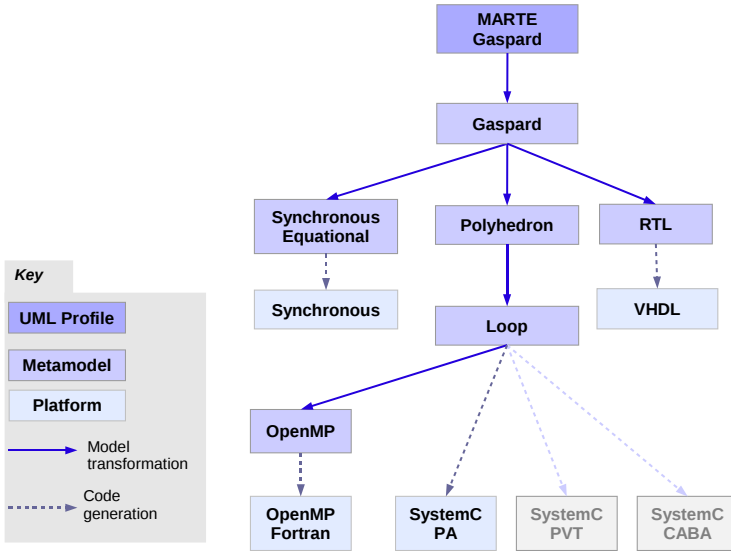
the application. Stereotyped dependencies specify the distribution of tasks of the application on the processors and the distribution of the data on the memory.

To address the second point required to specify an executable system, a package of the Gaspard profile called *deployment specification* permits to link the elementary components (which can be considered as “black boxes”) to source code. This mechanism can be used both for the architecture and the application components. In addition, special concepts allow to precisely map the interfaces of the components to the input and output of the functions. For a given functionality (e.g., Fast Fourier Transform, MIPS processor) several implementations can be provided, in different languages, or abstraction levels. With the creation of component libraries, this simplifies the SoC designer work as the deployment specification can be selected only once for each component and reused for each compilation target.

Figure 3 illustrates the relationships between the main packages used for the specification of an MPSoC in Gaspard. The two packages at the top define the architecture and the application. The deployment specification package introduces additional information concerning the implementation details, while the association package permit the mapping of the application on the architecture. Each of these packages correspond to a specific task during the design of the SoC. In particular, there is no dependency between the architecture and the application: it is possible to work on them concurrently.

### 3 Model Transformations for MPSoC Compilation

From the MPSoC model Gaspard provides several transformation chains. As output of a transformation chain, the user expects compilable code which can be used in already available tools. The Gaspard environment permits to select a *target* into which the SoC should be transformed. The most obvious target is a synthesizable hardware description and application code compilable for this particular hardware. As shown in Figure 4, other target possibilities encompass synchronous specification for formal verification, and simulations of the MPSoC at various abstraction levels. Each chain is a sequence of several model transformations separated by metamodels and finished by a code generation. For now the two code generations leading to *SystemC/PVT* and *SystemC/CABA* have not yet been fully implemented.



**Fig. 4.** The Gaspard compilation chains. From an MPSoC model in UML, indicated here as *MARTE/Gaspard*, each chain leads to a different target.

In the following subsections we will give an overview of the implementation of the transformations and the transformations chains. Interested readers are invited to refer to the Gaspard website [24] for downloading the environment (with the four transformation chains working) and examples of SoC models.

### 3.1 Implementation of the Transformations

Following the MDE recommendations, most of the transformations have a model as input and produce a model as output. They are called *model-to-model transformations*. In order to generate code, the final transformation of a chain takes a model as input but produces text as output. Such a transformation is called *model-to-text transformation* (we sometimes use simply *code generation*).

Figure 5 presents the organization of a model-to-model transformation. A point to emphasize is that both the input and output are clearly defined by the metamodels to which they conform to. Using the declarative language approach, each transformation is actually an organized set of rules. Each rule works on a small part of the input metamodel specified as a pattern called the *left hand side* and produces a small part of the output metamodel called the *right hand side*.

The transformation is executed using a transformation engine. All the model-to-model transformations in Gaspard have been implemented for MoMoTE, a transformation engine developed by the team based on EMF [28] (Eclipse Modeling Framework). Each transformation has one top-level rule which is called initially. It usually matches the root component of the input model. From this rule, all the other rules are called (directly or indirectly).

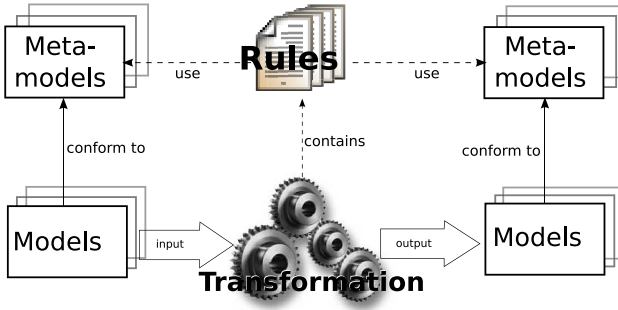


Fig. 5. Organization of a model-to-model transformation

Figure 6 presents an example of a transformation rule. It converts the concept of `HardwareComponentInstance` of the *Gaspard* metamodel into an equivalent concept of the *Polyhedron* metamodel. In MoMoTE, each rule is expressed as a Java class which contains five methods:

- The constructor for defining the sub-rules, and where the elements created by the sub-rules should be added. This is done via the `addRule()` method. In the example, it delegates the transformations of the *shape* and the *portInstances* to sub-rules.
- `getCondition()` for defining the left hand side using EMFT Query syntax (a part of EMF). In the example, the query looks for every `HardwareComponentInstance`.
- `create()` for defining the base element of the right hand side. In the example, it defines the right hand side as a `HardwareComponentInstance` of the *Polyhedron* metamodel.
- `process()` expresses how the new elements have to be created depending on the input. It is called once for each input element processed. In the example rule, it copies the name.
- `processReferences()` expresses how the references between the elements are produced depending on the input. In the example, it creates an equivalent reference as in the original model but pointing to the element created via another rule.

The last two methods are written with the usual Java imperative syntax, which confers a strong expressivity power to the rules. That is the place where the domain specific algorithms are situated. Even if for brevity, we have presented a rather straightforward rule, in the transformations that we have developed, one can find rules doing graph re-organization, static task scheduling, conversion between linear system representations, etc. Using Java as the underlying platform also proved to be advantageous for easily interfacing with external libraries or programs implementing a specific conversion.

These points are one of the reasons we used a Java-based transformation engine. Another reason is that at the beginning of this work, higher level engines

```

public class GaspardHI2PolyhedronHI extends Rule{

    public GaspardHI2PolyhedronHI()
    {
        super();
        setQueryFromContext(true);
        addRule(PolyhedronPackage.eINSTANCE.getComponentInstance_Dim(),
                new Gaspard2Shape2PolyhedronShape());
        addRule(PolyhedronPackage.eINSTANCE.getComponentInstance_PortInstance(),
                new Gaspard2PortInstance2PolyhedronPortInstance());
    }

    protected EObjectCondition getCondition(EObject srcElementContext)
    {
        return new EObjectTypeRelationCondition(
                Gaspard2Package.eINSTANCE.getHardwareComponentInstance());
    }

    protected EObject create(EObject srcElement)
    {
        return PolyhedronFactory.eINSTANCE.createHardwareComponentInstance();
    }

    protected void process(EObject srcElement, EObject tgtElement)
    {
        HardwareComponentInstance hci=(HardwareComponentInstance) srcElement;
        gaspard2.metamodel.polyhedron.HardwareComponentInstance shci=
            (gaspard2.metamodel.polyhedron.HardwareComponentInstance) tgtElement;
        shci.setName(hci.getName());
    }

    protected void processReferences(EObject srcElement, EObject tgtElement)
    {
        HardwareComponentInstance hci=(HardwareComponentInstance) srcElement;
        gaspard2.metamodel.polyhedron.HardwareComponentInstance shci=
            (gaspard2.metamodel.polyhedron.HardwareComponentInstance) tgtElement;
        shci.setComponent(((gaspard2.metamodel.polyhedron.HardwareComponent)
            getTransformation().getGlobalRefs().get(hci.getComponent())));
    }
}

```

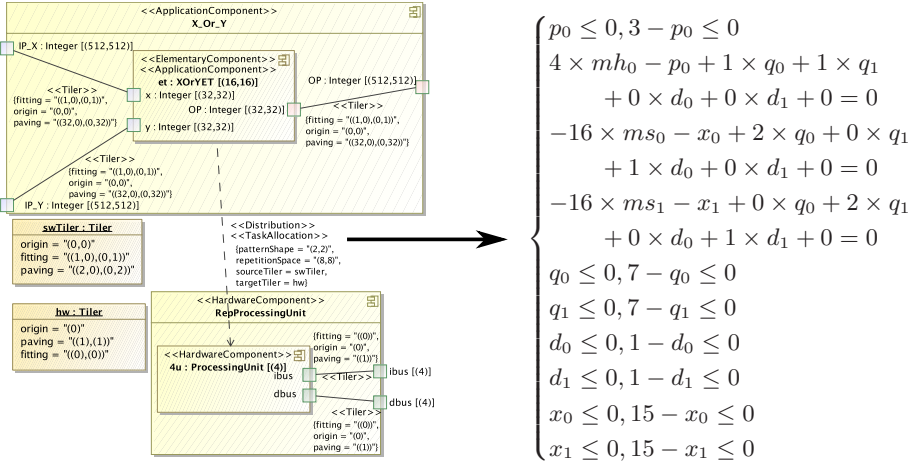
**Fig. 6.** Example of transformation rule using MoMoTE

such as TrML [29], or QVT [30] (Query, View, Transform) were not ready to be used. On the long term these kind of transformation engines will likely ease the development of the transformations.

In Gaspard, the code generations are executed using another engine, called MoCodeE. This engine, also developed internally, can be seen as a layer over JET [31] (Java Emitter Templates). It allows to associate one (or several) template for each class of the input metamodel. The transformation is then represented as a set of templates which are called depending on the type of the elements in the input model. Each template is the text as it should appear in the output intermixed with Java code.

### 3.2 Example of Transformation Chain

As an example, we will describe here the transformation chain towards SystemC/PA. The PA level is a very high-level of abstraction of the simulation which permits a quick simulation and allows the user to see the execution of the program in



**Fig. 7.** Example of input and output of one of the rules of the Gaspard to Polyhedron transformation

term of pattern usages, the data element of a Gaspard program, instead of reads and writes of bytes. More information on this level can be found in [32].

The first transformation of the chain is one common to all the chains. It takes as an input the UML model conform to the MARTE/Gaspard profile, created by the SoC designer, and outputs a model conform to a metamodel featuring the same concepts than in the profile. Deliberately, this transformation is limited to a simple *translation*: this allows to convert the standardized view in UML which benefits the user into the representation specific to the domain in the metamodel which is much easier to manipulate in the transformations (because the only required concepts are present, there is no need to go through stereotype mechanism of UML, etc.).

The second transformation, from the Gaspard metamodel to the Polyhedron metamodel is composed of approximately 60 rules. They mainly allow us: to express the repetitions as polyhedrons, to separate the application tree following the association specification, to map the data arrays on the memories, and to simplify the deployment specifications. A polyhedron is a set of linear equations and inequations. Works on parallel scheduling often rel because –explained very rapidly– they permit to compute which iteration of a loop should be executed on which processor. A whole set of theories *and tools* are already available.

Figure 7 illustrates the effect of the rule producing a polyhedron from a distribution of a task on a processor. On the left side is the information as defined by the user, on the right side is a text representation of the polyhedron computed by the rule. The rule is called for every *ApplicationComponent* which has one (or several) *Distribution* starting from it. Using the various information specified on the *ApplicationComponent*, the *HardwareComponent*, the *Tilers*, and the *Distribution*, a set of equations and inequations are generated. It is then inserted as a specific attribute of the element generated out of the *ApplicationComponent*.

The third model transformation of the chain, from the Polyhedron metamodel to the Loop metamodel, converts the mapping expressed by the polyhedrons into pseudo-code expressions, as used by the code implementations. Each polyhedron is transformed into nested-loops. Even if theoretically this transformation could be merged with the previous one, we have decided to separate them for technical reasons: the computation is done via the call to an external program, CLoog [33], it is easier to maintain and troubleshoot this call independently from the rest of the transformations.

The final transformation of the chain is a SystemC code generation from the Loop metamodel. Based on the usage of templates as described previously, it generates both the simulation of the hardware components and the application components. Each hardware component is transformed into a SystemC module with its ports linked. For each processor, the part of the application which has to be executed on this processor is generated as a set of *activities* dynamically scheduled and synchronised, following the model of execution defined for the Gaspard applications on MPSoCs. Additionally, the framework needed to automatically compile all the simulation code is also generated (as a Makefile).

## 4 Advantages of Model Transformations for SoC Compilation

The usage of models for the design of multi-processor SoC is on its own a great improvement over current practice because it provides a higher abstraction level that especially helps for component reuse and parallel coding. The graphical representation also facilitates the global vision of complex systems and of interactions between the parts of the system. As shown in [34], model-based approaches help the system designers to reuse preexistent works and to adapt it to new applications, increasing their productivity. Nonetheless MDE transformations also bring their batch of benefits into the SoC co-design when used for the compilation flow.

In Gaspard, the use of transformations permits to generate the various abstraction levels of the SoC out of the same model. This relieves the designer from manually re-writing the system each time a lower level of abstraction is targeted. In turn, this allows the designer to explore more configurations of the system to find one as close as possible from the optimal.

As Alanen et al. have emphasized in [21], one benefit of the introduction of transformations is the break down into small parts of the SoC compilation which can be more easily understood by the *SoC designer*. In contrast to custom monolithic tools where the meaning of the SoC model is only associated to the final output of the code generator, the compilation chains bring transparency. This transparency is useful for the users, as it helps to grasp the meaning of the concepts used for the SoC design at the different levels of compilation and see their evolution until the generated code.

The compilation flow is a chain of several small transformations. Of course, this can also be achieved with traditional methods, but the MDE simplifies the

separation between each transformation. Transformation inputs and outputs are formally and explicitly described by metamodels. Each intermediate metamodel is a strongly documented “synchronization point” of the compilation flow, which is complex to conceive within traditional compilers. The ease of expressing intermediate representations leads to the benefit of maximizing the *reuse* of transformations while compiling toward different targets. As in SoC compilation several targets are always necessary, at least for the different abstraction levels of simulation, and each compilation may have strong common points with the other compilations, the reuse possibility is very high. This is illustrated in the overview of our compilation chain available in Figure 4. The first transformation is reused across all the six chains and, similarly, the two successive transformations between the Gaspard metamodel and the Loop metamodel are shared by four chains. For the different SystemC targets, the chains vary only on the code generation. This minimizes the work required to create a transformation chain towards a new target and favors the use of already-validated transformations.

Additionally, the *explicit* separation between the various stages of compilation permits to easily share the development work of the compiler among several developers while maintaining the coherency of the global project. For instance, in our case, the current compilation flow of Gaspard have been carried out simultaneously by up to seven persons.

Moreover, model transformations can be written in a declarative way: the transformation is a set of mostly independent rules which have explicit declarations of their input and output patterns. In SoC design, the hardware is as versatile as the software. Likewise, the employed technologies evolve very quickly between the development of two products. For example, a few years ago all the SoCs were mono-processor. With the need to handle multi-processors, tools had to be adapted to manage the additional specificities introduced both on the hardware and the software parts. As another example, new abstraction levels for the simulation are proposed in order to accelerate the simulation as the size and complexity of the systems increase, such as described in [2]. The compilation flows have to be modified to also permit code generation at those higher abstraction levels. This fast evolution is typical of the SoC design. During its existence, the compilation flow is not static but must constantly adapt to the technology evolutions, be updated with the evolution of the standards used to represent the SoC, and has to integrate the improvements proposed by researchers on the already existing algorithms. Writing transformations in a declarative language increases their maintainability and simplifies their modification because it is easier to identify which part of the model they affect. This flexibility of evolution of the compiler speeds up the development of a SoC which is based on novel technologies.

In addition, independence in-between the rules is advantageous for reuse as they can be easily separated and regrouped. In Gaspard, the rules of the transformation from the Gaspard metamodel to the Polyhedron metamodel used to simplify the deployment specification have been integrated mostly as-is in the compilation chains towards VHDL and Synchronous languages.

Furthermore, correctness and reliability of the compiler are important factors in SoC design, especially in regard to the high cost of errors in its output. The constant evolution imposed on the compiler impedes on its quality. Model transformations permit to improve the quality for two reasons. First, this is due to the increased reuse and maintainability provided at the fine grain level of the rules and at the coarse grain level of the transformations. Second, the fact that each transformation has its output conform to metamodels allows to *formally* verify the structure of the models, at each stage of the compilation chain.

## 5 Conclusions

In this article we have first presented the various needs and constraints existing in the context of SoC co-design. The usage of Model-Driven Engineering in this context have been detailed using an example, our project Gaspard. First, the SoC designer creates a model of the SoC with all the information needed for the implementation, that is: without ambiguity and with all the details concerning the realization of even the most elementary components. Second, the model is used as the input of a chain of transformations taking the role of a compiler. Depending on the target selected by the user, a specific sequence of transformations is executed until the generation of code is reached. Each transformation has its input and output conforming to specific metamodels and is actually a set of rules. In addition, each rule is written in a declarative way, having an input pattern and an output pattern.

Then, we have emphasized the benefits of model transformations in the particular context of SoC compilation. The transparency brought by the transformations to the compiler contrasts with the usual monolithic tools and helps the user to better understand the concepts he has to manipulate. The strong separation between each transformation facilitates the simultaneous development of the compiler by different persons. Model transformations also ease the reuse between the numerous compilation chains present in the SoC design. Moreover, the separation of each transformation in rules permits not only to reuse them easily in different chains but also smooth the necessary constant evolution of the SoC compiler. Finally, another advantage is that the reliability of the compilation is improved thanks to better maintainability and the formal representation of each stage of the compilation as a metamodel.

The Gaspard environment already produces simulations capable of providing performance and consumption estimations. One of the topics we are now investigating is the automatization of the design space exploration. In order to remove a bottleneck pointed out by the generated simulation, the environment should be able to automatically find out which part of the original model has to be modified, and in which way. The traceability of a whole chain of transformations is an important step to achieve this goal. In future works we will explore additional features of the usage of MDE. In particular we will consider transformation composition as a way to facilitate even more the reuse within the compilation chains. Another perspective is the description of the transformations as models, which would allow a graphical representation of the transformation rules.



## References

1. ITRS, International Technology Roadmap for Semiconductors: Design, 2005 edition (2005), <http://www.itrs.net/>
2. Donlin, A.: Transaction level modeling: flows and use models. In: Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), Stockholm, Sweden, pp. 75–80 (2004)
3. Honda, S., Wakabayashi, T., Tomiyama, H., Takada, H.: RTOS-centric hardware/-software cosimulator for embedded system design. In: Conference on Hardware/-Software Codesign and System Synthesis (CODES+ISSS 2004), Stockholm, Sweden (September 2004)
4. Hamerly, G., Perelman, E., Lau, J., Calder, B.: Simpoint 3.0: Faster and more flexible program analysis. In: Workshop on Modeling, Benchmarking and Simulation, Madison, Wisconsin, USA (June 2005)
5. Kahn, G.: The semantics of a simple language for parallel programming. In: Rosenfeld, J.L. (ed.) Information Processing 1974: Proceedings of the IFIP Congress 1974, pp. 471–475. North-Holland, Amsterdam (1974)
6. Lee, E.A., Messerschmitt, D.G.: Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. on Computers* (January 1987)
7. Grotker, T., Liao, S.: *Al: System Design with SystemC*. Kluwer Publishers, Dordrecht (2002)
8. Gerstlauer, D., Peng, G.: *System Design: A Practical Guide with SpecC*. Kluwer Academic Publishers, Dordrecht (2001)
9. Ismail, T.B., Abid, M., Jerraya, A.: COSMOS: a codesign approach for communicating systems. In: CODES 1994: Proceedings of the 3rd international workshop on Hardware/software co-design, Los Alamitos, CA, USA, pp. 17–24. IEEE Computer Society Press, Los Alamitos (1994)
10. Chou, P., Ortega, R., Borriello, G.: The chinook hardware/software co-synthesis system. Technical Report TR-95-03-04 (1995)
11. Gajski, D., Vahid, F., Narayan, S., Gong, J.: Specsyn: An environment supporting the specify-explorerefine paradigm for hardware/software system design. *IEEE Transactions on Very Large Scale Integration Systems* 6(1), 84–100 (1998)
12. Chang, H., Cooke, L., Hunt, M., Martin, G., McNelly, A.J., Todd, L.: *Surviving the SOC revolution: a guide to platform-based design*. Kluwer Academic Publishers, Norwell (1999)
13. Schirrmeister, F., Sangiovanni-Vincentelli, A.: Virtual component co-design – applying function architecture co-design to automotive applications. In: Proceedings of the IEEE International Vehicle Electronics Conference, Tottori, Japan (September 2001)
14. CoWare inc.: CoWare N2C (2001), <http://www.coware.com/cowareN2C.html>
15. Cesário, O.W., Lyonnard, D., Nicolescu, G., Paviot, Y., Yoo, S., Jerraya, A. A., Gauthier, L., Diaz-Nava, M.: Multiprocessor SoC platforms: A component-based design approach. *IEEE Des. Test* 19(6), 52–63 (2002)
16. Jerraya, A.A., Yoo, S., Bouchhima, A., Nicolescu, G.: Validation in a component-based design flow for multicore SoCs. In: ISSS, pp. 162–167. IEEE Computer Society, Los Alamitos (2002)
17. Planet MDE: Model Driven Engineering (2007), <http://planetmde.org>
18. Object Management Group, Inc., ed.: Final Adopted OMG SysML Specification (May 2006), <http://www.omg.org/cgi-bin/doc?ptc/06-0504>

19. Object Management Group, Inc., ed.: UML Extension Profile for SoC RFC (March 2005), <http://www.omg.org/cgi-bin/doc?realtime/2005-03-01>
20. Nguyen, K.D., Sun, Z., Thiagarajan, P.S., Wong, W.F.: Model-driven SoC design via executable UML to SystemC. In: RTSS 2004: Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS 2004), Washington, pp. 459–468. IEEE Computer Society, Los Alamitos (2004)
21. Alanen, M., Lilius, J., Porres, I., Truscan, D., Oliver, I., Sandstrom, K.: Design method support for domain specific soc design. In: MBD-MOMPES 2006: Proceedings of the Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MBD-MOMPES 2006), pp. 25–32. IEEE Computer Society, Washington (2006)
22. Riccobene, E., Scandurra, P., Rosti, A., Bocchio, S.: A model-driven design environment for embedded systems. In: DAC 2006: Proceedings of the 43rd annual conference on Design automation, pp. 915–918. ACM, New York (2006)
23. Szemethy, T., Karsai, G., Balasubramanian, D.: Model transformations in the Model-Based Development of real-time systems. In: ECBS 2006: Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems, Washington, pp. 177–186. IEEE Computer Society, Los Alamitos (2006)
24. WEST Team LIFL, Lille, France: Graphical array specification for parallel and distributed computing (GASPARD-2) (2005), <http://www.lifl.fr/west/gaspard/>
25. ProMarte partners: UML Profile for MARTE, Beta 1 (August 2007), <http://www.omg.org/cgi-bin/doc?ptc/2007-08-04>
26. Boulet, P., Marquet, P., Piel, E., Taillard, J.: Repetitive Allocation Modeling with MARTE. In: Forum on specification and design languages (FDL 2007), Barcelona, Spain, (September 2007) (Invited Paper)
27. Boulet, P.: Array-OL revisited, multidimensional intensive signal processing specification. Research Report RR-6113, INRIA (February 2007)
28. Eclipse Consortium: EMF (2007), <http://www.eclipse.org/emf>
29. Etien, A., Dumoulin, C., Renaux, E.: Towards a unified notation to represent model transformation. Research Report RR-6187, INRIA (May 2007)
30. Object Management Group, Inc.: MOF Query / Views / Transformations, OMG paper (November 2005), <http://www.omg.org/docs/ptc/05-11-01.pdf>
31. Eclipse Consortium: JET, Java Emitter Templates (2007), <http://www.eclipse.org/modeling/m2t/?project=jet>
32. Atitallah, R.B., Piel, E., Niar, S., Marquet, P., Dekeyser, J.L.: Multilevel MPSoC simulation using an MDE approach. In: IEEE International SoC Conference (SoCC 2007), Hsinchu, Taiwan (September 2007)
33. Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: PACT 13 IEEE International Conference on Parallel Architecture and Compilation Techniques, Juan-les-Pins, France, pp. 7–16 (September 2004)
34. Bunse, C., Gross, H.G., Peper, C.: Applying a model-based approach for embedded system development. In: EUROMICRO 2007: Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO 2007), Washington, pp. 121–128. IEEE Computer Society, Los Alamitos (2007)

# Implementation of a Finite State Machine with Active Libraries in C++<sup>\*</sup>

Zoltán Juhász, Ádám Sipos, and Zoltán Porkoláb

Department of Programming Languages and Compilers  
Faculty of Informatics  
Eötvös Loránd University  
H-1117 Budapest, Pázmány Péter sétány 1/C  
{cad,shp,gsd}@inf.elte.hu

**Abstract.** Active libraries are code parts playing an active role during compilation. In C++ active libraries are implemented with the help of template metaprogramming (TMP) techniques. In this paper we present an active library designed as an implementation tool for Finite state machines. With the help of various TMP constructs, our active library carries out compile-time actions like optimizations via state-minimalization, and more sophisticated error-detection steps. Our library provides extended functionality to the Boost::Statechart library, the popular FSM implementation of the Boost library. We describe the implementation and analyze the efficiency.

## 1 Introduction

*Generative programming* is one of today's popular programming paradigms. This paradigm is primarily used for generating customized programming components or systems. *C++ template metaprogramming* (TMP) is a generative programming style. TMP is based on the C++ *templates*. Templates are key language elements for the C++ programming language [25], and are essential for capturing commonalities of abstractions. A cleverly designed C++ code with templates is able to utilize the type-system of the language and force the compiler to execute a desired algorithm [31]. In template metaprogramming the program itself is running during compilation time. The output of this process is still checked by the compiler and run as an ordinary program.

Template metaprogramming is proved to be a Turing-complete sublanguage of C++ [6]. We write metaprograms for various reasons, here we list some of them:

- *Expression templates* [32] replace runtime computations with compile-time activities to enhance runtime performance.
- *Static interface checking* increases the ability of the compile-time to check the requirements against template parameters, i.e. they form constraints on template parameters [18,23].

---

<sup>\*</sup> Supported by GVOP-3.2.2.-2004-07-0005/3.0.

- *Language embedding* makes it possible to introduce domain-specific code into a C++ program via a metaprogramming framework. Examples include SQL embedding [11], and a type-safe XML framework [13].
- *Active libraries* [29]. act dynamically during compile-time, making decisions based on programming contexts and making optimizations. These libraries are not passive collections of routines or objects, as are traditional libraries, but take an active role in generating code. Active libraries provide higher abstractions and can optimize those abstractions themselves.

*Finite State Machines* (FSMs) are well-known mathematical constructs, their practical applications include but are not limited to lexical analyzers, performance tests, and *protocol definition implementations*. Most protocols are described by a FSM, since FSMs provide a clear framework for distinguishing the possible state transitions when the protocol is in a certain state. However, since often only the results of test cases of a protocol are obtainable, the developer himself has to define and implement his own state machine description.

FSMs play a central role in many modern software systems. Besides their functionality, their correctness and effectiveness is also crucial. Unfortunately, recent implementation techniques provide no support for features like detecting unreachable states and carrying out automatic state reductions. This lack of features may reduce the quality and the effectiveness of FSM code used in critical applications.

With the help of active libraries we are able to define state machines, do sanity checking on their state tables, and enhance their run-time effectiveness at compile-time in a fully automated manner. Such process can either detect consistency errors during the compilation process, or produce a correct and optimized FSM for run-time usage.

Our goal is to demonstrate the possibility to implement and effectively use active libraries matching the above criteria. Our library is capable of carrying out compile-time operations and performs various checkings and optimizations on a state machine.

The paper is organized as follows. In section 2 we discuss C++'s templates and template metaprogramming concepts. Section 3 introduces the Finite State Machine's formal definition. Section 4 describes common implementation techniques for finite state machines. We discuss the possible implementation techniques in section 5. The code efficiency and compilation time measurement results are presented in section 6. Future development directions and related work are discussed in section 7.

## 2 C++ Template Metaprograms

### 2.1 Compile-Time Programming

*Templates* are an important part of the C++ language, by enabling data structures and algorithms to be parameterized by types. This abstraction is frequently needed when using general algorithms like finding an element in a data structure, or data types like a *list* of elements. The mechanism behind a list containing

integer numbers, or strings is essentially the same, it is only the *type* of the contained objects that differs. With templates we can express this abstraction, thus this *generic* language construct aids code reuse, and the introduction of higher abstraction levels. Let us consider the following code:

```
template <class T>                int main()
class list                        {
{                                  ...
public:                            list<int> li; // instantiation
    list();                        li.insert(1928);
    void insert(const T& x);    }
    T first();
    void sort();
    //...
};
```

This *list* template has one type parameter, called *T*, referring to the future type whose objects the list will contain. In order to use a list with some specific type, an *instantiation* is needed. This process can be invoked either implicitly by the compiler when the new list is first referred, or explicitly by the programmer. During instantiation the template parameters are substituted with the concrete arguments. This newly generated code part is compiled, and inserted into the program.

The template mechanism of C++ is unique, as it enables the definition of *partial* and *full specializations*. Let us suppose that for some type (in our example `bool`) we would like to create a more efficient type-specific implementation of the *list* template. We may define the following specialization:

```
template<>
class list<bool>
{
    // a completely different implementation may appear here
};
```

The specialization and the original template only share the *name*. A specialization does not need to provide the same functionality, interface, or implementation as the original.

## 2.2 Metaprograms

In case the compiler deduces that in a certain expression a concrete instance of a template is needed, an implicit instantiation is carried out. Let us consider the following code demonstrating programs computing the factorial of some integer number by invoking a recursion:

```
// compile-time recursion          // runtime recursion
template <int N>                  int Factorial(int N)
```

```

struct Factorial          {
{
    enum { value = N *      if (N==1) return 1;
      Factorial <N-1>::value }; }
};
return N*Factorial(N-1);

template<>
struct Factorial<1>
{
    enum { value = 1 };
};

int main()                int main()
{
    int r=Factorial<5>::value;    {
    int r=Factorial(5);          }
}

```

As the expression `Factorial<5>::value` must be evaluated in order to initialize `r` with a value, the `Factorial` template is instantiated with the argument 5. Thus in the template the parameter `N` is substituted with 5 resulting in the expression `5 * Factorial<4>::value`. Note that `Factorial<5>`'s instantiation cannot be finished until `Factorial<4>` is instantiated, etc. This chain is called an *instantiation chain*. When `Factorial<1>::value` is accessed, instead of the original template, the full specialization is chosen by the compiler so the chain is stopped, and the instantiation of all types can be finished. This is a *template metaprogram*, a program run in compile-time, calculating the factorial of 5.

In our context the notion *template metaprogram* stands for the collection of templates, their instantiations, and specializations, whose purpose is to carry out operations in compile-time. Their expected behavior might be either emitting messages or generating special constructs for the runtime execution. Henceforth we will call a *runtime program* any kind of runnable code, including those which are the results of template metaprograms.

C++ template metaprogram actions are defined in the form of template definitions and are “executed” when the compiler instantiates them. Templates can refer to other templates, therefore their instantiation can instruct the compiler to execute other instantiations. This way we get an instantiation chain very similar to a call stack of a runtime program. Recursive instantiations are not only possible but regular in template metaprograms to model loops.

Conditional statements (and stopping recursion) are solved via specializations. Templates can be overloaded and the compiler has to choose the narrowest applicable template to instantiate. Subprograms in ordinary C++ programs can be used as data via function pointers or functor classes. Metaprograms are first class citizens in template metaprograms, as they can be passed as parameters to other metaprograms [6].

Data is expressed in runtime programs as variables, constant values, or literals. In metaprograms we use `static const` and enumeration values to store

quantitative information. Results of computations during the execution of a metaprogram are stored either in new constants or enumerations. Furthermore, the execution of a metaprogram may trigger the creation of new types by the compiler. These types may hold information that influences the further execution of the metaprogram.

Complex data structures are also available for metaprograms. Recursive templates are able to store information in various forms, most frequently as tree structures, or sequences. Tree structures are the favorite implementation forms of expression templates [32]. The canonical examples for sequential data structures are `typelist` [2] and the elements of the `boost::mpl` library [16].

### 2.3 Active Libraries

With the development of programming languages, user libraries also became more complex. *FORTTRAN* programs already relied heavily on programming libraries implementing solutions for re-occurring tasks. With the emerging of object-oriented programming languages the libraries also transformed: the sets of functions were replaced by classes and inheritance hierarchies. However, these libraries are still *passive*: the writer of the library has to make substantial decisions about the types and algorithms at the time of the library's creation. In some cases this constraint is a disadvantage. Contrarily, an *active library* [29] acts dynamically, makes decisions in compile-time based on the calling context, chooses algorithms, and optimizes code. In C++ active libraries are often implemented with the help of template metaprogramming. Our compile-time FSM active library also utilizes TMP techniques.

## 3 Finite State Machine

The *Finite State Machine (FSM)* is a model of behavior composed of a finite number of states, transitions between those states, and optionally actions. The transitions between the states are managed by the transition function depending on then input symbol (event). In the rest of this paper we use the expression Finite State Machine (FSM), automaton or machine in terms of Deterministic Finite State Machine (DFSM). Deterministic Finite State Machines, Deterministic Finite Tree Automatons etc. are a widespread model for implementing a communication protocol, a program drive control flow or lexical analyzer among others. The solution of a complex problem with a FSM means the decomposition of the problem into smaller parts (states) whose tasks are precisely defined.

### 3.1 A Mathematical Model of Finite State Machine

A *transducer* Finite State Machine is a six tuple [20], consisting of

- Let  $\Sigma$  denote a finite, non empty set of *input symbols*. We are referring to this set as the set of events
- Let  $\Gamma$  denote a finite, non empty set of *output symbols*

- Let  $S$  denote a finite set of *States*
- Let  $q_0 \in Q$  denote the *Start or Initial state*, an element of  $S$
- A *Transition function*:  $\delta : Q \times \Sigma \rightarrow Q$
- Let  $\omega$  denote an *Output function*

The working mechanism of a FSM is as follows. First the FSM is in the Start state. Each input symbol (event) makes the FSM move into some state depending on the transition function, and the current state. If the event-state pair is not defined by the function, the event in that state is not legal. For practical purposes we introduce a 7th component to the FSM, which is a set of actions. These actions that are executed through a transition between two states. Note that our model uses the Moore machine [20].

## 4 Common Implementation Techniques

There are a number of different FSM implementation styles from hand-crafted to professional hybrid solutions. In the next section we review some common implementation techniques.

### 4.1 Procedural Solution

This is the simplest, but the least flexible solution of the implementation of a DFSM. The transition function's rules are enforced via control structures, like switch-case statements. States and events are regularly represented by enumerations, actions are plain function calls.

The biggest drawback of this implementation is that it is suitable only for the representation of simple machines, since no framework for sanity checking is provided, therefore the larger the automaton, the more error prone and hard to read its code [22]. Such implementations are rare in modern industrial programs, but often used for educational or demonstrational purposes.

### 4.2 Object-Oriented Solution with a State Transition Table

The object-oriented representation is a very widespread implementation model. The transition function behavior is modeled by the state transition table (STT). Table 1 shows a sample STT:

**Table 1.** State Transition Table

Current State	Event	Next State	Action
Stopped	play	Playing	start_playback
Playing	stop	Stopped	stop_playback

**Current State** contains a state of the automaton, **Event** is an event that can occur in that state, **Next State** is the following state of the state machine after the transition, and **Action** is a function pointer or a function object that is going to be executed during the state transition.



A good example of such an automaton implementation is the OpenDiameter communication protocol library's FSM [10]. One of the main advantages of the Object-Oriented solution over the hand-crafted version is that the state transition rules and the code of execution are separated and it supports the incrementality development paradigm in software engineering. The drawback of an average OO FSM implementation is that the state transition table is defined and built at runtime. This is definitely not free of charge. Sanity checking also results in runtime overhead and incapable of preventing run-time errors.

### 4.3 Hybrid Technique

The most promising solution is using the Object-Oriented and template-based generative programming techniques side by side. States, events and even actions are represented by classes and function objects, and the STT is defined at compilation time with the heavy use of C++ template techniques, like Curiously Recurring Template Pattern (CRTP) [8].

An outstanding example of such DFSM implementation is Boost::Statechart Library [9], which is UML compatible, supports multi-threading, type safe and can do some basic compile time consistency checking. However, Boost::Statechart is not based on template metaprograms, therefore it does not contain more complex operations, like FSM minimization.

## 5 Our Solution

As soon as the STT is defined at compilation time, algorithms and transformations can be executed on it, and also optimizations and sanity checking of the whole state transition table can be done. Therefore we decided to step forward towards using template metaprograms to provide automatic operations at compile-time on the FSM. Our goal was to develop an initial study that:

- carries out compound examinations and transformation on the state transition table,
- and shows the relationship between Finite State Machines and Active Libraries over a template metaprogram implementation of the Moore reduction procedure.

The library is based on a simplified version of *Boost::Statechart's* State Transition Table. In the future we would like to evolve this code base to a library that can cooperate with *Boost::Statechart* library and function as an extension.

### 5.1 Applied Data Structures and Algorithms

We used many C++ template facilities extensively, such as SFINAE, template specialization, parameter deduction etc [21]. In a metaprogram you use compile time constants and types instead of variables and objects respectively; class templates and function templates instead of functions and methods. To simulate cycles and if-else statements we used recursive template instantiations and partial and full template specializations.

```

template< typename T, typename From, typename Event, typename To,
         bool (T::* transition_func)(Event const&)>
struct transition
{
    typedef T          fsm_t;
    typedef From      from_state_t;
    typedef Event     event_t;
    typedef To        to_state_t;

    typedef typename Event::base_t base_event_t;
    static bool do_transition(T& x, base_event_t const& e)
    {
        return (x.*transition_func)(static_cast<event_t const &>(e));
    }
};

typedef mpl::list<
//   Current state   Event   Next state       Action
//   +-----+-----+-----+-----+
trans < Stopped   ,   play   ,   Playing   ,   &p::start_playback >,
trans < Playing   ,   stop   ,   Stopped   ,   &p::stop_playback  >
//   +-----+-----+-----+-----+
>::type sample_transition_table; // end of transition table

```

Fig. 1. Implementation of our State Transition Table

Assignment is also unknown in the world of metaprograms, we use typedef specifiers to introduce new type aliases, that hold the required result.

We used *Boost::MPL* [5], which provides C++ STL-style [19] compile-time containers and algorithms.

In our model the State Transition Table defines a directed graph. We implemented the Moore reduction procedure, used the Breadth-First Search (BFS) algorithm to isolate the graph's main strongly connected component and with the help of a special "Error" state we made it complete.

Much like the *Boost::Statechart*'s STT, in our implementation states and events are represented by classes, structs or any built-in types. The STT's implementation based on the *Boost::MPL::List* compile-time container is described in Figure 1.

A transition table built at compile-time behaves similarly to a counterpart built in runtime. The field `transition_func` pointer to member function represents the tasks to be carried out when a state transition happens. The member function `do_transition()` is responsible for the iteration over the table. The state appearing in the first row is considered the starting state.

## 5.2 A Case Study

In this section we present a simple use case. Let us imagine that we want to implement a simple CD player, and the behavior is implemented by a state machine. The state transition table skeleton can be seen in Figure 2.

```

typedef mpl::list<
//   Current state   Event   Next state           Action
//   +-----+-----+-----+-----+
trans < Stopped    ,   play   ,   Playing   ,   &p::start_playback >,
trans < Playing    ,   stop   ,   Stopped    ,   &p::stop_playback  >,
trans < Playing    ,   pause  ,   Paused     ,   &p::pause           >,
trans < Paused     ,   resume ,   Playing    ,   &p::resume          >,
... duplicated functionality ...
trans < Stopped    ,   play   ,   Running    ,   &p::start_running  >,
trans < Running    ,   stop   ,   Stopped    ,   &p::stop_running   >,
trans < Running    ,   pause  ,   Paused     ,   &p::pause           >,
trans < Paused     ,   resume ,   Playing    ,   &p::resume          >,
... unreachable states ...
trans < Recording  ,   pause  ,   Pause_rec  ,   &p::pause_recording>,
trans < Paused_rec,   resume ,   Recording  ,   &p::resume_rec     >
//   +-----+-----+-----+-----+
>::type sample_trans_table; // end of transition table

```

**Fig. 2.** Sample State Transition Table

The programmer first starts to implement the Stopped, Playing, and Paused states' related transitions. After implementing a huge amount of other transitions, eventually he forgets that a Playing state has already been added, so he adds it again under the name *Running*. This is an unnecessary redundancy, and in general could indicate an error or sign of a bad design. A few weeks later it turns out, that a recording functionality needs to be added, so the programmer adds the related transitions. Unfortunately, the programmer forgot to add a few transitions, so the Recording and Paused state cannot be reached. In general that also could indicate an error. On the other hand if the state transition table contains many unreachable states, these appear in the program's memory footprint and can cause runtime overhead.

Our library can address these cases by emitting warnings, errors messages, or by eliminating unwanted redundancy and unreachable states. The result table of the reduction algorithm can be seen here:

```
template struct fsm_algs::reduction< sample_trans_table >;
```

After this forced template instantiation, the `enhanced_table` typedef within this `struct` holds an optimized transition table is described in Figure 3.

### 5.3 Implementation of the Algorithms

In the following we present the minimization algorithm implemented in our active library.

**Locating Strongly Connected Components.** The first algorithm executed before the Moore reduction procedure is the localization of the strongly connected

```

typedef mpl::list<
//   Current state   Event   Next state           Action
//   +-----+-----+-----+-----+
trans < Stopped   ,   play   ,   Playing   ,   &p::start_playback >,
trans < Playing   ,   stop   ,   Stopped   ,   &p::stop_playback  >,
trans < Playing   ,   pause  ,   Paused    ,   &p::pause           >,
trans < Paused    ,   resume ,   Playing   ,   &p::resume          >,
    ... duplicated functionality has been removed ...
    ... unreachable states have been removed too ...
//   +-----+-----+-----+-----+
>::type sample_trans_table; // end of transition table

```

**Fig. 3.** Reduced Transition Table

component of the STT’s graph from a given vertex. We use Breadth-First Search to determine the strongly connected components. After we have located the main strongly connected component from a given state, we can emit a warning / error message if there is more than one component (unreachable states exist) or we can simply delete them. The latter technique can be seen in Figure 4 (several lines of code have been removed):

**Making the STT’s Graph Complete.** The Moore reduction algorithm requires a complete STT graph, so the second algorithm that will be executed before the Moore reduction procedure is making the graph complete. We introduce a special “Error” state, which will be the destination for every undefined state-event pair. We test every state and event and if we find an undefined event for a state, we add a new row to the State Transition Table. (Figure 5)

The destination state is the “Error” state. We can also define an error-handler function object [19]. After this step, if the graph was not complete, we’ve introduced a lot of extra transitions. If they are not needed by the user of the state machine, these can be removed after the reduction. The result after the previously executed two steps is a strongly connected, complete graph. Now we are able to introduce the Moore reduction procedure.

**The Moore Reduction Procedure.** Most of the algorithms and methods used by the reduction procedure have already been implemented in the previous two steps.

First we suppose that all states may be equivalent i.e. may be combined into every other state. Next we group non-equivalent states into different groups called equivalence partitions. When no equivalence partitions have states with different properties, states in the same group can be combined. We refer to equivalent partitions as sets of states having the same properties. [14]

We have simulated partitions and groups with Boost::MPL’s compile time type lists. Every partition’s groups are represented by lists in lists. The outer list represents the current partition, the inner lists represent the groups. Within two steps we mark group elements that need to be reallocated. These elements will be reallocated before the next step into a new group (currently list).

```

// Breadth-First Search
template < typename Tlist, typename Tstate, typename Treached,
//          STT ^ Start state ^ Reached states ^
          typename Tresult = typename mpl::clear<Tlist>::type,
//          ^ Result list is initialized with empty list
          bool is_empty = mpl::empty<Treated>::value >
struct bfs
{
    // Processing the first element of the reached list
    typedef typename mpl::front<Treated>::type process_trans;
    typedef typename process_transition::to_state_t next_state;

    // (...) Removing first element
    typedef typename mpl::pop_front<Treated>::type
        tmp_reached_list;

    // (...) Adding recently processed state table rows
    // to the already processed (reachead) list
    typedef typename merge2lists<tmp_result_list, tmp_reached_list>
        ::result_list tmp_check_list;

    // (...) Recursively instantiates the bfs class template
    typedef typename bfs< Tlist, next_state, reached_list,
        tmp_result_list, mpl::empty<reached_list>::value>
        ::result_list result_list;
};

```

**Fig. 4.** Implementation of Breadth-First Search

After the previous three steps the result is a reduced, complete FSM whose STT has only one strongly connected component. All of these algorithms are executed at compile time, so after the compilation we are working with a minimized state machine.

## 6 Results

The aim of the previously introduced techniques is to prove that we are able to do sanity checks and transformations on an arbitrary FSM State Transition Table. With the help of these utilities we can increase our automaton's efficiency and reliability without any runtime cost. We can also help the developer since compile-time warnings and errors can be emitted to indicate STT's inconsistency or unwanted redundancy. Our algorithms are platform independent because we are only using standard facilities defined in the C++ 2003 language standard (ISO/IEC 14882) [21], and elements of the highly portable Boost library. Supported and tested platforms are the following:

```
// Current state  Event      Next state      Action
// +-----+-----+-----+-----+
trans <   Stop  ,  pause  ,   Error   ,  &p::handle_error  >
```

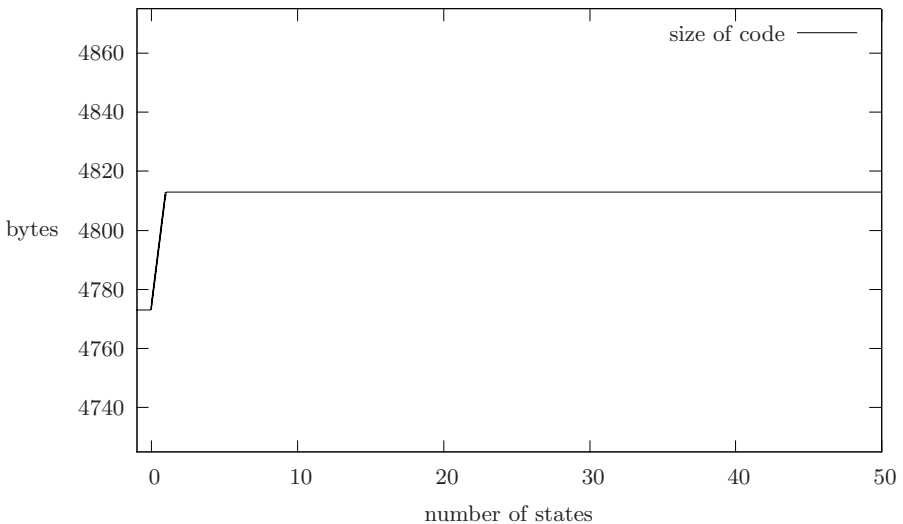
**Fig. 5.** Adding new transition

- Comeau C/C++ 4.2.45, 4.3.3
- Compaq C++ (Tru64 UNIX) 6.5
- GCC 3.2.2, 3.3.1, 3.4, 4.1.0
- Intel C++ 7.1, 8.0, 9.1
- Metrowerks CodeWarrior 4.2.45, 4.3.3
- Microsoft Visual C++ 7.1

In the following we present code size and compilation time measurements with the gcc 4.1.0 20060304 (Red Hat 4.1.0-3) compiler. The test consists of the definition and consistency checking of a state transition table.

## 6.1 Code Size

The  $x$  axis represents the number of states, while  $y$  shows the resulting code size in bytes. At 0 states the program consists of our library, while no state transition table is used. Binary code size is increased only when the first state is introduced. The graph shows no further code size increase when the FSM consists of more states. (Figure 6) The reason is that the representation of each state is a type, which is compile-time data. This data is not inserted into the final code.



**Fig. 6.** Number of states and size of code

## 6.2 Compilation Time

The testing method is essentially the same as above. Compilation time does not grow linearly with the introduction of new states (Figure 7).

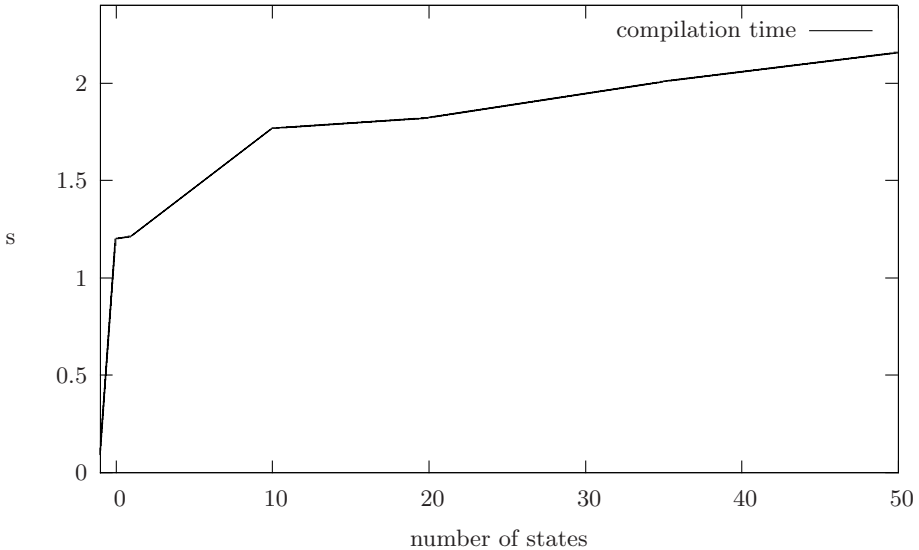


Fig. 7. Number of states and compilation time

## 7 Related Work and Future Work

Final State Machine implementations vary from fully procedural [22] to object-oriented solutions [10]. Flexibility and maintainability are becoming better, but the correctness of the created automaton ultimately depended on the programmers caution. Template techniques were introduced to enhance run-time performance [8], but not for providing sanity checks on the FSM.

The Boost Statechart Library supports a straightforward transformation of UML statecharts to executable C++ code [9]. The library is type safe, supports thread-safety and performs some basic compile time consistency checking. However, Boost::Statechart is not based on template metaprograms, therefore it does not contain more complex operations, like FSM minimization.

In the future we intend to extend the library with the following functionalities.

- *Warnings, error messages* - The library minimizes the graph without asking for confirmation from the programmer. Warnings and errors could be emitted by the compiler whenever an isolated node or reducible states are found.
- *Joker states, events* - In some cases it would be convenient to have a state in the FSM that acts the same regardless of the input event. Now we have to define all such state transitions in the STT. With future “joker” states and events, the STT definition would be simpler for the user, and also the

reduction algorithm would have a smaller graph to work on. On the other hand the representation and the library logic would get more complex.

- *Composite states* - A composite state is a state containing a FSM. In case such state is reached by the outer machine, this inner automaton is activated. This FSM might block the outer automaton.

## 8 Conclusion

We created an active library to implement Final State Machines functionally equivalent to the Boost::Statechart library [9]. Our library is *active* in the sense that it carries out various algorithms at compile time. Algorithms include state machine reduction, and extended error checking.

The library carries out checking and transformations on a FSM's State Transition Table. The active library contains an implementation of the Moore reduction procedure and other algorithms. The algorithms are executed during compilation in the form of template metaprograms, therefore no runtime penalties occur. If the reduction is possible, the FSM is expected to be faster during its execution in runtime. The usage of such compile time algorithms has little impact on the code size.

On the other hand, with the aid of compile time checking and the emitted warnings and error messages the program code will be more reliable, since the program can only be compiled if it meets the developer's requirements. These requirements can be assembled through compile time checking.

Our implementation is based on the Boost::MPL and Boost::Statechart Libraries. As the library uses only standard C++ features, it is highly portable and successfully tested in different platforms.

## References

1. Abrahams, D., Gurtovoy, A.: C++ template metaprogramming, Concepts, Tools, and Techniques from Boost and Beyond. Addison-Wesley, Boston (2004)
2. Alexandrescu, A.: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley, Reading (2001)
3. ANSI/ISO C++ Committee. Programming Languages - C++. ISO/IEC 14882:1998(E). American National Standards Institute (1998)
4. Boost Concept Checking library,  
[http://www.boost.org/libs/concept\\_check/concept\\_check.html](http://www.boost.org/libs/concept_check/concept_check.html)
5. Boost Metaprogramming library,  
<http://www.boost.org/libs/mpl/doc/index.html>
6. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools and Applications. Addison-Wesley, Reading (2000)
7. Czarnecki, K., Eisenecker, U.W., Glck, R., Vandevoorde, D., Veldhuizen, T.L.: Generative Programming and Active Libraries. Springer, Heidelberg (2000)
8. James, O.: Coplien: Curiously Recurring Template Patterns. C++ Report (February 1995)
9. Dnni, A.H.: Boost:Statechart,  
<http://boost-sandbox.sourceforge.net/libs/statechart/doc/index.html>



10. Fajardo, V., Ohba, Y.: Open Diameter, <http://www.opendiameter.org/>
11. Gil, Y., Lenz, K.: Simple and Safe SQL Queries with C++ Templates. In: Proceedings of the 6th international conference on Generative programming and component engineering, pp. 13–24, Salzburg, Austria (2007)
12. Gregor, D., Jrvi, J., Siek, J.G., Reis, G.D., Stroustrup, B., Lumsdaine, A.: Concepts: Linguistic Support for Generic Programming in C++. In: Proceedings of the 2006 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA 2006) (October 2006)
13. Solodkyy, Y., Järvi, J., Mlaih, E.: Extending Type Systems in a Library — Type-safe XML processing in C++. In: Workshop of Library-Centric Software Design at OOPSLA 2006, Portland Oregon (2006)
14. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading (2000)
15. Juhász, Z.: Implementing Finite State Automata with Active Libraries M.Sc. Thesis. Budapest (2006)
16. Karlsson, B.: Beyond the C++ Standard Library, An Introduction to Boost. Addison-Wesley, Reading (2005)
17. Knuth, D.E.: An Empirical Study of FORTRAN Programs. *Software - Practice and Experience* 1, 105–133 (1971)
18. McNamara, B., Smaragdakis, Y.: Static interfaces in C++. In: First Workshop on C++ Template Metaprogramming (October 2000)
19. Musser, D.R., Stepanov, A.A.: Algorithm-oriented Generic Libraries. *Software-practice and experience* 27(7), 623–642 (1994)
20. Hopcroft, J.E., Motwani, R., Ullman, J.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading (1969)
21. Programming languages C++, ISO/IEC 14882 (2003)
22. Samek, M.: Practical Statecharts in C/C++. CMP Books (2002)
23. Siek, J., Lumsdaine, A.: Concept checking: Binding parametric polymorphism in C++. In: First Workshop on C++ Template Metaprogramming (October 2000)
24. Siek, J.: A Language for Generic Programming. PhD thesis, Indiana University (August 2005)
25. Stroustrup, B.: The C++ Programming Language Special Edition. Addison-Wesley, Reading (2000)
26. Stroustrup, B.: The Design and Evolution of C++. Addison-Wesley, Reading (1994)
27. Unruh, E.: Prime number computation. ANSI X3J16-94-0075/ISO WG21-462
28. Vandevoorde, D., Josuttis, N.M.: C++ Templates: The Complete Guide. Addison-Wesley (2003)
29. Veldhuizen, T.L., Gannon, D.: Active libraries: Rethinking the roles of compilers and libraries. In: Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO 1998), pp. 21–23. SIAM Press, Philadelphia (1998)
30. Veldhuizen, T.: Five compilation models for C++ templates. In: First Workshop on C++ Template Metaprogramming (October 2000)
31. Veldhuizen, T.: Using C++ Template Metaprograms. *C++ Report* 7(4), 36–43 (1995)
32. Veldhuizen, T.: Expression Templates. *C++ Report* 7(5), 26–31 (1995)
33. Zólyomi, I., Porkoláb, Z.: Towards a template introspection library. In: Karsai, G., Visser, E. (eds.) GPCE 2004. LNCS, vol. 3286, pp. 266–282. Springer, Heidelberg (2004)

# Automated Merging of Feature Models Using Graph Transformations\*

Sergio Segura, David Benavides, Antonio Ruiz-Cortés, and Pablo Trinidad

University of Seville, Spain

{sergiosegura, benavides, aruiz, ptrinidad}@us.es

**Abstract.** Feature Models (FMs) are a key artifact for variability and commonality management in Software Product Lines (SPLs). In this context, the merging of FMs is being recognized as an important operation to support the adoption and evolution of SPLs. However, providing automated support for merging FMs still remains an open challenge. In this paper, we propose using graph transformations as a suitable technology and associated formalism to automate the merging of FMs. In particular, we first present a catalogue of technology-independent visual rules to describe how to merge FMs. Next, we propose a prototype implementation of our catalogue using the AGG system. Finally, we show the feasibility of our proposal by means of a running example inspired by the mobile phone industry. To the best of our knowledge, this is the first approach providing automated support for merging FMs including feature attributes and cross-tree constraints.

## 1 Introduction

*Software Product Line* (SPL) engineering is an approach to developing families of software systems in a systematic way [10]. Roughly speaking, an SPL can be defined as a set of software products sharing a common set of features. A feature is defined as an increment in product functionality [5]. In this context, *Feature Models* (FMs) are commonly used to provide a compact and visual representation of all the products of an SPL in terms of features.

Typical SPL adoption strategies involve building an SPL from a set of existing software products or extending an existing SPL to include new products [17]. In both cases, the artifacts of different software systems must be combined into a single SPL. In this context, the usage of specific SPL refactoring techniques is emerging as a key practice to support the adoption and evolution of SPLs [11, 18, 30].

It is accepted that not only programs should be refactored in the context of an SPL but also FMs. In particular, the merging of FMs emerges as an appealing operation to support the evolution of SPLs at the model level [11, 12]. In this context, different semantics for the operation of merging of FMs have been proposed in the literature [26]. For the proof of concept presented in this paper, we consider the merging of FMs as an operation that takes as input a set of FMs and returns a new FM representing, as a

---

\* This work has been partially supported by the European Commission (FEDER) and Spanish Government under CICYT project Web-Factories (TIN2006-00472) and the Andalusian Government project ISABEL (TIC-2533).

minimum, the same set of products than the input FMs. Once this FM representing all products is generated, it may be used as a starting point for driving future development.

*Graph Transformations* are a very mature approach, having been used for 30 years for the generation, manipulation, recognition and evaluation of graphs [25]. Most visual languages can be interpreted as a type of graph (directed, labeled, etc.). This makes graph transformations a natural and intuitive way for transforming models [11][13][21]. Graph transformations are defined in a visual way and are provided with a set of tested tools to define, execute and test transformations. Additionally, graph transformation theory provides a solid formal foundation enabling the checking of interesting formal properties such as confluence, sequential and parallel (in)dependence, etc. [15]. All these characteristics make graph transformations a suitable technology and associated formalism for model refactoring [20][23] and software merging [19][31].

In previous work [27] we detailed our intention of providing automated tool support for FM refactoring using graph transformations. In this paper we present our first results in that direction. In particular, the contribution of this paper is twofold:

- We propose a catalogue of 30 visual rules to merge FMs. In contrast to existing proposals, our catalogue includes rules to describe how to merge FMs including feature attributes and cross-tree constraints.
- We propose using graph transformations as a suitable technology to automate the merging of FMs. In order to show the feasibility of our proposal, we present a prototype implementation of our catalogue of rules using the AGG system [29]. To the best of our knowledge, this is the first approach providing automated support for merging FMs including feature attributes and cross-tree constraints.

The remainder of the paper is structured as follows: in Section 2 the main concepts of feature models and graph transformations are introduced. Our proposal is presented in Section 3. In Section 4, we survey related work. We describe the main challenges remaining for our future work in Section 5. Finally, we summarize our main conclusions in Section 6.

## 2 Preliminaries

### 2.1 Feature Models

Feature Models (FM) [16] are used to model sets of software systems in terms of features and relations among them (see Figure 1). A feature can be defined as an increment in product functionality [5]. FMs are commonly used as a compact representation of all the products of an SPL in terms of features.

A FM is visually represented as a tree-like structure in which nodes represent features, and connections illustrate the relationships between them. Figure 1 depicts a simplified example FM inspired by the mobile phone industry. The model illustrates how features are used to specify and build software for mobile phones. The software loaded in the phone is determined by the features that it supports. The root feature identifies the SPL. The relationships between a parent feature and its child features can be divided into:

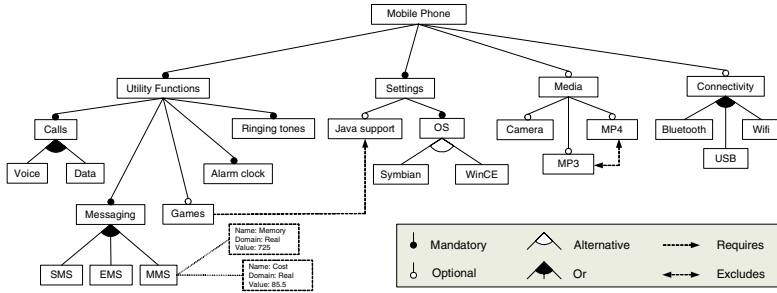


Fig. 1. A sample feature model

- *Mandatory*. If a child feature is mandatory, it is included in all products in which its parent feature appears. Hence, for instance, according to the sample model, all mobile phones must provide support for *ringing tones*.
- *Optional*. If a child feature is defined as optional, it can be optionally included in all products in which its parent feature appears. For instance, the sample feature model defines *games* as an optional feature.
- *Alternative*. A set of child features are defined as alternative if only one feature can be selected when its parent feature is part of the product. As an example, according to the model, a mobile phone will use a *Symbian* or a *WinCE* operating system but not both in the same product.
- *Or-Relation*. A set of child features are said to have an or-relation with their parent when one or more of them can be included in the products in which its parent feature appears. Hence, for instance, according to the sample model, a mobile phone can provide connectivity support for *bluetooth*, *USB*, *wifi* or any combination of the three.

Notice that a child feature can only appear in a product if its parent feature does. The root feature is a part of all the products within the SPL. In addition to the parental relationships between features, a FM can also contain cross-tree constraints between features. These are typically of the form:

- *Requires*. If a feature A requires a feature B, the inclusion of A in a product implies the inclusion of B in such product. Hence, for instance, in the example shown, mobile phones including *games* require *Java support*.
- *Excludes*. If a feature A excludes a feature B, both features cannot be part of the same product. As an example, the SPL represented in Figure 1 removes the possibility of offering support for *MP3* and *MP4* formats in the same product.

Feature Models were first introduced as a part of the Feature-Oriented Domain Analysis method (FODA) by Kang back in 1990 [16]. Since then, multiple extensions to the traditional notation have been proposed in order to increase its expressiveness [7]. In this context, some well known extensions are the so-called *Extended Feature Models* [4,5,6]. Roughly speaking, extended FMs propose adding extra-functional information to the features using attributes. There is no consensus on a notation to define attributes.

However, most proposals agree that an attribute should consist at least of a *name*, a *domain* and a *value*. For instance, the feature 'MMS' in Figure 1 includes some feature attributes using the notation proposed by Benavides *et al.* in [6]. As illustrated, attributes can be used to specify extra-functional information such as cost, speed or RAM memory required to support the feature.

## 2.2 Graph Transformations

*Graph Grammars* are a mature approach for the generation, manipulation, recognition and evaluation of graphs [25]. Graph grammars have been studied and applied in a variety of different domains such as pattern recognition, syntax definition of visual languages, model transformations, description of software architectures, etc. This development is documented in several surveys, tutorials and technical reports [3, 13, 20, 21, 25].

Graph grammars can be considered as the application of the classic string grammar concepts to the domain of graphs. Hence, a graph grammar is composed of an initial graph, a set of terminal labels and a set of transformation rules (sometimes also called graph productions). A transformation rule is composed mainly of a source graph or Left-Hand Side (LHS) and a target graph or Right-Hand Side (RHS). The application of a transformation rule to a so-called host graph, also called direct derivation, consists of looking for an occurrence of the LHS graph in the host graph. If this match is found, the occurrence of the LHS in the graph is replaced by the RHS of the given rule. Thus, each rule application transforms a graph by replacing a part of it by another graph. The set of all graphs labelled with terminal symbols that can be derived from the initial graph by applying the set of transformation rules iteratively is the language specified by the graph grammar.

The application of transformation rules to a given graph is called *Graph Transformation*. Graph transformations are usually used as a general rule-based mechanism to manipulate graphs. Most visual modelling languages can be interpreted as a type of graph (directed, labelled, attributed, etc.). This makes graph transformations recognized as a suitable technology for the specification and application of model transformations [11, 13, 21], model refactoring [20, 23] and software merging [19, 31]. Hence, as documented in the literature, the reasons to select graph transformations as a suitable approach for the transformation and refactoring of visual models are manifold:

- Graph transformations are a natural and intuitive way of performing pattern-based visual model transformations.
- The maturity of graph transformations has provided it with a solid theoretical foundation in the form of useful properties [15, 22]. Hence, for instance, the properties of *sequential* and *parallel dependence* are used to detect the set of transformation rules that must be applied in a given sequence and the set of rules that can be applied in parallel.
- There is a variety of mature tools to define, execute and test transformations rules. Fujaba<sup>1</sup> and the AGG System<sup>2</sup> are two of the most popular general-purpose graph

<sup>1</sup> <http://wwwcs.uni-paderborn.de/cs/fujaba/>

<sup>2</sup> <http://tfs.cs.tu-berlin.de/agg/>

transformation tools within the research community. Nevertheless, other tools such as GREAT<sup>3</sup>, VIATRA2<sup>4</sup> or GROOVE<sup>5</sup> are also starting to emerge as a consequence of the increasing popularity of graph transformations in the model-driven development domain.

### 3 Our Proposal

In this section we present our proposal. In particular, we first propose a catalogue of visual rules describing how to merge FMs. Then, we present a prototype implementation of the proposed catalogue using graph transformations and the AGG system. Finally, we clarify our contribution by means of an example inspired by the mobile phone industry.

#### 3.1 Catalogue of Rules

In Appendix A (see page 503), we present a catalogue of 30 visual, technology-independent rules to describe how to merge FMs. More specifically, our catalogue of *merge rules* describes how to build a FM including all the products represented by two given FMs.

Each merge rule consists of two input patterns of the FMs to be merged (preconditions) and an output pattern of the new FM generated as a result of the merging (postconditions). The rules can be iteratively applied by looking for matches in the input patterns on the two FMs to be merged. A match is an assignment of the variables of the patterns to concrete values. The elements not mentioned in any of the patterns remain unchanged by default. The result of the merging is defined as a new FM including all the products represented by the merged FMs. This resulting FM could be later used as an input model in any other merging operation.

As previously mentioned in Section 1, the merging of FMs makes sense especially when dealing with a set of related products in an SPL domain. Therefore, in order to make this first proposal possible, we make a double assumption:

- Input FMs represent related products using a common catalogue of features. For the sake of simplicity, we assume that features with the same name<sup>6</sup> refer to the same feature and consequently to the same software artifacts.
- The parental relationship between features is equal in all the FMs. That is, a feature must have the same parent feature in all the models in which it appears.

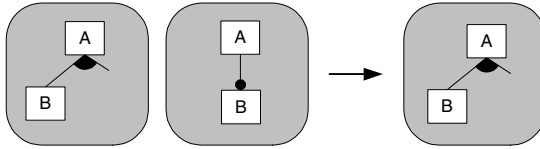
Figure 2 depicts one of the merge rules defined in our catalogue. The input and output patterns of the rule are placed on the left and right side of the arrow respectively. The sample rule illustrates the case in which a feature is defined as a child of an or-relationship and as a mandatory feature in both inputs FMs respectively. As a result of the merging operation, the feature is included in an or-relationship in the resulting FM preserving the configurability options (products) and the grouping structure.

<sup>3</sup> <http://www.escherinstitute.org/Plone/tools>

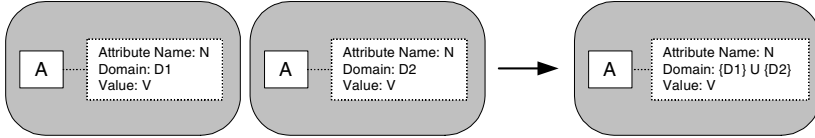
<sup>4</sup> <http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2>

<sup>5</sup> <http://groove.sf.net>

<sup>6</sup> A name could be a string, an identifier, a signature, etc.



**Fig. 2.** A sample merge rule



**Fig. 3.** Merging feature attributes

Providing basic support for merging extended FMs is also a part of our contribution. To this aim, we also define rules describing how to merge feature attributes. As an example, Figure 3 illustrates the case in which a given feature has an attribute with the same name in both input FMs. More specifically, both attributes have the same name and value but different domains. As a result of the merging operation, an attribute with the same name and value is created in the resulting model. The domain of the new attribute consists of the union of the domains of the merged attributes. This way we guarantee that the attribute can take the same values as in the input FMs.

### 3.2 Correctness of the Catalogue

We consider our catalogue of merge rules to be correct if it guarantees that the set of products represented by the resulting FM includes, as a minimum, the set of products represented by the merged FMs. Performing an exhaustive check of the correctness and completeness of the catalogue of rules is out of the scope of this paper. However, in order to perform a preliminary validation of the catalogue, we used FAMA<sup>7</sup> [8], a framework for the edition and automated analysis of FMs. Next, we describe the main steps we followed to test each merge rule:

1. We modelled two example input FMs that matched the input patterns of the merge rule to be validated.
2. We used FAMA to extract the set of products represented by the example input FMs.
3. We created a new FM simulating the application of the given merge rule to the example input FMs.
4. We used FAMA again to extract the set of products represented by the new FM.
5. We finally checked that the set of products represented by the input FMs were included in the set of products represented by the output FM.

<sup>7</sup> <http://www.isa.us.es/fama>



### 3.3 Implementation Using Graph Transformations

In this section we propose using model transformations as an appropriate mechanism to provide automated tool support for merging FMs. In particular, we present a prototype implementation of our catalogue of merge rules for merging FMs using graph transformations.

In order to implement our proposal, we selected a popular tool within the graph grammar community: *The Attributed Graph Grammar System (AGG)*<sup>8</sup> [29]. AGG is a free Java graphical tool for editing and transforming graphs by means of graph transformations. AGG graphs may be typed over a type graph and attributed by Java objects and types. Rule application order may be controlled by dividing rules into layers. Due to its formal foundation, AGG offers validation support for consistency-checking of graph transformation systems according to graph constraints, critical pair analysis to find conflicts between rules [20,22] and checking of termination criteria. All these reasons made us select AGG as a suitable tool to implement our proposal.

AGG graph transformation rules consist of three parts: a left-hand side graph (LHS) and a right-hand side graph (RHS), a mapping between nodes and edges on both sides and a set of Negative Application Conditions (NACs). NACs are preconditions prohibiting certain object structures on the graph from being transformed. Figure 4 shows a screenshot of the AGG GUI. On the left hand side, a tree view displays the working graph and the rules of the proposed grammar. In the upper central area, the NAC (if any) and the LHS and RHS graphs of the selected rule are displayed. Finally, the central area is reserved for the host graph.

Merge rules and graph transformation rules are based on very similar concepts. In particular, both approaches use visual patterns to describe modifications in the structure of a model/graph in terms of pre- and postconditions. Hence, in order to implement our proposal, we mapped our catalogue of merge rules into AGG rules. Next, we outline the main steps we followed to implement our proposal in AGG:

1. Firstly, we defined a set of typed nodes and edges in order to represent FM as graphs. Hence, for instance, we defined a feature as a type of node and an optional relationship as a type of edge. Additionally, we set different visual layouts for the different types of nodes and edges in order to make graphs easily comprehensible. As an example, we used solid and dashed edges to represent mandatory and optional relationships respectively.
2. Next, we created an attributed type graph. From an intuitive point of view, a type graph may be considered as a meta-graph expressing the well-formedness rules that must hold for all graphs. AGG type graphs may include abstract nodes (i.e. not instantiable), node type inheritance and UML-like multiplicities. Graphs obtained as a result of a transformation rule are automatically checked for compliance with the type graph. This way, consistency-checking is automatically performed during the merging process. In the current version of our prototype, the type graph was based on a simplified version of the meta-model for attributed FMs presented in [9].

---

<sup>8</sup> Version 1.6.2.2.



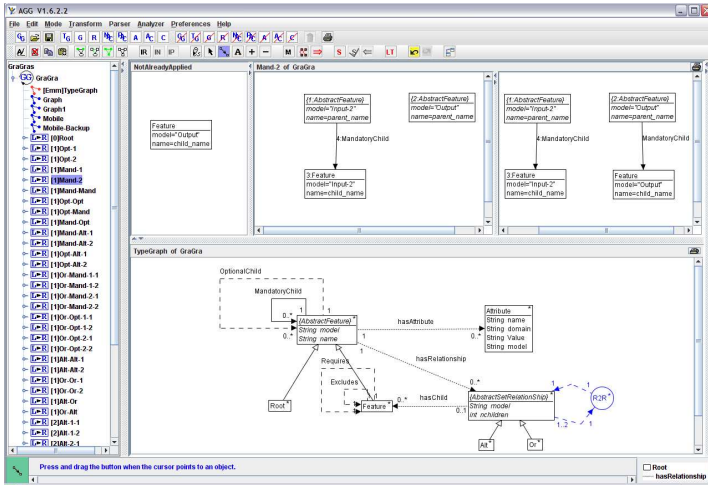


Fig. 4. The AGG System

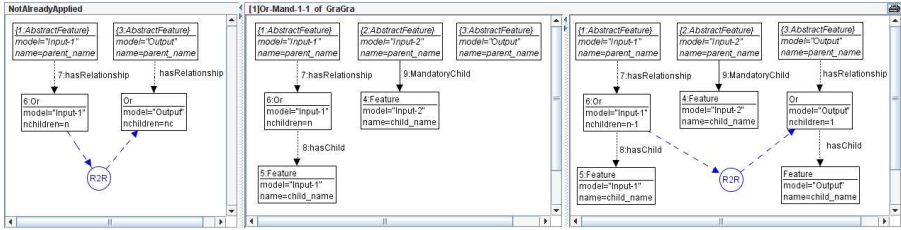
3. The following step was mapping each merge rule into one or more AGG rules. To this aim, we implemented the input and output patterns of the merge rules as the LHS and RHS graphs of the graph transformation rule respectively.
4. In addition to the LHS and RHS graphs, we finally defined additional NACs to restrict when rules can be applied. Hence, for instance, typical NACs avoid the execution of a transformation rule more than once.

As an example, Figure 5 shows a screenshot of the AGG rules used to implement the merge rule presented in Figure 2. From left to right, the NAC, LHS and RHS graphs of each rule are presented. As illustrated, different typed nodes are used to represent features and relationships. In a similar way, different types of edges are used to represent the relationships between features. Additionally, both nodes and edges are provided with attributes which are used to set properties such as the name of features or the kind of FM a node/edge is representing (i.e. input or output).

In addition to the elements of the model, we also used some helper structures and attributes to implement our proposal. This is a common practice when implementing graph transformations [13]. For instance, we used auxiliary nodes (visually represented as circles) to maintain traceability between the or relationships in the input and output graphs. In a similar way, we used attributes to define helpful properties as the number of children of the or relationships (see Figure 5).

Both AGG rules execute the same merging operation on the models but assume a different starting situation. On the one hand, transformation rule 1 is executed when the or-relation has not been created on the output FM yet. In this case, the application of the rule implies the creation of the or-relation. On the other hand, transformation rule 2 illustrates the case in which the or-relation has been previously created as a result of a previous transformation rule. In both cases, a NAC is used to guarantee that the rule is applied to the same elements only once.

## TRANSFORMATION RULE 1



## TRANSFORMATION RULE 2

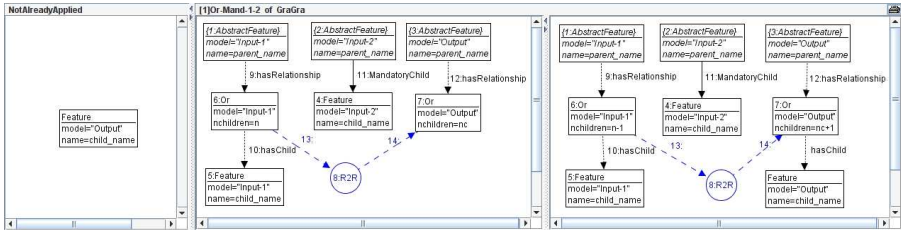


Fig. 5. AGG rules

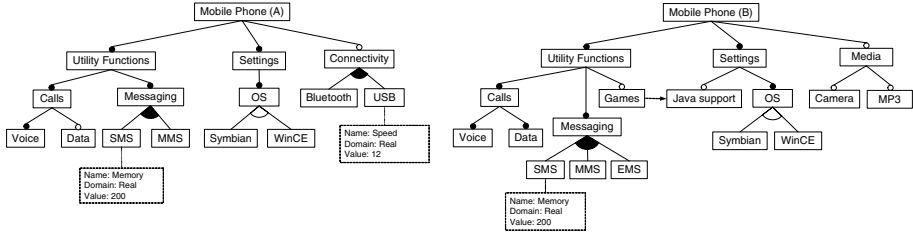
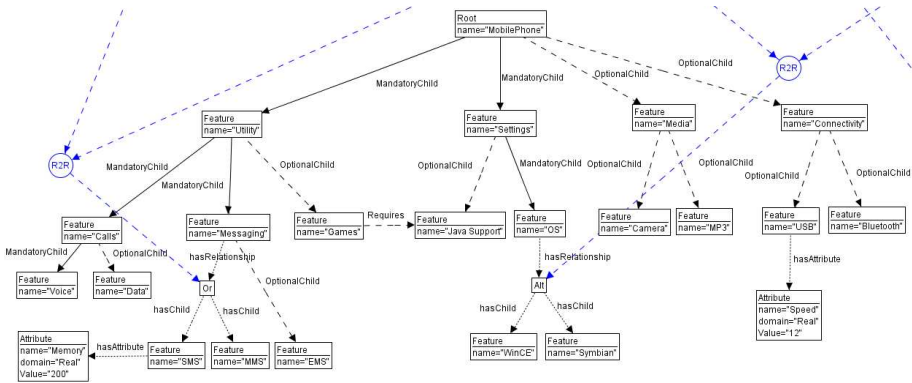


Fig. 6. Feature models to be merged

AGG works exclusively with the graphs created using its editor and not with external models. Thus, input models must be represented as AGG graphs before applying transformations. In a similar way, once the transformation is performed, the obtained graph must be translated to the target model. The automated translation of a FM to an AGG graph and vice versa is out of the scope of this paper. However, since AGG uses XML to store the graphs, we consider XSL transformations could be used as a suitable strategy for the translations model-to-graph and graph-to-model in the context of AGG.

### 3.4 Overview and Running Example

A software company specialized in mobile phone control systems provides two main families of products to its customers. The company has noticed that both families of products share a wide common set of features, and it has decided to combine all of them into an SPL in order to reduce development costs and time-to-market.



**Fig. 7.** Automated merging of feature models in AGG

As a first step, the software architect has decided to design the FM of the SPL as a starting point for driving future development. However, the high number and complexity of existing products make the design of this FM a time-consuming and error-prone activity. As a result of this, the software architect decides to use an automated approach like the one presented in this paper in order to improve the efficiency and reliability of the process. Next, we summarize the main steps he/she should follow to apply or re-use our proposal:

1. Design the catalogue of merge rules to be used. Notice that different application domains could require different merging criteria. In a similar way, other extensions to the traditional notation of FMs could require a different set of merge rules. In our example, the software architect selects our catalogue of rules as a suitable approach to merge extended FMs.
2. Check the correctness of the catalogue. We have proposed using automated analysis of FMs by means of FAMA. However, other alternatives such as using theorem provers may be also feasible [11].
3. Implement the catalogue of rules. We propose using graph transformations and the AGG system as a suitable approach. However, we consider other graph transformation engines could also be used for this purpose. In the context of our example, the software architect decides to use our implementation of the catalogue in AGG.
4. Design a common catalogue of features and use it to model the FMs to be merged. Figure 6 depicts the simplified FMs of the two families of products of the company.
5. Execute the merging process. Figure 7 illustrates a screenshot of the FM generated in AGG as a result of the automated merging process. Notice that the input FMs and some of the attributes are not included due to space constraints.

Once the resulting FM is generated, it could be automatically analysed to extract helpful information such as the number of potential products, commonality of features, set of features of minimum cost, etc [8]. This information could be later used to make relevant design decisions such as selecting the set of features that should be part of the core architecture of the SPL [24].

## 4 Related Work

This work is partially inspired by the proposal of Alves *et al.* [1], in which they motivate the need for refactoring FMs. In their work, the authors propose a catalogue of refactoring rules describing the refactoring operations that can be performed on single FMs. They also motivate the need for merging FMs (what they call *bidirectional refactorings*). Additionally, the authors propose using the automated analysis of FMs, using Alloy as a suitable mechanism, to check the correctness of the catalogue [14]. In contrast to our work, the merging of extended FMs or the automated support for FM refactoring are topics not covered by their proposal. Nevertheless, we presume that graph transformation could be also used as a suitable mechanism to implement their catalogue. This way, their proposal could be complementary to ours for providing a complete tool support for FM refactoring.

Schobbens *et al.* [26] survey feature diagrams variants and generalize the various syntaxes through a generic artifact called *Free Feature Diagrams* (FFD). In their work, the authors identify and define three kinds of merging operations on FMs: *intersection*, *union* and *reduced product*. To the best of our knowledge, they do not provide automated support for the merging of FMs. However, we consider this a complement to our proposal, since it states clearly the semantic of the different merging operations on FMs. In this context, we presume that our proposal could be used to implement any of the identified merging operations by designing an appropriate catalogue of rules.

Czarnecki *et al.* [12] propose an algorithm to compute a FM from a given propositional formula. They point at reverse engineering and merging of FMs as some of the main applications of their approach. In contrast to our work, their algorithm does not support the merging of feature attributes and cross-tree constraints.

Another related work is proposed by Apel *et al.* [2], who present an algebra for Feature-Oriented Software Development (FOSD). As part of their approach, they introduce the so-called Feature Structure Trees (FST) as a mechanism to organize the structural elements of a feature hierarchically. In this context, the authors present a procedure for composing features based on the composition (merging) of FST using tree superimposition. Roughly speaking, tree superimposition describes how to compose trees by starting from the root and proceeding recursively. As in our work, they assume that nodes with the same name refer to the same software artifacts. Compared to our work, they focus on single features instead of complete FMs. In addition, they do not consider cross-tree constraints or feature attributes as we explore in our proposal.

Liu *et al.* study SPL refactoring at the code level and propose what they call Feature Oriented Refactoring (FOR) [18]. They focus on providing a semi-automatic refactoring methodology to enable the decomposition of a program, usually legacy, into features. This approach complements our work, since it could be used as a suitable strategy to obtain the FMs of the legacy systems to be merged into an extractive approach.

## 5 Discussion and Future Work

In this paper we present our first research results toward the automated merging of FMs by using graph transformations. However, there still are many open issues that must be

addressed to provide a solid tool support. In particular, we identify several challenges for our future work:

- Providing formal semantics to our approach. To this aim, we plan to define the merge operation using a formal semantic for FMs [26].
- Validating the catalogue or rules to be correct and complete according to the given semantics. We consider that the tools for the automated analysis of FMs may be helpful for that purpose. However, we also plan to study theorem provers and model checkers such as PVS<sup>9</sup> or Groove<sup>10</sup> for that aim.
- For the proof of concept performed in this paper, we deliberately made some strong assumptions (e.g. feature must have the same name). A more complex approach should allow the merging of FMs including synonym names or different attribute domains. We plan to study the works in the area of the integration of database schemas and ontologies for this aim [28].
- As we previously mentioned, some of the main advantages of using AGG are its mechanisms for consistency-checking and conflict analysis of rule applications. Exploiting massively these mechanisms and especially the critical pair analysis technique to detect conflicts between merge rules [22] is an important part of our on-going research.
- Finally, we plan to make our proposal available by integrating it into the FAMA plug-in [8].

## 6 Conclusions

In this paper we propose using graph transformations as a suitable technology and associated formalism to implement the merging of FMs. In particular, we first presented a catalogue of visual rules to describe how to merge FMs. In this context, we detailed how we used the FAMA plug-in for a basic validation of the catalogue. Then, we introduced a prototype implementation of our catalogue using graph transformations and the AGG system. Finally, we looked at how to apply and re-use our proposal by means of a running example inspired by the mobile phone industry. In contrast to existing proposals, we support the merging of FMs including feature attributes and cross-tree constraints. We also emphasize that our proposal could be extended or adapted to support other merging criteria (e.g. intersection) or FM's notations.

## Acknowledgments

We would like to thank the reviewers of the Second Summer School on Generative and Transformational Techniques in Software Engineering, whose comments and suggestions helped us to improve the paper substantially. We also thank Patrick Heymans for his useful comments.

<sup>9</sup> <http://pvs.csl.sri.com/>

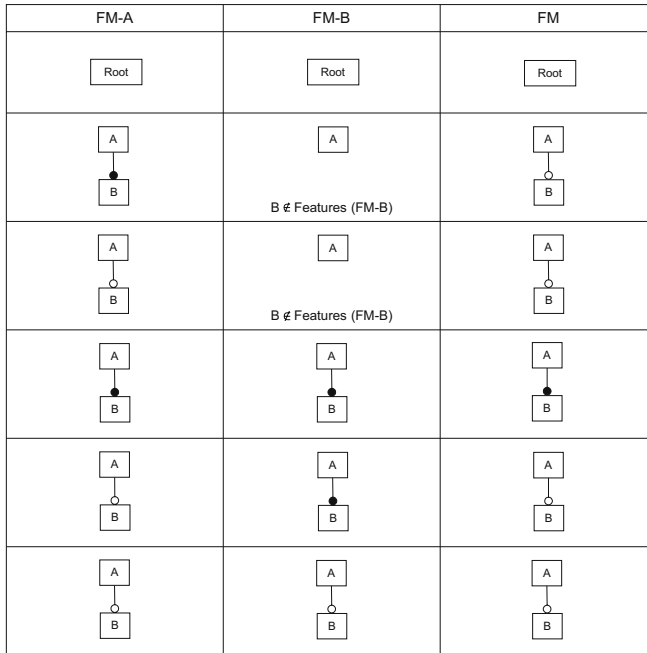
<sup>10</sup> <http://groove.sf.net>

## References

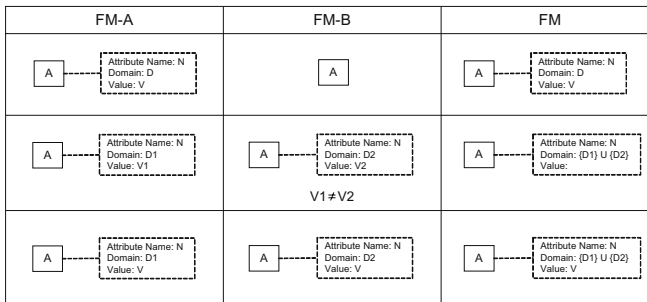
1. Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., Lucena, C.: Refactoring product lines. In: GPCE 2006: Proceedings of the 5th international conference on Generative programming and component engineering, pp. 201–210. ACM Press, New York (2006)
2. Apel, S., Lengauer, C., Batory, D., Möller, B., Kästner, C.: An algebra for feature-oriented software development. Technical Report MIP-0706, Department of Informatics and Mathematics, University of Passau, Germany (July 2007)
3. Baresi, L., Heckel, R.: Tutorial introduction to graph transformation: A software engineering perspective. In: ICGT 2002: Proceedings of the First International Conference on Graph Transformation, London, UK, pp. 402–429. Springer, Heidelberg (2002)
4. Batory, D.: Feature models, grammars, and propositional formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)
5. Batory, D., Benavides, D., Ruiz-Cortés, A.: Automated analysis of feature models: Challenges ahead. *Communications of the ACM* (December 2006)
6. Benavides, D., Ruiz-Cortés, A., Trinidad, P.: Automated reasoning on feature models. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 491–503. Springer, Heidelberg (2005)
7. Benavides, D., Ruiz-Cortés, A., Trinidad, P., Segura, S.: A survey on the automated analyses of feature models. In: *Jornadas de Ingeniería del Software y Bases de Datos (JISBD)* (2006)
8. Benavides, D., Segura, S., Trinidad, P., Ruiz-Cortés, A.: FAMA: Tooling a framework for the automated analysis of feature models. In: *Proceeding of the First International Workshop on Variability Modelling of Software-intensive Systems (VAMOS)*, pp. 129–134 (2007)
9. Benavides, D., Trujillo, S., Trinidad, P.: On the modularization of feature models. In: *First European Workshop on Model Transformation (September 2005)*
10. Clements, P., Northrop, L.: *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley, Reading (2001)
11. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. *IBM Syst. J.* 45(3), 621–645 (2006)
12. Czarnecki, K., Wasowski, A.: Feature diagrams and logics: There and back again. In: *11th International Software Product Line Conference (SPLC 2007)*, pp. 23–34. IEEE Computer Society, Los Alamitos (2007)
13. Ehrig, K., Guerra, E., de Lara, J., Lengyel, L., Levendovszky, T., Prange, U., Taentzer, G., Varró, D., Varró-Gyapay, S.: Model transformation by graph transformation: A comparative study. In: *MTiP 2005, International Workshop on Model Transformations in Practice (Satellite Event of MoDELS 2005)* (2005)
14. Gheyi, R., Massoni, T., Borba, P.: A theory for feature models in alloy. In: *First Alloy Workshop*, pp. 71–80, Portland, United States (November 2006)
15. Heckel, R., Malte Küster, J., Taentzer, G.: Confluence of typed attributed graph transformation systems. In: ICGT 2002: Proceedings of the First International Conference on Graph Transformation, London, UK, pp. 161–176. Springer, Heidelberg (2002)
16. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (November 1990)
17. Krueger, C.W.: Easing the transition to software mass customization. In: *PFE 2001: Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, London, UK, pp. 282–293. Springer, Heidelberg (2002)
18. Liu, J., Batory, D., Lengauer, C.: Feature oriented refactoring of legacy applications. In: *ICSE 2006: Proceeding of the 28th international conference on Software engineering*, pp. 112–121. ACM Press, New York (2006)

19. Mens, T.: Conditional graph rewriting as a domain-independent formalism for software evolution. In: AGTIVE 1999: Proceedings of the International Workshop on Applications of Graph Transformations with Industrial Relevance, London, UK, pp. 127–143. Springer, Heidelberg (2000)
20. Mens, T.: On the use of graph transformations for model refactoring. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2005. LNCS, vol. 4143, pp. 219–257. Springer, Heidelberg (2006)
21. Mens, T., Gorp, P.V., Varró, D., Karsai, G.: Applying a model transformation taxonomy to graph transformation technology. *Electronic Notes in Theoretical Computer Science (ENTCS)* 152, 143–159 (2006)
22. Mens, T., Taentzer, G., Runge, O.: Analysing refactoring dependencies using graph transformation. *Software and Systems Modeling* 6(3), 269–285 (2007)
23. Mens, T., Tourwé, T.: A survey of software refactoring. *IEEE Trans. Softw. Eng.* 30(2), 126–139 (2004)
24. Peña, J., Hinchey, M., Ruiz-Cortés, A., Trinidad, P.: Building the core architecture of a multi-agent system product line: With an example from a future nasa mission. In: 7th International Workshop on Agent Oriented Software Engineering. LNCS (2006)
25. Rozenberg, G. (ed.): *Handbook of Graph Grammars and Computing by Graph Transformations*. Foundations, vol. 1. World Scientific, Singapore (1997)
26. Schobbens, P., Heymans, P., Trigaux, J., Bontemps, Y.: Feature Diagrams: A Survey and A Formal Semantics. In: Proceedings of the 14th IEEE International Requirements Engineering Conference (RE 2006), Minneapolis, Minnesota, USA (September 2006)
27. Segura, S., Benavides, D., Ruiz-Cortés, A., Trinidad, P.: Toward automated refactoring of feature models using graph transformations. In: Pimentel, E. (ed.) VII Jornadas sobre Programación y Lenguajes, PROLE 2007, Zaragoza, Spain, pp. 275–284 (September 2007)
28. Shvaiko, P., Euzenat, J.: A survey of schema-based matching approaches. *Journal on Data Semantics IV*, 146–171 (2005)
29. Taentzer, G.: Agg: A graph transformation environment for modeling and validation of software. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062. Springer, Heidelberg (2004)
30. Trujillo, S., Batory, D., Diaz, O.: Feature refactoring a multi-representation program into a product line. In: GPCE 2006: Proceedings of the 5th international conference on Generative programming and component engineering, pp. 191–200. ACM Press, New York (2006)
31. Westfechtel, B.: Structure-oriented merging of revisions of software documents. In: Proceedings of the 3rd international workshop on Software configuration management, pp. 68–79. ACM, New York (1991)

## Appendix A. Merge Rules



(a) Root and binary relationships



(b) Feature attributes



FM-A	FM-B	FM
	A B ∉ Features (FM-B)	
	A B ∉ Features (FM-B)	

(c) Or-/Alternative relationships

FM-A	FM-B	FM
	A, B ∉ Features (FM-B)	
	A, B ∉ Features (FM-B)	
	 B ∉ Features (FM-B)	
	 A ∉ Features (FM-B)	
	 B ∉ Features (FM-B)	
	 A ∉ Features (FM-B)	

(d) Cross-tree constraints

# Modelling the Operational Semantics of Domain-Specific Modelling Languages

Guido Wachsmuth

Humboldt-Universität zu Berlin  
Unter den Linden 6  
D-10099 Berlin, Germany  
guwac@gk-metrik.de

**Abstract.** Domain-specific modelling languages provide modelling means tailored to a particular domain. In Model-driven Engineering, it is common practice to specify such languages by modelling means as well. In this paper, we investigate structural operational semantics for domain-specific modelling languages. Thereby, we rely completely on standard modelling means as provided by the Object Management Group. As examples, we specify structural operational semantics for Petri nets as well as for a stream-oriented language from the domain of earthquake detection. The approach is useful to provide prototypical tool support for domain-specific modelling languages. It can be instrumented to specify interpreters and debuggers in a generic way.

## 1 Introduction

**Domain-specific modelling languages.** In Model Driven Engineering (MDE), models are the primary engineering artefacts. Models are expressed by means of modelling languages. This includes general-purpose languages like the Unified Modeling Language (UML) [1] as well as domain-specific modelling languages (DSMLs). In contrast to general-purpose languages, DSMLs provide modelling means tailored to a particular domain [2,3]. They include concepts and notations to which experts from this domain are used. This enables domain experts to participate in software engineering by capturing their knowledge in precise and comprehensible models. Requirements for a DSML are often sketchy and evolve over time. Thus, prototypical tool support is needed. Since the application range of a DSML and thus its reuse will usually be limited, generative or generic solutions are preferable. In this paper, we instrument standardised modelling means to specify interpreters and debuggers for DSMLs in a generic way.

**Model-driven language engineering.** In MDE, it is common practice to specify modelling languages by modelling means. Metamodels capture the abstract syntax of these languages. Language semantics are usually expressed by model transformations. Thereby, language instances are transformed into instances of another language. Often, this language is somehow executable, for example a standard programming language like Java. This process of code generation corresponds

to traditional compilation. In this paper, we address model interpretation. Therefore, we adopt the idea of structural operational semantics from grammar-based language engineering [4]. The semantics of a modelling language is not given by a translation but by a transition system. A transition system consists of a set of valid configurations and a transition relation between configurations. For the modelling of transition systems, we rely completely on modelling means standardised by the Object Management Group (OMG). We instrument metamodels to model configuration sets, and use model transformations to specify transition relations.

Our approach has several advantages. First, we stay in the expert domain to describe its semantics in terms of its structure. Domain experts understand these structures and should be able to explain the effects of execution. Second, semantics descriptions are instantly executable in the expert domain. This enables us to test and validate the specified behaviour together with the domain experts. Thus, our approach is well suited for prototyping. Third, we rely only on standard modelling technologies. In a MDE setting, language engineers are used to these. Finally, we help to transfer the good habit to define language semantics formally from traditional DSLs [5] to DSMLs.

**Structure of the paper.** In the next section, we give a brief introduction to metamodels, model transformations, and operational semantics. In Section 3, we define the operational semantics of Petri nets by syntactic manipulation. In Section 4, we discuss the semantics of a stream-oriented language from the domain of earthquake detection as a more sophisticated example. Here, we specify the operational semantics by a transition relation between configurations. In Section 5, we discuss the benefits of our approach in the context of language prototyping and generic tool support for DSMLs. Related work is discussed in Section 6. The paper is concluded in Section 7.

## 2 Preliminaries

**Metamodels.** Metamodels model the structure of models. From a language perspective, they define an abstract syntax for a modelling language. With its MetaObject Facility (MOF) [6], the OMG provides standard description means for metamodels. The MOF is rooted in the UML and reuses its basic concepts like packages, classes, properties, and associations. Constraints expressed in the Object Constraint Language (OCL) [7] can be used to restrict the instance set of a MOF compliant metamodel. Additionally, MOF offers sophisticated features like property redefinition, union sets, and package merge. The semantics of these features was formally defined in [8].

*Example 1 (Petri net metamodel).* Fig. 1 provides a MOF compliant metamodel for Petri nets. A Petri net consists of places and transitions. Each transition has input and output places. Places are marked with a number of tokens. This number is constrained to be non-negative.

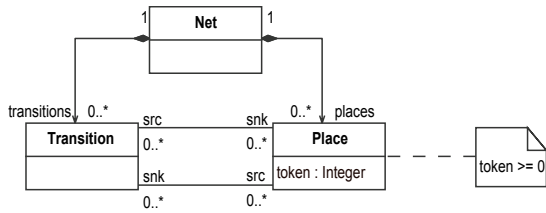


Fig. 1. A metamodel for Petri nets

**Model transformations.** Model transformations take a central place in MDE. They are used to translate models to other models, e.g. platform-independent models into platform-specific ones. Furthermore, model transformations can be instrumented to translate models into an executable language, e.g. Java. The OMG standard for model transformations is Query/View/Transformation(QVT) [9]. QVT defines three model transformation languages: *QVT Relations* and *QVT Core* are declarative languages at two different levels of abstraction. The *QVT Operational Mappings* language is an imperative language. In this paper, we focus on QVT Relations, the high-level declarative language. It extends OCL and its semantics is given in a self-descriptive way by a mapping to QVT Core.

In QVT Relations, a *transformation* needs to declare parameters for holding the models involved in the transformation. These parameters are typed over appropriate metamodels. Upon invocation, the user needs to specify the *direction* for the execution of a transformation. A transformation can be executed in the direction of one parameter. The model held by this parameter is the *target model* of the transformation. All models held by other parameters are *source models*. In special cases, the target model is one of the source models. Then, the transformation is executed as an *in-place* transformation, i.e. the target model results from changing the source model.

A transformation consists of queries and relations. A *query* returns the result of an OCL expression. A *relation* declares several *domains*. A domain is bound to a model held by a parameter. Furthermore, it declares a *pattern* which will be bound to elements from this model. A pattern consists of a variable and a type declaration which may specify some of the properties of this type by sub-patterns. All relations of a transformation which are declared as *top* need to hold. If a relation does not hold, the transformation tries to satisfy it by changing elements in domains which are bound to the target model and declared as *enforce*. Relations can have two kinds of clauses: A *when clause* specifies a condition under which a relation applies. A *where clause* specifies additional constraints among the involved model elements, which may need to be enforced. These concepts are illustrated in the example that follows.

*Example 2 (Queries and relations in QVT Relations).* Figure 2 shows a transformation in QVT Relations. The transformation has two parameters **input** and **output** both typed with the Petri net metamodel given in Ex. 1. As parameter names suggest, we assume the execution of the transformation in direction

```

transformation petri_sos(input:petri, output:petri) {
  top relation run {
    trans: Transition;
    checkonly domain input net:Net{};
    enforce domain output net:Net{};
    where { trans = getActivated(net); fire(trans); }
  }

  relation fire {
    checkonly domain input trans:Transition{};
  }

  query isActivated(trans: Transition): Boolean {
    trans.src -> forAll(place | place.token > 0)
  }

  query getActivated(net: Net): Transition {
    net.transitions -> any(trans | isActivated(trans))
  }

  top relation produce {
    checkonly domain input p:Place{ src = t:Transition{}, token = n:Integer{} };
    enforce domain output p:Place{ token = n+1 };
    when { fire(t); t.src -> excludes(p); }
  }

  top relation consume {
    checkonly domain input p:Place{ snk = t:Transition{}, token = n:Integer{} };
    enforce domain output p:Place{ token = n-1 };
    when { fire(t); t.snk -> excludes(p); }
  }

  top relation preserve {
    checkonly domain input p:Place{ token = n:Integer{} };
    enforce domain output p:Place{ token = n };
    when { not produce(p, p); not consume(p, p); }
  }
}

```

**Fig. 2.** Operational semantics for Petri nets

output. Thus, `input` and `output` will hold the source and target model, respectively. The transformation is designed for *in-place* execution. Thus, both parameters must be bound to the same model. Otherwise, the execution will not yield correct results. The transformation declares the queries `isActivated` and `getActivated`. The first query returns whether a given transition in a Petri net is activated or not. It inspects all input places of the transition. If they are all marked with tokens, the transition is activated and the query returns `true`. The second query returns an activated transition of a given Petri net. Therein, the OCL predicate `any` ensures a non-deterministic choice. Furthermore, the transformation contains several relations. The relation `run` declares two domains. The first domain pattern binds the variable `net` to a Petri net in the source model (held by `input`). The second domain pattern specifies the same Petri net to occur in the target model (held by `output`). The second domain is declared *enforced*. Thus, the net will be created in the target model if it does not already exist. A net in the target model, for which no corresponding net in the source model can be found, will be deleted. In its *where* clause, the relation `run` obtains an activated transition of the net by calling the query `getActivated`. Additionally,

the unary relation `fire` needs to hold for this transition. `fire` simply checks if the transition can be found in the source model. The first relation `run` is declared as *top*. Therefore, it has to hold to process the transformation successfully. Since the relation `fire` is not declared as *top*, it only has to hold to fulfil other relations calling it from their *where* clauses. We will discuss the remaining relations later on.

**Operational semantics.** The operational semantics of a language describes the meaning of a language instance as a sequence of computational steps. Generally, a transition system  $\langle \Gamma, \rightarrow \rangle$  forms the mathematical foundation, where  $\Gamma$  is a set of *configurations* and  $\rightarrow \subseteq \Gamma \times \Gamma$  is a *transition relation*. Inference rules are a common way to define the valid transitions in the system inductively. Plotkin pioneered these ideas. In his work on structural operational semantics [4], he proposed to describe transitions according to the abstract syntax of the language. This allows for reasoning about programs by structural induction and correctness proofs of compilers and debuggers [10]. The structural operational semantics of a language defines an interpreter for this language working on its abstract syntax. This can be instrumented as a reference to test implementations of compilers and interpreters. Starting from Plotkin’s ideas, structural operational semantics became very popular in traditional grammar-based language engineering [11][12]. In this paper, we apply the idea of structural operational semantics to model-driven language engineering. Thereby, we rely only on standard modelling techniques: Configuration sets are captured in MOF compliant metamodels. Transition relations are specified in QVT Relations.

### 3 Syntactic Manipulation: Petri Nets

For some simple languages, configurations can be expressed in the language itself. Thus, computational steps can be expressed as purely syntactic manipulation of language instances. A well-known example for this is the lambda calculus [13]. Petri nets provide another example. In this section, we give an operational semantics for Petri nets based on the metamodel given in Fig. 1. A computation step in a Petri net chooses an activated transition non-deterministically and fires it. The marking of a place is

- (i) Increased by one token iff it is only an output place,
- (ii) Decreased by one token iff it is only an input place,
- (iii) Preserved otherwise.

The transformation given in Fig. 2, specifies these semantics in QVT Relations. We discussed its relations `run` and `fire` as well as the queries `getActivated` and `isActivated` already in Ex. 2. The relations `produce`, `consume`, and `preserve` cover the different cases for places:

- (i) The relation `produce` matches a place `p` in the input, an incoming transition `t` of this place, and its number of tokens `n`. It enforces an increase of the

number of tokens in the output by one. The relation needs to hold only if the matched transition is fired and if the matched place is not an input place of this transition.

- (ii) The relation **consume** works similarly. It matches a place in the input, an outgoing transition of this place, and its number of tokens. The number of tokens in the output is decreased by one. The relation only has to hold if the transition is fired and if the place is not an output place of the transition.
- (iii) Otherwise, **preserve** keeps places unchanged.

## 4 Configuration Transitions: A Stream-Oriented Language

Describing operational semantics by syntactic manipulation works only for simple languages. In this section, we provide operational semantics for a stream-oriented language by transitions between configurations. We rely on MOF to define a metamodel for configurations and on QVT Relations to capture the transition relation between configurations.

**The language.** Our group develops technologies for earthquake early warning systems. Earthquake detection algorithms form an integral part of such systems. Developing such algorithms requires knowledge from the domain of seismology. To involve domain experts in the system engineering process, we provide them with domain-specific modelling means [14].

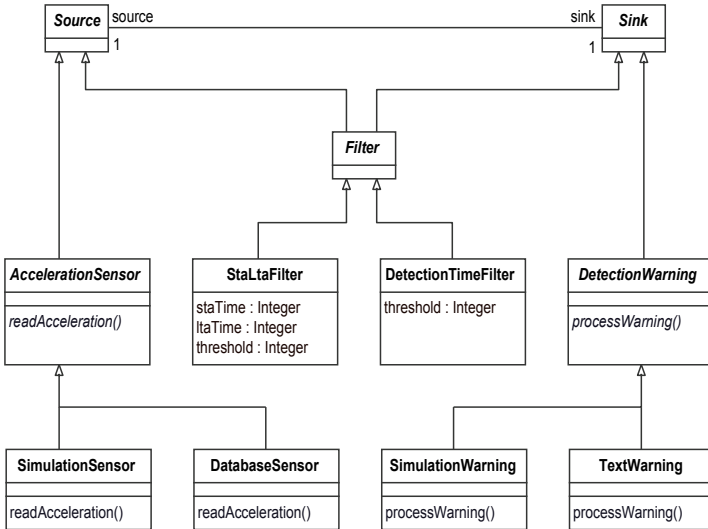


Fig. 3. A metamodel for a stream-oriented language



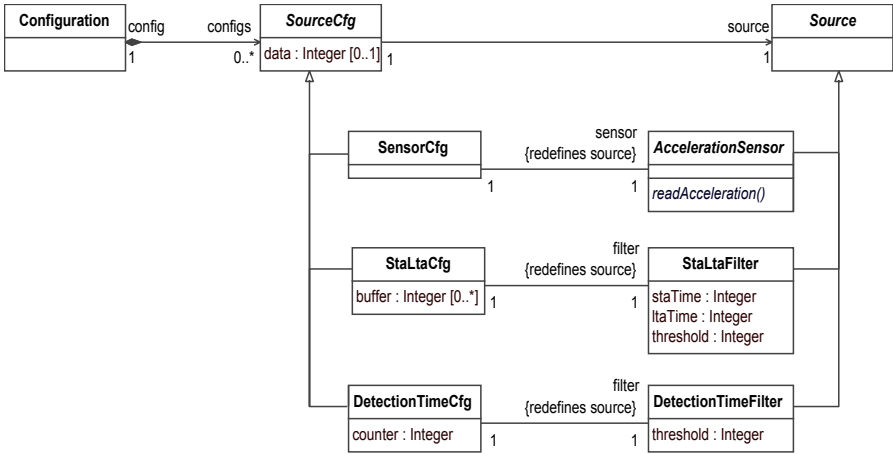


Fig. 4. A metamodel for configurations

Seismologists might think of earthquake detection in terms of streams: Sensors produce streams of measured data, filters process these streams, and sinks consume the provided data. Fig. 3 shows a MOF compliant metamodel for a stream-oriented language. In this paper, we mention only one kind of source, that is acceleration sensors, and only one kind of sink, that is detection warnings.

For acceleration sensors, various implementations of `readAcceleration()` can be used to provide sensor readings. For example, the method can connect to a database of sensor readings stored during former earthquake events. Alternatively, simulation of a wave propagation model might calculate the data needed. In the same way, `processWarning()` can be customised to write detection warnings to a file or to provide it to other simulations, e.g. for evacuation.

Filters process data. Therefore, a filter acts as a source and as a sink. We consider two kind of filters: The Short Term Averaging/Long Term Averaging (STA/LTA) detection algorithm [15] is realised by STA/LTA filters. The idea of this algorithm is to determine the acceleration average in the short and the long term. If the ratio between both averages exceeds a threshold, an earthquake is detected. The filters can be configured with values for the STA time, the LTA time, and the threshold. Detection time filters refrain from forwarding data for a certain amount of time. Once a filter forwards data from its source, it stops considering data from its source for the specified amount of time.

**Configurations.** Configurations form the central concept for the structural operational semantics of the stream-oriented language. As a first step, we need to define the set of valid configurations by a metamodel as shown in Fig. 4. The metamodel defines a configuration for each kind of sources in the stream-oriented language. Generally, the configuration for a source contains the current data provided by the source. Additionally, configurations for STA/LTA filters keep a buffer of input data. Detection time filter configurations store a time counter.

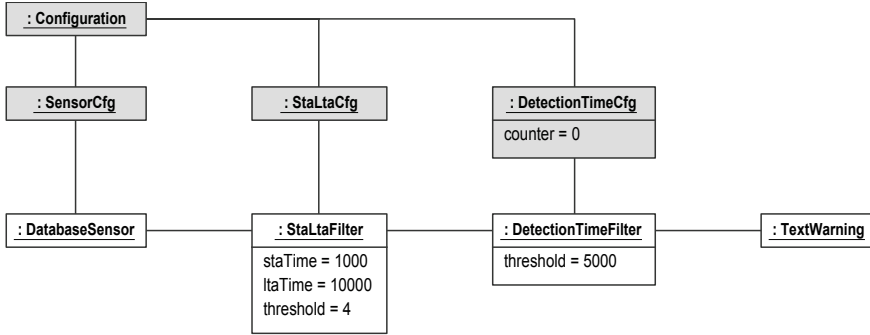


Fig. 5. Initial configuration of a stream-oriented model

Figure 5 shows an abstract representation of an initial configuration model for an example stream-oriented model. Here, model elements from the stream-oriented model are filled white while model elements from the configuration model are filled grey.

**Transition relation.** The operational semantics of the stream-oriented language is now expressed as a transition relation between configurations in time. For simplicity, we assume equidistant time steps. We define the transition relation in QVT Relations. As in the Petri net example, the transformation changes a configuration in-place. Its relations and queries are given in Fig. 6. We will now discuss the details.

In each step, sensors need to provide new acceleration data. The configuration of a sensor is updated by the new value. The relation `readSensor` defines this behaviour. It matches a sensor configuration in the input and in the output model. The output configuration is enforced to contain new data read from the sensor.

If a source provides data to a warning sink, a detection warning has to be processed. The behaviour is specified by the relation `processWarning`. It matches source configurations in the input which provide data and for which the sink of the configured source is a warning sink. The configuration is preserved in the output model. As specified in the *where* clause, the warning is processed by calling the corresponding method on the sink.

The relation `processStaLta` matches the configuration for a STA/LTA filter in the input and the output model. Furthermore, the corresponding filter, its source, and the configuration of the source are matched in the input model. For the output configuration, new values for the buffer and the filter data are enforced. The new buffer is empty if the source does not provide any data. Otherwise, the new buffer consists of the data provided from the source and the values from the old buffer. If the new buffer overflows, older elements will be removed from the buffer. The ratio between short and long term average is calculated in terms of the new buffer. If it exceeds the specified threshold of the filter, it will be used as the new filter data. Otherwise, the filter does not provide any data.

```

top relation readSensor {
  checkonly domain input cfg:SensorCfg{ sensor = sensor:AccelerationSensor{} };
  enforce domain output cfg:SensorCfg{ data = sensor.readAcceleration() };
}

top relation processWarning {
  checkonly domain input cfg:SourceCfg{
    source = src:Source{ sink = warn:DetectionWarning{} }, data = d:Integer{}
  };
  enforce domain output cfg:SourceCfg{};
  where { warn.processWarning(); }
}

top relation processStaLta {
  checkonly domain input fcfg1:StaLtaCfg{
    filter = filter:StaLtaFilter{
      source = src:Source{}, staTime = sta:Integer{},
      ltaTime = lta:Integer{}, threshold = threshold:Integer{}
    },
    config = cfg:Configuration{ configs = scfg:SourceCfg{ source = src } }
  };
  enforce domain output fcfg2:StaLtaCfg{
    buffer = fillBuffer(fcfg1.buffer, lta, scfg.data),
    data = provideRatio(calculateRatio(fcfg2.buffer, sta, lta), threshold)
  };
  when { fcfg1 = fcfg2; }
}

query fillBuffer(buffer: Sequence(Integer), size: Integer, value: Integer): Sequence(Integer) {
  if value -> oclIsUndefined() then Sequence{} else
    trimBuffer(buffer -> prepend(value), size)
  endif
}

query trimBuffer(buffer: Sequence(Integer), size: Integer): Sequence(Integer) {
  if buffer -> size() <= size then buffer else buffer -> subSequence(1, size) endif
}

query calculateRatio(buffer: Sequence(Integer), sta: Integer, lta: Integer): Integer {
  if buffer -> size() >= lta then
    trimBuffer(buffer, sta) -> sum() div trimBuffer(buffer, lta) -> sum()
  else
    undefined
  endif
}

query provideRatio(ratio: Integer, threshold: Integer): Integer {
  if ratio >= threshold then ratio else undefined endif
}

top relation processDetectionTime {
  checkonly domain input tcfg:DetectionTimeCfg{
    filter = filter:DetectionTimeFilter{ source = src:Source{}, waitTime = wait:Integer{} },
    counter = 0,
    config = cfg:Configuration{
      configs = scfg:SourceCfg{ source = src, data = data:Integer{} }
    }
  };
  enforce domain output tcfg:DetectionTimeCfg{ data = data, counter = wait };
}

top relation updateDetectionTime {
  checkonly domain input tcfg:DetectionTimeCfg{ counter = cnt:Integer{} };
  enforce domain output tcfg:DetectionTimeCfg{ data = undefined, counter = (cnt - 1).max(0) };
  when { not processDetectionTime(tcfg, tcfg); }
}

```

**Fig. 6.** Operational semantics for the stream-oriented language

The relations `processDetectionTime` and `updateDetectionTime` specify the desired behaviour of detection time filters. The former relation matches a configuration for a detection time filter with a reset counter, its source, the configuration of this source in the input model, and the provided data. In the new configuration, this data is provided by the filter and the counter is started. The latter relation updates the internal counter and ensures that no data is provided by the filter. It only has to hold if the first relation fails. The counter is decreased by one until it equals zero.

## 5 Applications

**Interpreters.** Modelling the structural operational semantics of a language with MOF and QVT Relations provides a generic way for interpretation. For a compliant metamodel, the MOF standard defines how to serialise the abstract representation of models. This ensures tool interoperability. Thus, we can load any model compliant to a metamodel together with the QVT transformation describing the language's semantics into a QVT engine. Then, execution of the transformation interprets the model.

*Example 3 (Interpreting stream-oriented models).* We want to interpret the model from Fig. 5. Therefore, we load the transformation specifying the operational semantics of the stream-oriented language into a QVT engine. We discussed this transformation in the preceding section. Then, we apply the transformation to the initial configuration of the stream-oriented model. This results in a new configuration. We can instruct the QVT engine to apply the transformation until the configuration is not affected anymore. During the execution, the transformation will call the method `readAcceleration` to read sensor data from a database. If an earthquake is detected, the transformation will call the method `processWarning` to present a textual warning to the user.

We can benefit from the direct executability for language prototyping. Structural operational semantics descriptions provide a generic way to specify interpreters. The specification depends on the structure of the language and can be expressed in the particular modelling domain which the language addresses. Thereby, execution takes place in the expert domain. In contrast to translational semantics, error and debug information remain in the modelling domain. Thus, domain experts can be involved into a prototyping cycle for language semantics.

Once the semantics meet the expectations of domain experts, equivalent semantics can be implemented in another style. Usually, translational semantics will be preferred to include platform-dependent information or for performance issues. We can test these semantics as well as generated or hand written language processors against the generic interpreter. Thereby, the interpreter acts as a reference. Then, generated code and hand written processors should lead to the same result as specified in the operational semantics.

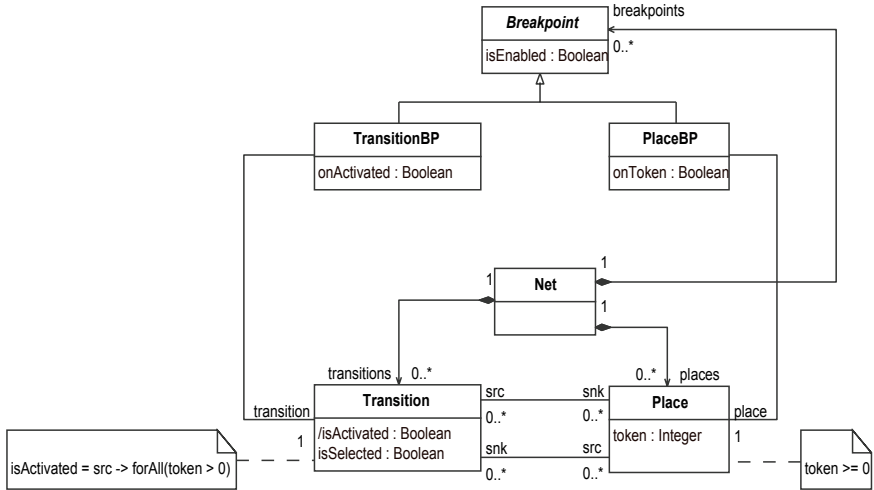


Fig. 7. Petri nets metamodel extended for debugging purposes

**Debuggers.** Debugging differs from interpretation in two ways: The user is interested in the internal state of the execution, and the user wants to control the execution. With our approach, we can extend operational semantics to offer generic support for debugging. The internal state of the execution is represented by configurations. In order to provide debugging information, the metamodel for configurations has to include this information. The introduction of derived attributes can provide such extra information not needed for interpretation.

*Example 4 (Debugging Petri nets).* When we debug Petri nets, we want to know if a transition is currently activated or not. Furthermore, we want to trigger which transition fires in the execution step. We can easily extend the metamodel from Fig. 1 to include these features. The new metamodel is shown in Fig. 7. For transitions, we introduce a derived attribute `isActivated` which indicates the activation of the transition. Additionally, we introduce an attribute `isSelected`. Only selected transitions will be fired. Furthermore, we specify breakpoints. There are two kinds of breakpoints: Transition breakpoints interrupt the execution if the referred transition becomes activated or inactivated. Place breakpoints interrupt the execution if the referred place is marked with the specified number of tokens.

We need to modify the transition relation in two ways: First, the query `isActivated` must be expressed in terms of the new attributes. Second, additional relations that interrupt the execution at breakpoints are required. We omit the result due to space limitations.

In general, the configuration metamodel needs only a few extensions for debugging purposes. Usually, the runtime state already provides many of the information needed for debugging. Only a few additions are needed particular for

debugging, e.g. breakpoints or watch expressions. Since debugging affects the control flow of the execution, more changes are needed for the transition relation. Though, parts concerned with basic language constructs will usually stay unchanged.

**Implementation.** We implemented our approach in the Eclipse Modeling Framework [16]. Furthermore, we rely on a QVT Relations engine provided by ikv [17], an industrial partner of our research group. We developed abstract interpreters and debuggers for various kinds of Petri nets, an extended version of the stream-oriented language, and for a scripting language for metamodel adaptation [18]. The latter language provides over 50 language concepts. Its operational semantics is defined by over 200 relations.

Efficiency of the developed interpreters and debuggers depends basically on the efficiency of the QVT engine. The engine we use is designed for industrial projects. Amongst others, it was applied in a project between ikv and the biggest consumer electronics vendor in Korea in the area of embedded systems. The engine provides an efficiency which is adequate for prototyping purposes. For the metamodel adaptation language, an adaptation script of over 40 adaptation steps is interpreted in less than a second.

Furthermore, we provide a combination of operational semantics and model-based visualisation techniques. The approach allows for the prototyping of graphical interpreters and debuggers for visual DSMLs [19]. As a technological foundation, we rely on the Eclipse Modeling Framework and the Graphical Modeling Framework for the Eclipse platform.

## 6 Related Work

**Translational approaches.** Model-to-model transformations are a common way to define the semantics of a language model by translation into a target language. Usually, this ends in an executable model, e.g. a program in a general purpose programming language. For example, the Model Driven Architecture [20], the OMG's standard for MDE, suggests the transformation of platform independent models into platform specific models. By providing standard means for these transformations, QVT forms the center of this approach.

Translational semantics are harmful in a domain-specific context. The corresponding transformations are specified in terms of the source and the target language model. While we can expect domain experts to understand the source language model, we cannot do so with respect to the target language. Furthermore, complicated translations hide the underlying language semantics in the details of the target language. Executing the target program, another problem arises. To assist domain experts, error messages and debugging information need to be mapped into the source domain which is a sophisticated task. Thus, translational semantics do not provide an appropriate level of abstraction for prototyping purposes.

An operational approach is more suited to testing, validation, and even formal verification than the translational approach. We stay in the expert domain to

describe the operational semantics of the language in terms of its structure. The domain experts understand these structures and should be able to explain the effects of execution. Furthermore, the structural operational semantics description is instantly executable in the expert domain. This enables us to test and validate the specified behaviour together with the domain experts. Thus, our approach is well suited for prototyping.

Nevertheless, translational semantics are needed in addition. They allow for the integration of platform dependent details, can match a particular target platform, can address existing tools and provide a higher efficiency.

**Semantics description languages.** Several approaches address a metamodel based formalisation of language semantics. All these provide their own description means, sometimes a variation or extension of existing modelling languages. Hausmann et al. describe the operational semantics of UML behaviour diagrams in terms of collaboration diagrams and graph transformations [21,22]. Sunyé et al. recommend UML action semantics for executable UML models [23]. Furthermore, they suggest activities with action semantics for language modelling. Scheidgen and Fischer follow this suggestion and provide description means for the operational semantics of MOF compliant language models [24].

Other approaches originate in language development frameworks. The AMMA framework integrates Abstract State Machines for the specification of execution semantics [25]. Muller et al. integrate OCL into an imperative action language [26] to provide semantics description means for the Kermet framework [27]. In a similar way, OCL is extended with actions to provide semantics description means in the Mosaic framework [28].

All these approaches have in common the fact that they use languages particularly designed for semantics description. Users have to learn these languages in order to use a certain tool. Then, they are restricted to a certain tool, its language modelling formalism, and a particular way to describe the semantics of the intended language. In contrast, our approach relies only on standard modelling techniques. Both, MOF and QVT are OMG standards. One goal of the OMG standards is to provide interoperability between various tools. In a MDE setting, users are accustomed to apply these techniques. They are used to define model transformations, e.g. for translational semantics. With our approach, they can benefit from language engineering knowledge without switching to a different technology space.

## 7 Conclusion

**Contribution.** In this paper, we demonstrated how to define operational semantics of a language in a model-driven way. Thereby, only standard modelling techniques were applied. The abstract syntax of a language as well as runtime configurations were captured in MOF compliant models. The operational semantics were defined inductively in a declarative way by means of QVT Relations. Furthermore, we discussed the applications of our approach in the areas of language prototyping.

**Future work.** We plan to integrate the work presented in the paper with our existing work on metamodel adaptation and model co-adaptation [18]. That work is concerned with well-defined adaptation steps for metamodels similarly to object-oriented refactoring. Adaptation steps are performed by transformation. Language instances are automatically co-adapted. It provides a first step to agile model-driven language engineering. Generic support for execution as mentioned in this paper is a second step. To integrate both steps, we need to consider co-adaptation of semantics. So far, co-adaptation of language instances only concerns the syntax. When adapting a metamodel, language instances are co-adapted to comply to the new metamodel. Considering semantics, language instances and semantics descriptions need to be co-adapted in a way that preserves the meaning of those instances. Furthermore, adaptation of semantics is a topic on its own. Here, we can think of transformations that extend, preserve, or restrict semantics descriptions.

**Acknowledgement.** This work is supported by grants from the DFG (German Research Foundation, Graduiertenkolleg METRIK). The author is indebted to the anonymous referees for valuable comments as well as to Eckhardt Holz, Daniel Sadilek, and Markus Scheidgen for encouraging discussion and helpful suggestions. Daniel Sadilek invented the first versions of the stream-oriented language used as the second example in this paper. The author is thankful to Hajo Eichler, Omar Ekine, and Jörg Kieglend for help with ikv's QVT engine.

## References

1. Object Management Group: Unified Modeling Language: Infrastructure, version 2.0 (July 2005)
2. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling. IEEE Computer Society, Los Alamitos (2008)
3. Cook, S.: Domain-specific modeling. Microsoft Architect Journal 9 (August 2006)
4. Plotkin, G.D.: A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus (1981)
5. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. SIGPLAN Not 35(6), 26–36 (2000)
6. Object Management Group: Meta Object Facility Core Specification, version 2.0 (January 2006)
7. Object Management Group: Object Constraint Language, version 2.0 (May 2006)
8. Alanen, M., Porres, I.: Basic operations over models containing subset and union properties. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 469–483. Springer, Heidelberg (2006)
9. Object Management Group: MOF Query/View/Transformation, Final Adopted Specification (July 2007)
10. da Silva, F.Q.B.: Correctness Proofs of Compilers and Debuggers: an Approach Based on Structural Operational Semantics. PhD thesis, University of Edinburgh (1992)
11. Nielson, H.R., Nielson, F.: Semantics with Applications: A Formal Introduction. Wiley, Chichester (1992)



12. Aceto, L., Fokink, W., Verhoef, C.: Structural operational semantics. In: Bergstra, J., Ponse, A., Smolka, S. (eds.) *Handbook of Process Algebra*. Elsevier, Amsterdam (2001)
13. Barendregt, H.: *The Lambda Calculus its Syntax and Semantics*, 2nd edn. North Holland, Amsterdam (1987)
14. Sadilek, D., Theisselmann, F., Wachsmuth, G.: Challenges for model-driven development of self-organising disaster management information systems. In: *IRTGW 2006: Proceedings of the International Research Training Groups Workshop*, Dagstuhl, Germany, Berlin, GITO-Verlag, pp. 24–26 (November 2006)
15. Stewart, S.W.: Real time detection and location of local seismic events in central California. *Bull. Seism. Soc. Am.* 67, 433–452 (1977)
16. Budinsky, F., Merks, E., Steinberg, D.: *Eclipse Modeling Framework*, 2nd edn. Addison-Wesley, Reading (2006)
17. ikv: Company home page (2007), <http://www.ikv.de>
18. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In: Ernst, E. (ed.) *ECOOP 2007*. LNCS, vol. 4609. Springer, Heidelberg (2007)
19. Sadilek, D.A., Wachsmuth, G.: Prototyping visual interpreters and debuggers for domain-specific modelling languages. In: Schieferdecker, I., Hartman, A. (eds.) *ECMDA-FA 2008*. LNCS, vol. 5095. Springer, Heidelberg (2008)
20. Object Management Group: *MDA Guide Version 1.0.1* (June 2003)
21. Hausmann, J.H.: *Dynamic meta modeling: A semantics description technique for visual modeling languages*. PhD thesis, University of Paderborn (2005)
22. Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in uml. In: Evans, A., Kent, S., Selic, B. (eds.) *UML 2000*. LNCS, vol. 1939, pp. 323–337. Springer, Heidelberg (2000)
23. Sunyé, G., Pennaneach, F., Ho, W.M., Guennec, A.L., Jéquel, J.M.: Using uml action semantics for executable modeling and beyond. In: Dittrich, K.R., Gerpert, A., Norrie, M.C. (eds.) *CAiSE 2001*. LNCS, vol. 2068, pp. 433–447. Springer, Heidelberg (2001)
24. Scheidgen, M., Fischer, J.: Human comprehensible and machine processable specifications of operational semantics. In: Akehurst, D.H., Vogel, R., Paige, R.F. (eds.) *ECMDA-FA 2007*. LNCS, vol. 4530, pp. 157–171. Springer, Heidelberg (2007)
25. Di Ruscio, D., Jouault, F., Kurtev, I., Bezivin, J., Pierantonio, A.: Extending AMMA for supporting dynamic semantics specifications of DSLs. Technical Report HAL - CCSD - CNRS, Laboratoire D'Informatique de Nantes-Atlantique (2006)
26. Muller, P., Fleurey, F., Jézéquel, J.: Weaving executability into object-oriented meta-languages. In: Briand, L.C., Williams, C. (eds.) *MoDELS 2005*. LNCS, vol. 3713, pp. 264–278. Springer, Heidelberg (2005)
27. Fleurey, F., Drey, Z., Vojtisek, D., Faucher, C.: *Kermeta language* (October 2006)
28. Clark, T., Evans, A., Sammut, P., Willans, J.: *Applied metamodelling: A foundation for language driven development* (September 2004), [www.xactium.com](http://www.xactium.com)

## Author Index

- Antkiewicz, Michał 3  
Avgustinov, Pavel 78
- Benavides, David 489
- Chellappa, Srinivas 196  
Costanza, Pascal 396  
Czarnecki, Krzysztof 3
- Dekeyser, Jean-Luc 459  
de Moor, Oege 78
- Ekman, Torbjörn 78
- Franchetti, Franz 196
- Gašević, Dragan 377  
Giurca, Adrian 377
- Hajiyev, Elnar 78  
Haupt, Michael 396  
Hirschfeld, Robert 396
- Jarzabek, Stan 47  
Juhász, Zoltán 474
- Lukichev, Sergey 377
- Mali, Yogesh 442  
Marquet, Philippe 459  
Milanović, Milan 377
- Oliveira, José N. 134  
Ongkingco, Neil 78
- Püschel, Markus 196  
Piel, Éric 459  
Porkoláb, Zoltán 474
- Ribarić, Marko 377  
Ruiz-Cortés, Antonio 489
- Segura, Sergio 489  
Sereni, Damien 78  
Sipos, Ádám 474  
Stevens, Perdita 408
- Taha, Walid 260  
Tibble, Julian 78  
Tratt, Laurence 425  
Trinidad, Pablo 489
- Van Wyk, Eric 442  
Verbaere, Mathieu 78  
Visser, Eelco 291
- Wachsmuth, Guido 506  
Wagner, Gerd 377