

Lecture Notes in Computer Science

Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

2487

Springer

Berlin

Heidelberg

New York

Barcelona

Hong Kong

London

Milan

Paris

Tokyo

Don Batory Charles Consel
Walid Taha (Eds.)

Generative Programming and Component Engineering

ACM SIGPLAN/SIGSOFT Conference, GPCE 2002
Pittsburgh, PA, USA, October 6-8, 2002
Proceedings



Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Don Batory
University of Texas at Austin, Department of Computer Sciences
2.124 Taylor Hall, TX 78712 Austin, USA
E-mail: batory@cs.utexas.edu

Charles Consel
University of Paris IV, Department of Telecommunications, ENSEIRB
1, Ave. du Docteur Albert Schweitzer, 33402 Talence Cedex, France
E-mail: consel@enseirb.fr

Walid Taha
Rice University, Department of Computer Science
Mail Stop 132, 6100 S. Main Street, 77005-1892 Houston, Texas, USA
E-mail: taha@cs.rice.edu

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Generative programming and component engineering : ACM SIGPLAN SIGSOFT
conference ; proceedings / GPCE 2002, Pittsburgh, PA, USA, October 6 - 8, 2002.
Don Batory ... (ed.). - Berlin ; Heidelberg ; New York ; Hong Kong ; London ;
Milan ; Paris ; Tokyo : Springer, 2002
(Lecture notes in computer science ; Vol. 2487)
ISBN 3-540-44284-7

CR Subject Classification (1998): D.2, K.6, J.1, D.1, D.3

ISSN 0302-9743

ISBN 3-540-44284-7 Springer-Verlag 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-Verlag. Violations are liable for prosecution under the German Copyright Law.

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

<http://www.springer.de>

© Springer-Verlag Berlin Heidelberg 2002
Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP Berlin, Stefan Sossna e.K.
Printed on acid-free paper SPIN: 10870627 06/3142 5 4 3 2 1 0

Preface

This volume constitutes the proceedings of the first ACM SIGPLAN/SIGSOFT International Conference on Generative Programming and Component Engineering (GPCE 2002), held October 6–8, 2002, in Pittsburgh, PA, USA, as part of the PLI 2002 event, which also included ICFP, PPDP, and affiliated workshops.

The future of Software Engineering lies in the automation of tasks that are performed manually today. Generative Programming (developing programs that synthesize other programs), Component Engineering (raising the level of modularization and analysis in application design), and Domain-Specific Languages (elevating program specifications to compact domain-specific notations that are easier to write and maintain) are key technologies for automating program development. In a time of conference and workshop proliferation, GPCE represents a counter-trend in the merging of two distinct communities with strongly overlapping interests: the Generative and Component-Based Software Engineering Conference (GCSE) and the International Workshop on the Semantics, Applications, and Implementation of Program Generation (SAIG). Researchers in the GCSE community address the topic of program automation from a contemporary software engineering viewpoint; SAIG correspondingly represents a community attacking automation from a more formal programming languages viewpoint. Together, their combination provides the depth of theory and practice that one would expect in a premier research conference.

Three prominent PLI invited speakers lectured at GPCE 2002: Neil Jones (University of Copenhagen), Catuscia Palamidessi (Penn State University), and Janos Sztipanovits (Vanderbilt University).

GPCE 2002 received 39 submissions, of which 18 were accepted. The topics included generative programming, metaprogramming and program specialization, program analysis and program transformation, domain-specific languages, software architectures, and aspect-oriented programming. Two different program committees, each representative of the GCSE and SAIG communities, jointly reviewed the submissions. The vast majority of the papers could have been submitted to either conference had a merger not occurred. Referees from *both* communities agreed on the acceptance of such papers, and in no case was there disagreement about such papers. More than anything else, this unity of viewpoint underscores the need for community integration.

We thank the participants for their excellent contributions, and the program committee members for their efforts. The conference received financial support from the National Science Foundation (NSF), grant number CCR-0215394, Sun Microsystems, and Avaya Labs.

Program Committee (GCSE Focus)

Don Batory, U. T. Austin (Chair)	Peter Knauber, Fraunhofer Institute
Jan Bosch, U. Groningen	Hausi Müller, U. Victoria
Greg Butler, Concordia U.	Nenad Medvidovic, U. S. California
Prem Devanbu, U. C. Davis	Wolfgang Pree, U. Salzburg
Cristina Gacek, U. Newcastle u. Tyne	Yannis Smaragdakis, Georgia Tech
Stan Jarzabek, N. U. Singapore	Douglas R. Smith, Kestrel Institute
Kyo Kang, Pohang U.	

Program Committee (SAIG Focus)

Craig Chambers, U. Washington	Gregor Kiczales, U. British Columbia
Shigeru Chiba, Tokyo Tech	Martin Odersky, EPFL
Pierre Cointe, E. d. Mines de Nantes	Calton Pu, Georgia Tech
Charles Consel, INRIA, LaBRI (Chair)	Peter Thiemann, U. Freiburg
Dawson Engler, Stanford U.	Andrew Tolmach, Portland State U.
Siau Cheng Khoo, N. U. Singapore	

External Reviewers

Abhik Roychoudhury, N. U. Singapore
Gheorghe Stefanescu, N. U. Singapore
Razvan Voicu, N. U. Singapore

General Chair

Walid Taha, Rice U.

GCSE Advisory Committee

Don Batory, U. T. Austin	Ulrich Eisenecker, U. Kaiserslautern
Jan Bosch, U. Groningen	Bogdan Franczyk, U. Leipzig
Krzysztof Czarnecki, DaimlerChrysler	

SAIG Advisory Committee

Don Batory, U. T. Austin	Tim Sheard, OGI
Eugenio Moggi, U. Genova	Walid Taha, Rice U.
Greg Morrisett, Cornell U.	

Table of Contents

Invited Papers

Program Generation, Termination, and Binding-Time Analysis	1
<i>Neil D. Jones and Arne J. Glenstrup</i>	
Generative Programming for Embedded Systems	32
<i>Janos Sztipanovits, Gabor Karsai</i>	

Regular Papers

Self Reflection for Adaptive Programming	50
<i>Giuseppe Attardi, Antonio Cisternino</i>	
DataScript – A Specification and Scripting Language for Binary Data	66
<i>Godmar Back</i>	
Memoization in Type-Directed Partial Evaluation	78
<i>Vincent Balat, Olivier Danvy</i>	
A Protocol Stack Development Tool Using Generative Programming	93
<i>Michel Barbeau, Francis Bordeleau</i>	
Building Composable Aspect-Specific Languages with Logic Metaprogramming	110
<i>Johan Brichau, Kim Mens, Kris De Volder</i>	
Architectural Refactoring in Framework Evolution: A Case Study	128
<i>Greg Butler</i>	
Towards a Modular Program Derivation via Fusion and Tupling	140
<i>Wei-Ngan Chin, Zhenjiang Hu</i>	
Generative Programming for Embedded Software: An Industrial Experience Report	156
<i>Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, Ulrich Eisenecker</i>	
A Framework for the Detection and Resolution of Aspect Interactions	173
<i>Rémi Douence, Pascal Fradet, Mario Südholt</i>	
Aspect-Oriented Modeling: Bridging the Gap between Implementation and Design	189
<i>Tzilla Elrad, Omar Aldawud, Atef Bader</i>	

Macros That Compose: Systematic Macro Programming	202
<i>Oleg Kiselyov</i>	
Program Termination Analysis in Polynomial Time	218
<i>Chin Soon Lee</i>	
Generators for Synthesis of QoS Adaptation in Distributed Real-Time Embedded Systems	236
<i>Sandeep Neema, Ted Bapty, Jeff Gray, Aniruddha Gokhale</i>	
Optimizing Content Management System Pipelines (Separation and Merging of Concerns)	252
<i>Markus Noga, Florian Krüper</i>	
Component-Based Programming for Higher-Order Attribute Grammars . . .	268
<i>João Saraiva</i>	
Altering Java Semantics via Bytecode Manipulation	283
<i>Éric Tanter, Marc Ségura-Devillechaise, Jacques Noyé, José Piquer</i>	
Meta-programming with Concrete Object Syntax	299
<i>Eelco Visser</i>	
Managing Dynamic Changes in Multi-stage Program Generation Systems	316
<i>Zhenghao Wang, Richard R. Muntz</i>	
Author Index	335

Program Generation, Termination, and Binding-Time Analysis

Invited Paper

Neil D. Jones and Arne J. Glenstrup

DIKU, University of Copenhagen
`{neil,panic}@diku.dk`

Abstract. Recent research suggests that the goal of fully automatic and reliable program generation for a broad range of applications is coming nearer to feasibility. However, several interesting and challenging problems remain to be solved before it becomes a reality. Solving them is also *necessary*, if we hope ever to elevate software engineering from its current state (a highly-developed handiwork) into a successful branch of engineering, capable of solving a wide range of new problems by systematic, well-automated and well-founded methods.

We first discuss the relations between problem specifications and their solutions in program form, and then narrow the discussion to an important special case: program transformation. Although the goal of fully automatic program generation is still far from fully achieved, there has been some success in a special case: partial evaluation, also known as program specialization.

A key problem in all program generation is *termination* of the generation process. This paper describes recent progress towards automatically solving the termination problem, first for individual programs, and then for specializers and “generating extensions,” the program generators that most offline partial evaluators produce.

The paper ends with a list of challenging problems whose solution would bring the community closer to the goal of broad-spectrum, fully automatic and reliable program generation.

1 On Program Generation

Program generation is a rather old idea, dating back to the 1960s and seen in many forms since then. Instances include code templates, macro expansion, conditional assembly and the use of if-defs, report generators, partial evaluation/program specialization [15,41,51], domain-specific languages (at least, those compiled to an implementation language) [37,55], program transformation [8,11,25], and both practical and theoretical work aimed at generating programs from specifications [37,43].

Recent years have seen a rapid growth of interest in generative, automatic approaches to program management, updating, adaptation, transformation, and evolution, e.g., [43,52,70]. The motivations are well-known: persistence of the

“software crisis” and dreams of achieving industrial-style automation, automatic adaptation of programs to new contexts, and automatic optimization.

The overall goal is to increase efficiency of software production by making it possible to write fewer but more abstract or more heavily parameterized programs. Ideally, one would like to transform a problem specification automatically into a solution in program form. Benefits of such an approach would include increased reliability in software production: The software developer can debug/test/verify a small number of high-level, compact programs or specifications; and then generate from these as many machine-near, efficiently executable, problem-specific programs as needed, all guaranteed to be faithful to the specification from which they were derived.

1.1 What Are Programs Generated From?

On a small scale, programs can be generated from *problem parameters* (e.g., device characteristics determine the details of a device driver). Ideally, and on a larger scale, programs could be generated from *user-oriented problem descriptions or specifications*. Dreams of systematically transforming specifications into solutions were formulated in the 1970s by Dijkstra, Gries, Hoare, Manna, etc.

This approach has been tried out in real-world practice and some serious problems have been encountered. First, it is *very hard* to see whether a program actually solves the problem posed by a description or specification. Second, it seems impossible for specifications of manageable size to be complete enough to give the “whole picture” of what a program should do for realistic problem contexts. While there have been recent and impressive advances in bridging some gaps between specifications and programs by applying model-checking ideas to software [18], they are incomplete: the specifications used nearly always describe only certain misbehaviors that cannot be accepted, rather than total program correctness.

Executable specifications. It has proven to be quite difficult to build a program from an unexecutable, nonalgorithmic specification, e.g., expressed in first-order or temporal logic. For realistic problems it is often much more practical to use an *executable specification*. Such a specification is, in essence, also a program; but is written on a higher level of abstraction than in an implementation language, and usually has many “don’t-care” cases.

Consequence: Many examples of “generative software engineering” or “transforming specifications into programs” or “program generation” can be thought of as transformation from one high-level program (specification-oriented) to another lower-level one (execution-oriented). In other words, much of generative software engineering can be done by (variations on) *compilation* or *program transformation*.

This theme has been seen frequently, first in early compilers, then in use of program transformation frameworks for optimizations, then in partial evaluation, and more recently in multi-stage programming languages.

Partial evaluation for program generation. This paper concerns some improvements needed in order to use partial evaluation as a technology for automatically transforming an executable problem specification into an efficient stand-alone solution in program form. There have been noteworthy successes and an enormous literature in partial evaluation: see [62] and the PEPM series, as well as the Asia-PEPM and SAIG conferences. However, its use for large-scale program generation has been hindered by the need, given an executable problem specification, to do “hand tuning” to ensure *termination of the program generation process*: that the partial evaluator will yield an output program for every problem instance described by the current problem specification.

1.2 Fully Automatic Program Transformation: An Impossible Dream?

Significant speedups have been achieved by program transformation on a variety of interesting problems. Pioneers in the field include Bird, Boyle, Burstall, Darlington, Dijkstra, Gries, the Kestrel group, Meertens and Paige; more recent researchers include De Moor, Liu and Pettorossi. As a result of this work, some significant common principles for deriving efficient programs have become clear:

- Optimize by *changing the times* at which computations are performed (code motion, preprocessing, specialization, multi-staged programming languages [2,15,27,41,35,69,72,73]).
- Don’t solve *the same sub-problem* repeatedly. A successful practical example is the XSB logic programming system [59], based on *memoization*: Instead of recomputing results, store them for future retrieval when first computed.
- Avoid *multiple passes* over same data structure (tupling, deforestation [8,11,33,77]).
- Use an abstract, *high-level specification language*. SETL [9] is a good example, with small mathematics-like specifications (sets, tuples, finite mappings, fixpoints) that are well-suited to automatic transformation and optimization of algorithms concerning graphs and databases.

The pioneering results already obtained in program transformation are excellent academic work that include the systematic reconstruction of many state-of-the-art fundamental algorithms seen in textbooks. Such established principles are being used to develop algorithms in newer fields, e.g., computational geometry and biocomputation. On the other hand, these methods seem ill-suited to large-scale, heterogeneous computations: tasks that are broad rather than deep.

Summing up, program transformation is a promising field of research but its potential in practical applications is still far from being realized. Much of the earlier work has flavor of a “fine art,” practiced by highly gifted researchers on one small program at a time. Significant automation has not yet been achieved for the powerful methods capable of yielding superlinear speedups. The less ambitious techniques of partial evaluation have been more completely automated, but even there, termination remains a challenging problem.

This problem setting defines the focus of this paper. For simplicity, in the remainder of this article we use a simple first-order functional language.

1.3 Requirements for Success in Generative Software Engineering

The major bottlenecks in generative software engineering have to do with humans. For example, most people cannot and should not have to understand automatically generated programs—e.g., to debug them—because they did not write them themselves. A well-known analogy: the typical user does not and probably cannot read the output of parser generators such as Yacc and Lex, or the code produced by any commercial compiler.

If a user is to trust programs he/she did not write, a *firm semantic basis* is needed, to ensure that user intentions match the behavior of the generated programs. Both intentions and behavior must be clearly and precisely defined. How can this be ensured?

First, the source language or specification language must have a precisely understood semantics (formal or informal; but tighter than, say, C++), so the user can know exactly what was specified. Second, evidence (proof, testing, etc.) is needed that the output of the program generator always has the same semantics as specified by its input. Third, familiar software quality demands must be satisfied, both by the program generator itself and the programs that it generates.

Desirable properties of a program generator thus include:

1. *High automation level*. The process should
 - accept any well-formed input, i.e., not commit generation-time failures like “can’t take tail of the empty list”
 - issue sensible error messages, so the user is not forced to read output code when something goes wrong
2. The *code generation process* should
 - terminate for all inputs
 - be efficient enough for the usage context, e.g., the generation phase should be very fast for run-time code generation, but need not be for highly-optimizing compilation
3. The *generated program*
 - should be efficient enough for the usage context, e.g., fast generated code in a highly-optimizing compilation context
 - should be of predictable complexity, e.g., not slower than executing the source specification
 - should have an acceptable size (no code explosion)
 - may be required to terminate.

1.4 On the Meaning of “Automatic” in Program Transformation

The goals above are central to the field of automatic program transformation. Unfortunately, this term seems to mean quite different things to different people, so we now list some variations as points on an “automation” spectrum:

1. Hand work, e.g., program transformation done and proven correct on paper by gifted researchers. Examples: recursive function theory [56], work by McCarthy and by Bird.
2. Interactive tools for individual operations, e.g., call folding and unfolding. Examples: early theorem-proving systems, the Burstall-Darlington program transformation system [8].
3. Tools with automated strategies for applying transformation operations. Human interaction to check whether the transformation is “on target.” Examples: Chin, Khoo, Liu [11,50].
4. Hand-placed program annotations to guide a transformation tool. Examples: Chambers’ and Engeler’s DyC and Tick C [34,57]. After annotation, transformation is fully automatic, requiring no human interaction.
5. Tools that automatically recognize and avoid the possibility of nontermination during program transformation, e.g., homeomorphic embedding. Examples: Sørensen, Glück, Leuschel [47,65].
6. Computer-placed program annotations to guide transformation, e.g., binding-time annotation [6,14,15,31,32,41,35,74]. After annotation, transformation is fully automatic, requiring no human interaction.

In the following we use the term “automatic” mostly with the meaning of point 6. This paper’s goal is automatically to perform the annotations that will ensure termination of program specialization.

2 Partial Evaluation and Program Generation

Partial evaluation [15,20,41,35,51] is an example of automatic program transformation. While speedups are more limited than sometimes realizable by hand transformations based on deep problem or algorithmic knowledge, the technology is well-automated. It has already proven its utility in several different contexts:

- Scientific computing ([4,32], etc.);
- Extending functionality of existing languages by adding binding-time options ([34], etc.);
- Functional languages ([6,14,38,44,75], etc.);
- Logic programming languages ([25,26,48,51], etc.);
- Compiling or other transformation by specializing interpreters ([5,6,28,41,40,53], etc.);
- Optimization of operating systems, e.g., remote procedure calls, device drivers, etc. ([16,52,57], etc.);

2.1 Equational Definition of a Parser Generator

To get started, and to connect partial evaluation with program generation, we first exemplify our notation for program runs on an example. (Readers familiar with the field may wish to skip to Section 2.4.)

Parser generation is a familiar example of program generation. The construction of a parser from a grammar by a parser generator, and running the generated parser, can be described by the following two program runs:¹

```
parser    := [[parse-gen]] [grammar]
parsetree := [[parser]] [inputstring]
```

The combined effect of the two runs can be described by a nested expression:

```
parsetree := [[ [parse-gen] [grammar] ] ] [inputstring]
```

2.2 What Goals Does Partial Evaluation Achieve?

A partial evaluator [15,41,35] is a *program specializer*: Suppose one is given a subject program p , expecting two inputs, and the first of its input data, $in1$ (the “static input”). The effect of specialization is to construct a new program p_{in1} that, when given p ’s remaining input $in2$ (the “dynamic input”), yields the same result that p would have produced if given both inputs.

The process of partial evaluation is shown schematically in Figure 1.

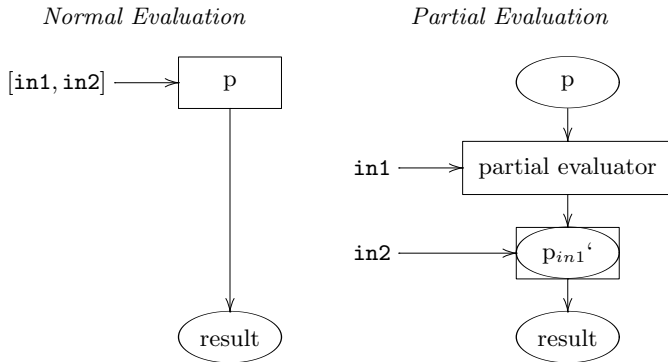


Fig. 1. Comparison of normal and partial evaluation. Boxes represent programs, ovals represent data objects. The *residual program* is p_{in1} .

Example 1: the standard toy example for partial evaluation is the program *power*, computing x^n with code:

```
f(n,x) = if n=0 then 1 else if odd(n) then x*f(n-1,x) else f(n/2,x)**2
```

¹ Notation: $[[p]] [in1, \dots, ink]$ is a partial value: the result yielded by running program p on input values $in1, \dots, ink$ if this computation terminates, else the *undefined* value \perp .

This example illustrates that partial evaluation in effect does an aggressive constant propagation across function calls. Specialization to static input $n = 13$ yields a residual program that runs several times faster than the general one above:

```
f_13(x) = x*((x*(x**2))**2)**2
```

Example 2: Ackermann’s function. If calls are not unfolded, then specialized functions are generated in the specialized program, as seen by this example of specialization to static input $m=2$:

<i>Source program</i>	<i>Specialized program (for $m = 2$)</i>
ack(m,n) = if m=0 then n+1 else	ack_2(n) = if n=0 then
if n=0 then	ack_1(1) else
ack(m-1,1) else	ack_1(ack_2(n-1))
ack(m-1,ack(m,n-1))	ack_1(n) = if n=0 then
	ack_0(1) else
	ack_0(ack_1(n-1))
	ack_0(n) = n+1

2.2.1 Equational definition of a partial evaluator. Correctness of p_{in1} as described by Figure 1 can be described equationally as follows:²

Definition 1. A partial evaluator *spec* is a program such that for all programs p and input data $in1, in2$:

$$\llbracket p \rrbracket [in1, in2] = \llbracket \llbracket spec \rrbracket [p, in1] \rrbracket [in2]$$

Writing p_{in1} for $\llbracket spec \rrbracket [p, in1]$, the equation can be restated without nested semantic parentheses $\llbracket \rrbracket$:

$$\llbracket p \rrbracket [in1, in2] = \llbracket p_{in1} \rrbracket [in2]$$

Program p_{in1} is often called the “residual program” of p with respect to $in1$, as it specifies remaining computations that were not done at specialization time.

2.2.2 Desirable properties of a partial evaluator.

Efficiency. This is the main goal of partial evaluation. Run $out := \llbracket p \rrbracket [in1, in2]$ is often slower than the run $out := \llbracket p_{in1} \rrbracket [in2]$, as seen in the two examples above. Said another way: It is slower to run the *general* program p on $[in1, in2]$ than it is to run the *specialized* residual program p_{in1} on $in2$.

The reason is that, while constructing p_{in1} , some of p ’s operations that depend only on $in1$ have been *precomputed*, and some function calls have been *unfolded* or “inlined.” These actions are never done while running $\llbracket p_{in1} \rrbracket [in2]$; but they will be performed *every time* $\llbracket p \rrbracket [in1, in2]$ is run. This is especially relevant if p is run often, with $in1$ changing less frequently than $in2$.

² An equation $\llbracket p \rrbracket [in1, \dots, ink] = \llbracket q \rrbracket [in1, \dots, ink]$ expresses equality of partial values: if either side terminates, then the other side does too, and they yield the same value.

Correctness. First and foremost, a partial evaluator should be correct. Today's state of the art is that most partial evaluators give correct residual programs when they terminate, but sometimes loop infinitely or sometimes give code explosion. A residual program, once constructed, will in general terminate at least as often as the program from which it was derived.

Completeness. Maximal speedup will be obtained if *every computation* depending only on `in1` is performed. For instance the specialized Ackermann program above, while faster than the original, could be improved yet more by replacing the calls `ack_1(1)` and `ack_0(1)`, respectively, by constants 3 and 2.

Termination. Ideally:

- The specializer should terminate for all inputs `[p, in1]`.
- The generated program `pin1` should terminate on `in2` just in case `p` terminates on `[in1, in2]`.

These termination properties are easier to state than to achieve, as there is an intrinsic conflict between demands for completeness and termination of the specializer. Nonetheless, significant progress has been made in the past few years [21, 22, 29, 31, 64]. A key is carefully to select which parts of the program should be computed at specialization time. (In the usual jargon: which parts are considered as “static.”)

2.2.3 A breakthrough: the generation of program generators. For the first time in the mid 1980s, a breakthrough was achieved in practice that had been foreseen by Japanese and Russian researchers early in the 1970s: the automatic generation of compilers and other program generators by self-application of a specializer. The following definitions [23, 24] are called the *Futamura projections*:

Generic Futamura projections

1. `pin1` := `[[spec]] [p, in1]`
2. `p-gen` := `[[spec]] [spec, p]`
3. `cogen` := `[[spec]] [spec, spec]`

Consequences:

- A. `[[p]] [in1, in2] = [[pin1]] [in2]`
- B. `pin1 = [[p-gen]] [in1]`
- C. `p-gen = [[cogen]] [p]`

Program `p-gen` is called `p`'s “generating extension.” By B it is a *generator of specialized versions of p*, that transforms static parameter `in1` into specialized program `pin1`. Further, by C and B the program called `cogen` behaves as a *generator of program generators*.

Consequence A is immediate from Definition 1. Proof of Consequence B is by simple algebra:

$$\begin{aligned}
\llbracket \text{p-gen} \rrbracket [\text{in1}] &= \llbracket \llbracket \text{spec} \rrbracket [\text{spec}, \text{p}] \rrbracket [\text{in1}] && \text{By definition of p-gen} \\
&= \llbracket \text{spec} \rrbracket [\text{p}, \text{in1}] && \text{By Definition 1} \\
&= \text{p}_{\text{in1}} && \text{By definition of p}_{\text{in1}}
\end{aligned}$$

and consequence C is proven by similar algebraic reasoning, left to the reader.

Efficiency gained by self-application of a partial evaluator. Compare the two ways to specialize program p to static input in1 :

- I. $\text{p}_{\text{in1}} := \llbracket \text{spec} \rrbracket [\text{p}, \text{in1}]$
- II. $\text{p}_{\text{in1}} := \llbracket \text{p-gen} \rrbracket [\text{in1}]$

Way I is in practice usually several times slower than Way II. This is for exactly the same reason that p_{in1} is faster than p : Way I runs spec , which is a *general* program that is able to specialize *any* program p to any in1 . On the other hand, Way II runs p-gen : a *specialized* program, only able to produce specialized versions of this particular p .

By exactly the same reasoning, computing $\text{p-gen} := \llbracket \text{spec} \rrbracket [\text{spec}, p]$ is often several times slower than using the compiler generator $\text{p-gen} := \llbracket \text{cogen} \rrbracket [p]$.

Writing cogen instead of spec. In a sense, any program generator is a “generating extension,” but without an explicitly given general source program p . To see this, consider the two runs of the parser generator example:

```

parser      :=  $\llbracket \text{parse-gen} \rrbracket [\text{grammar}]$ 
parsetree :=  $\llbracket \text{parser} \rrbracket [\text{inputstring}]$ 

```

Here in principle a universal parser (e.g., Earley’s parser) satisfying:

$$\text{parsetree} = \llbracket \text{universal-parser} \rrbracket [\text{grammar}, \text{inputstring}]$$

could have been used to obtain parse-gen as a generating extension:

$$\text{parse-gen} := \llbracket \text{cogen} \rrbracket [\text{universal-parser}]$$

Further, modern specializers such as C-mix [28], Tempo [16] and PGG [75] are *cogen* programs written directly, rather than built using self-application of *spec* as described above and in book [41]. (Descriptions of the idea can be found in [5,36,74].) Although not built by self-application, the net effect is the same: programs can be specialized; and a program may be converted into a generating extension with respect to its static parameter values.

2.2.4 Fromt-end compilation: an important case of the Futamura projections. Suppose that we now have *two* languages: the specializer’s input-output language L , and another language S that is to be implemented. Let $\llbracket \cdot \rrbracket^S$ be the “semantic function” that assigns meanings to S -programs.

Definition 2. Program *interp* is an interpreter for language S written in language L if for any S -program *source* and input *in*,

$$\llbracket \text{source} \rrbracket^S[\text{in}] = \llbracket \text{interp} \rrbracket[\text{source}, \text{in}]$$

(See Figure 2 in Section 2.2.5 for an example interpreter.) We now apply the Futamura projections with some renaming: Replace program `p` by `interp`, static input `in1` by *S*-program `source`, dynamic input `in2` by `in`, and `p-gen` by `compiler`.

Compilation by the Futamura projections

1. `target` := $\llbracket \text{spec} \rrbracket[\text{interp}, \text{source}]$
2. `compiler` := $\llbracket \text{spec} \rrbracket[\text{spec}, \text{interp}]$
3. `cogen` := $\llbracket \text{spec} \rrbracket[\text{spec}, \text{spec}]$

After this renaming, the “Consequences” of Section 2.2.3 become:

- A. $\llbracket \text{interp} \rrbracket[\text{source}, \text{in}] = \llbracket \text{target} \rrbracket[\text{in}]$
- B. `target` = $\llbracket \text{compiler} \rrbracket[\text{source}]$
- C. `compiler` = $\llbracket \text{cogen} \rrbracket[\text{interp}]$

Program `target` deserves its name, since it is an *L*-program with the same input-output behavior as *S*-program `source`:

$$\llbracket \text{source} \rrbracket^S[\text{in}] = \llbracket \text{interp} \rrbracket[\text{source}, \text{in}] = \llbracket \text{target} \rrbracket[\text{in}]$$

Further, by the other consequences program `compiler` = `interp-gen` transforms `source` into `target` and so really is a compiler from *S* to *L*; and `cogen` is a *compiler generator* that transforms interpreters into compilers.

The compilation application of specialization imposes some natural demands on the quality of the specializer:

- The compiler `interp-gen` *must terminate* for all source program inputs.
- The target programs should be *free of all source code* from program `source`.

2.2.5 Example of compiling by specializing an interpreter. Consider the simple interpreter `interp` given by the pseudocode in Figure 2.³ An interpreted source program (to compute the factorial function *n!*) might be:

$$\text{pg} = ((f \ (n \ x) \ (if \ =(n,0) \ then \ 1 \ else \ *(x, \ call \ f(-(n,1), \ x))))))$$

Let `target` = $\llbracket \text{spec} \rrbracket[\text{interp}, \text{pg}]$ be the result of specializing `interp` to static source program `pg`. Any reasonable compiler should “specialize away” all source code. In particular, all computations involving syntactic parameters `pg`, `e` and `ns` should be done at specialization time (considered as “static.”)

For instance, when given the source program above, we might obtain a specialized program like this:

³ The interpreter is written as a first-order functional program, using Lisp “S-expressions” as data values. The current binding of names to values is represented by two lists: `ns` for names, and parallel list `vs` for their values. For instance computation of $\llbracket \text{interp} \rrbracket[\text{pg}, [2,3]]$ would use initial environment `ns` = `(n x)` and `vs` = `(2 3)`.

Operations are `car`, `cdr` corresponding to ML’s `hd`, `tl`, with abbreviations such as `cadr(x)` for `car(cdr(x))`.

```

run(prog,d) = [1] eval(lkbody(main,prog),lkparm(main,prog), d, prog)
eval(e,ns,vs,pg) = case e of
  c                : valueof(c)
  x                : lkvar(x, ns, vs)
  basefn(e1,...,en) : apply(basefn,[2] eval(e1,ns,vs,pg), ...,
                                [3] eval(en,ns,vs,pg))

  let x = e1 in e2   : [4] eval(e2,
                                cons(x, ns),
                                cons([5] eval(e1,ns,vs,pg),vs),
                                pg)

  if e1 then e2 else e3: if [6] eval(e1, ns, vs, pg)
                                then [7] eval(e2, ns, vs, pg)
                                else [8] eval(e3, ns, vs, pg)

  call f(e1,...,en)   : [9] eval(lkbody(f,pg),
                                lkparm(f,pg),
                                list([10] eval(e1,ns,vs,pg), ...,
                                    [11] eval(en,ns,vs,pg)),
                                pg)

lkbody(f,pg) = if caar(pg)=f then caddar(pg) else lkbody(f,cdr(pg))
lkparm(f,pg) = if caar(pg)=f then cadar(pg) else lkparm(f,cdr(pg))
lkvar(x,ns,vs) = if car(ns)=x then car(vs) else lkvar(x,cdr(ns),cdr(vs))

```

Fig. 2. `interp`, an interpreter for a small functional language. Parameter `e` is an expression to be evaluated, `ns` is a list of parameter names, `vs` is a parallel list of values, and `pg` is a program (for an example, see Section 2.2.5.) The `lkparm` and `lkbody` functions find a function’s parameter list and its body. `lkvar` looks up the name of a parameter in `ns`, and returns the corresponding element of `vs`.

```

eval_f(vs) =
  if apply('=', car(vs), 0) then 1 else
    apply('*', cadr(vs),
           eval_f(list(apply('-', car(vs), 1), cadr(vs))))

```

Clearly, running `eval_f(d)` is several times faster than running `run(pg,d)`.⁴

⁴ Still better code can be generated! Since list `vs` always has length 2, it could be split into two components `v1` and `v2`. Further generic “`apply(op, ...)`” can be replaced by specialized “`op(...)`”. These give a program essentially identical to the interpreter input:

```

eval_f(v1,v2) = if =(v1,0) then 1 else *(v2,eval_f(-(v1,1), v2))

```

This is as good as can reasonably be expected, cf. the discussion of “optimal” specialization in [41,72].

2.3 How Does Partial Evaluation Achieve Its Goals?

Partial evaluation is analogous to memoization, but not the same. Instead of saving a complete *value* for later retrieval, a partial evaluator generates *specialized code* (possibly containing loops) that will be executed at a later stage in time.

The specialization process is easy to understand provided the residual program contains no loops, e.g., the “power” function seen earlier can be specialized by computing values statically when possible, and generating residual code for all remaining operations. But how to proceed if *residual program loops* are needed, e.g., as in the “Ackermann” example, or when specializing an interpreter to compile a source program that contains repetitive constructs? One answer is program-point specialization.

2.3.1 Program-point specialization. Most if not all partial evaluators employ some form of the principle: A control point in the specializer’s output program corresponds to a pair (*pp*, *vs*): a source program control point *pp*, plus some knowledge (“static data”) *vs* about of the source program’s runtime state. An example is shown for Ackermann’s function in Figure 3 (program points are labeled by boxed numbers.) Residual functions *ack_2*, *ack_1*, *ack_0* correspond to program point 0 (entry to function *ack*) and known values *m* = 2, 1, 0, respectively.

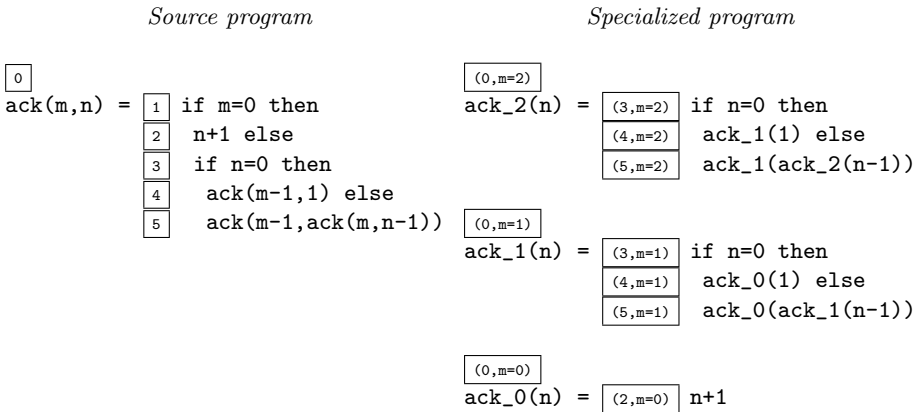


Fig. 3. Partial evaluation of Ackermann’s function for static *m*=2

2.3.2 Sketch of a generic expression reduction algorithm. To make the issues clearer and more concrete, let *exp* be an expression and let *vs* be

the known information about program p 's run-time state.⁵ The “reduced” or residual expression $\text{exp}^{\text{resid}}$ can be computed roughly as follows:

Expression reduction algorithm $\text{Reduce}(\text{exp}, vs)$:

1. Reduce any subexpressions e_1, \dots, e_n of exp to $e_1^{\text{resid}}, \dots, e_n^{\text{resid}}$.
2. If $\text{exp} = \text{“basefn}(e_1, \dots, e_n)\text{”}$ and some e_i^{resid} is not fully known, then
 $\text{exp}^{\text{resid}} =$ the **residual expression** “basefn($e_1^{\text{resid}}, \dots, e_n^{\text{resid}}$)”, else
 $\text{exp}^{\text{resid}} =$ the **value** $\llbracket \text{basefn} \rrbracket(e_1^{\text{resid}}, \dots, e_n^{\text{resid}})$.
3. If $\text{exp} = \text{“if } e_0 \text{ then } e_1 \text{ else } e_2\text{”}$ and e_0^{resid} is not fully known, then
 $\text{exp}^{\text{resid}} =$ the residual expression “if e_0^{resid} then e_1^{resid} else e_2^{resid} ”, else
 $\text{exp}^{\text{resid}} = e_1^{\text{resid}}$ in case $e_0^{\text{resid}} = \text{“True”}$, else $\text{exp}^{\text{resid}} = e_2^{\text{resid}}$.
4. A function call $\text{exp} = \text{“f}(e_1, \dots, e_n)\text{”}$ can either be
residualized: $\text{exp}^{\text{resid}} = \text{f}(e_{i_1}, \dots, e_{i_m})$ with some known e_{i_j} omitted⁶,
or **unfolded**: $\text{exp}^{\text{resid}} = \text{Reduce}(\text{definition of f}, vs')$
where vs' is the static knowledge about $e_1^{\text{resid}}, \dots, e_n^{\text{resid}}$.

This algorithm computes expression values depending only on static data, reduces static “if” constructs to their “then” or “else” branches, and either unfolds or residualizes function calls. The latter enables the creation of loops (recursion) in the specialized program. Any expression computation or “if”-branching that depends on dynamic data is postponed by generating code to do the computation in the specialized program.

Naturally, an expression may be reached several times by the specializer (it may be in a recursive function). If this program point and the current values of the source program, (pp, vs) , have been seen before, one of two things can happen: If pp has been identified as a specialization point, the specializer generates a call in the specialized program to the code generated previously for (pp, vs) . Otherwise, the call is unfolded and specialization continues as usual.

This sketch leaves several choices unspecified, in particular:

- Whether or not to unfold a function call
- Which among the known function arguments are to be removed (if any)

⁵ This knowledge can take various forms, and is often a term with free variables. For simplicity we will assume in the following that at specialization time every parameter is either fully static (totally known) or fully dynamic (totally unknown), i.e., that there is no partially known data.

Some partial evaluators are more liberal, for example, allowing lists of statically known length with dynamic elements. This is especially important in partial evaluation of Prolog programs or for supercompilation [33,48,65,76].

⁶ If a function call is residualized, a definition of a specialized function must also be constructed by applying *Reduce* to the function's definition.

2.3.3 Causes of nontermination during specialization. First, we show an example of nontermination during specialization. A partial evaluator should not indiscriminately compute computable static values and unfold function calls, as this can cause specialization to loop infinitely even when normal evaluation would not do so. For example, the following program computing $2x$ in “base 1” terminates for any x :

```
double(x)      = dblplus(x,0)
dblplus(u,v)   = if u <= 0 then v else dblplus(u-1,v+2)
```

Now suppose x is a dynamic (unknown) program input. A naïve specializer might “reason” that parameter v first has value 0, a constant known at specialization time. Further, if at any stage the specializer knows one value v of v , then it can compute its next value, $v + 2$. Alas, repeatedly unfolding calls to `dblplus` assigns to v the values 0, 2, 4, 6, ... causing an infinite loop at specialization time.

The way to avoid this problem is not to unfold the second `dblplus` call. The effect is to *generalize* v , i.e., to make it dynamic.

Definition 3. *A parameter x will be called potentially static if dynamic program inputs are not needed to compute its value.*

Three effects are the main causes of nontermination in partial evaluation:

1. A loop controlled by a dynamic conditional can involve a potentially static parameter that takes on infinitely many values, generating an infinite set of specialized program control points $\{(\mathbf{pp}, vs_1), (\mathbf{pp}, vs_2), \dots\}$. Parameter v of function `dblplus` illustrates this behavior.
2. A too liberal policy for unfolding of function calls can cause an attempt to generate an infinitely large specialized program.
3. As both branches of dynamic conditionals are (partially) evaluated, partial evaluation is *over-strict* in the sense that it may evaluate *more* expressions than normal evaluation would, and thus risk nontermination not present in the source program being specialized.

If there is danger of specializing a function with respect to infinitely many static values, then some of the parameters in question should be generalized, i.e., made dynamic. Instead of specializing the function with respect to *all* potentially static values, some will then become parameters in the specialized program, to be computed at run-time.

We will show how this can be done automatically, before specialization begins (of automation degree 6 in the list of Section 1.4.). A companion goal is to unfold function calls “just enough but not too much.”

2.3.4 Online and offline specialization. Partial evaluators fall in two categories, *on-line* and *off-line*, according to the time at which it is decided whether parameters should be generalized or calls should be unfolded (cf. Step 4 of the *Reduce* algorithm).

An online specializer generalizes during specialization. Usually this involves, at each call to a function f , comparing the newly-computed static arguments with the values of *all* parameters seen in earlier calls to f , in order to detect potentially infinite value sequences [4,26,47,48,65,76].

An offline specializer works in two stages. Stage 1, called a *binding-time analysis* (BTA for short) yields an *annotated program* in which:

- each parameter is marked as either “static” or “dynamic,”
- each expression is marked as “reduce” or “specialize,” and
- each call is marked as “unfold” or “residualize.”

Annotation is done before the static program input is available, and only requires knowing *which* inputs will be known at Stage 2. Stage 2, given the annotated program and the values of static inputs, only needs to obey the annotations; argument comparisons are not needed [6,12,14,16,21,28,32,41,35,53].

Both kinds of specialization have their merits. Online specialization sometimes does more reduction: For example, in the specialization of Ackermann’s function in Figure 3, when the call at site $\boxed{4}$ is unfolded, an online specializer will realize that the value of n is known and compute the function call. In contrast, the binding-time analysis of offline specialization must classify n as dynamic at call site $\boxed{4}$, since its value is not known at call site $\boxed{5}$.⁷

Although an online specializer can sometimes exploit static data better, this extra precision comes at a cost. A major problem is how to determine online *when to stop* unfolding function calls. Comparison of *vs* with previously seen values can result in an extremely slow specialization phase, and the specialized program may be inefficient if specialization is stopped too soon.

An offline specializer is often faster since it needs to take no decision-making online: it only has to obey the annotations. Further, *full self-application*, as in the Futamura projections, has to date only been achieved by using offline methods.

2.4 Multi-level Programming Languages

Many informal algorithm optimizations work by changing the times at which computations are done, e.g., by moving computations out of loops; by caching values in memory or in files for future reference; or by generating code implicitly containing partial results of earlier computations. Such a change to an algorithm is sometimes called a *staging transformation*, or a *binding-time shift*. The field of *domain-specific languages* [37,43,55] employs similar strategies to improve efficiency. Partial evaluation automates this widely-used principle to gain speedup.

A BTA-annotated program can be thought of as a program in a *two-level language* in which statically annotated parts are executed, while the syntactically similar dynamic parts are in effect templates for generation of code to

⁷ Some partial evaluators allow more static computation than in the present paper by using *polyvariant BTA*, in which the binding-time analysis can generate a finite set of different combinations of static and dynamic parameters [12,14].

be executed at a later time. This line of thinking has been developed much further, e.g., [34] for C and, for typed functional languages, in early work on Meta-ML by Sheard and others [54,63,73], and subsequent work by Taha and other researchers papers [68,69,72]. Goals of the work include efficiency improvement by staging computations to understanding how multi-level languages may be designed and implemented; resolution of tricky semantic issues, in particular questions of renaming; and formal proofs to guarantee that multi-level type safety can be achieved. A recent paper applies these ideas to ensure type-safe use of macros [27].

Termination remains, however, a problem to be solved one case at a time by the programmer. Conceivably the ideas of this paper could contribute to a future strongly-terminating multi-level language.

Remark on language evolution: Figure 1 assumes that p_{in1} is an explicit, printable program. A program in a multi-level language such as Meta-ML can construct program bits as data values and then run them, but it does not produce stand-alone, separately executable programs.

Meta-ML is an interesting example of internalization of a concept *about* programming languages *into* a programming language construct. Analogous developments have happened before: continuations were developed to explain semantics of the **goto** and other control structures, but were quickly incorporated into new functional languages. Multi-level languages perhaps derive similarly from internalizing the fact that one language's semantics can be defined within another language, a concept also seen in Smith, Wand, Danvy and Asai's "reflective tower" languages Brown, Blonde and Black.

2.5 Challenging Problems

2.5.1 On the state of the art in partial evaluation. Partial evaluation has been most successful on simple "pure" languages such as Scheme [6,7,14,45] and Prolog [60], and there has been some success on C: the C-mix [28] and Tempo systems [16]. Further, Schultz has succeeded in doing Java specialization by translating to C, using the Tempo system, and the translating back [61].

There have been a number of practically useful applications of partial evaluation, many exploiting its ability to build program generators [32,40,52,45,67].

Partial evaluation also has some weaknesses. One is that speedups are at most linear in the subject program's runtime (although the size of the constant coefficient can be a function of the static input data.) Further, its use can be delicate: Obtaining good speedups requires a close knowledge of the program to be specialized, and some "binding-time improvements" may be needed to get expected speedups. (These are hand work, of automation degree 4 in the list of Section 1.4.) Another weakness is that the results of specialization can be hard to predict: While speedup is common, slowdown is also possible, as is the possibility of code explosion or infinite loops at specialization time.

Little success has been achieved to date on partial evaluation of the language C++, largely due to its complex, unclear semantics; or on Java, C#, and other languages with objects, polymorphism, modules and concurrency.

One reason for the limited success in these languages is that partial evaluation requires precomputing as much as is possible, based on partial knowledge of a program's input data. To do this requires anticipating the space of all possible run-time states. Such analysis can be done by abstract interpretation [19,42] for simple functional or logic programming languages, but becomes much more difficult for more complex or more dynamic program semantics, as both factors hinder the specialization-time prediction of run-time actions.

Languages such as C++, Java and C# seem at the moment to be too complex to be sufficiently precisely analyzed, and to allow reliable assurances that program semantics is preserved under specialization.

2.5.2 How to make specialization terminate? Reliable termination properties are essential for highly automatic program manipulation – and are not perfect in existing partial evaluation systems. In fact, few papers have appeared on the subject, exceptions being the promising but unimplemented [64], and works by Das and Glenstrup [21,22,29,31]. Although the generation of program generators is a significant accomplishment with promise for wider applications, program generator generation places even higher demands on predictability in the behavior of specialized output programs, and on the specialization process itself.

Termination of specialization can always be achieved; an extreme way is to make everything dynamic. This is useless, though, because no computations at all occur at specialization time and the specialized program is only a copy of the source program.

Our goal is thus to specialize in a way that maximizes the number of static computations, but nonetheless to guarantee termination. As a “stepping stone” in order to achieve the *two-level termination* required for partial evaluation, we first investigate a novel automatic approach to ordinary termination analysis.

3 Program Termination by “Size-Change Analysis”

Our ultimate goal is to ensure, given program p , that program specialization: $p_{in1} = \llbracket spec \rrbracket [p, in1]$ will terminate for all inputs $in1$. Before explaining our solution to this subtle problem, we describe an automated *solution to a simpler problem*: how to decide whether a (one-stage) program terminates on all inputs.

The termination problem is interesting in itself, of considerable practical interest, and has been studied by several communities: logic programming [49], term rewriting [3], functional programming [1,46,78] and partial evaluation [21, 22,29,31,64].

A guarantee of termination is hard to achieve in practice, and undecidable in general (it is the notorious uniform halting problem). Still, dependable positive answers are essential in practice to ensure system liveness properties: that a

system will never “hang” in an infinite loop, but continue to offer necessary services and make progress towards its goals. To this end, we have recently had some success in using *size-change analysis* to detect termination [46].

3.1 The Size-Change Termination Principle

Size-change analysis is based only on local information about parameter values. The starting point is a “size-change graph,” G_c , for each call $c : \mathbf{f} \rightarrow \mathbf{g}$ from function \mathbf{f} to function \mathbf{g} in the program. Graph G_c describes parameter value changes when call c is made (only equalities and decreases). The graphs are derivable from program syntax, using elementary properties of base functions (plus a size analysis for nested function calls, see [10,39,41]).

In the following, we shall consider programs written in a first-order functional language with well-founded data,⁸ and we say that a call sequence $cs = c_1 c_2 \dots$ is *valid* for a program if it follows the program’s control flow. The key to our termination analyses is the following principle:

A program terminates on all inputs if *every valid infinite call sequence* would, if executed, cause an infinite decrease in some parameter values.

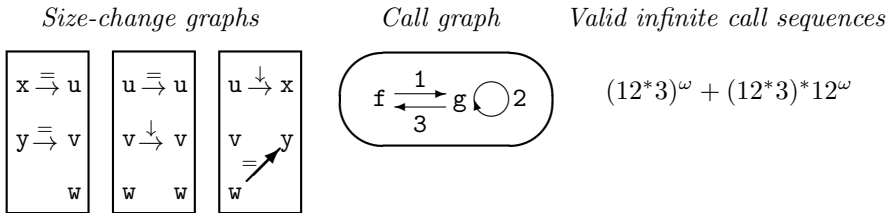
Example of size-change termination analysis. Consider a program on natural numbers that uses mutual recursion to compute $2^x \cdot y$.

```

f(x,y)  = if x = 0 then y else [1] g(x,y,0)
g(u,v,w) = if v > 0 then [2] g(u,v-1,w+2)
              else [3] f(u-1,w)

```

Following are the three size-change graphs for this program, along with its call graph and a description of its valid infinite call sequences. There is one size-change graph G_c for each call c , describing both the data flow and the parameter size changes that occur if that call is taken. To keep the analysis simple, we approximate the size changes by either ‘=’ (the change does not increase the value) or \downarrow (the change decreases the value). The valid infinite call sequences can be seen from the call graph, and are indicated by ω -regular expressions⁹.



$G_1 : \mathbf{f} \rightarrow \mathbf{g}$ $G_2 : \mathbf{g} \rightarrow \mathbf{g}$ $G_3 : \mathbf{g} \rightarrow \mathbf{f}$

⁸ I.e., no infinitely decreasing value sequences are possible. With the aid of other analyses this assumption may be relaxed so as to cover, for example, the integers.

⁹ * indicates any finite number of repetitions, and ω indicates an infinite number of repetitions.

Informal argument for termination: Examination of the size-change graphs reveals that a call sequence ending in 12^ω causes parameter v to decrease infinitely, but is impossible by the assumption that data values are well-founded. Similarly, a call sequence in $(12^*3)^\omega$ causes parameters x and u to decrease infinitely. Consequently *no valid infinite call sequence is possible*, so the program terminates.

3.2 A Termination Algorithm

It turns out that the set of valid infinite call sequences that cause infinite descent in some value is finitely describable¹⁰. Perhaps surprisingly, the infinite descent property is *decidable*, e.g., by automata-theoretic algorithms. We sketch an algorithm operating directly on the size-change graphs without the passage to automata.

Two size-change graphs $G : f \rightarrow g$ and $G' : g \rightarrow h$ may be composed in an obvious way to yield a size-change graph $G; G' : f \rightarrow h$. Their total effect can be expressed in a single size-change graph, like $G_{13} = G_1; G_3$, shown below. These compositions are used in the central theorem of [46]:

Theorem 1. *Let the closure \mathcal{S} of the size-change graphs for program p be the smallest graph set containing the given size-change graphs, such that $G; G' \in \mathcal{S}$ whenever \mathcal{S} contains both $G : f \rightarrow g$ and $G' : g \rightarrow h$. Then p is size-change terminating if and only if every idempotent $G \in \mathcal{S}$ (i.e., every graph satisfying $G = G; G$) has an in situ descent $z \downarrow \rightarrow z$.*

The closure set for the example program is:

$$\mathcal{S} = \{G_1, G_2, G_3, G_{12}, G_{13}, G_{131}, G_{23}, G_{231}, G_{31}, G_{312}\}$$

This theorem leads to a straightforward algorithm. The idempotent graphs in \mathcal{S} are G_2 (above) as well as G_{13} and G_{231} (below). Each of them has an *in situ* decreasing parameter, so no infinite computations are possible.

$$G_{13} = \begin{array}{|c|} \hline \begin{array}{c} x \downarrow \rightarrow x \\ y \quad y \end{array} \\ \hline \end{array} \qquad G_{231} = \begin{array}{|c|} \hline \begin{array}{c} u \downarrow \rightarrow u \\ v \quad v \\ w \quad w \end{array} \\ \hline \end{array}$$

3.3 Assessment

Compared to other results in the literature, termination analysis based on the size-change principle is surprisingly simple and general: lexicographical orders, indirect function calls and permuted arguments (descent that is not *in-situ*) are all handled automatically and without special treatment, with no need for manually supplied argument orders, or theorem-proving methods not certain to terminate at analysis time. We conjecture that programs computing all and only Péter’s “multiple recursive” functions [56] can be proven terminating by this method. Thus, a great many computations can be treated.

¹⁰ It is an ω -regular set, representable by a Büchi automaton.

Converging insights: This termination analysis technique has a history of independent rediscovery. We first discovered the principle while trying to communicate our earlier, rather complex, binding-time analysis algorithms to ensure termination of specialization [29,31]. The size-change termination principle arose while trying to explain our methods simply, by starting with one-stage computations as an easier special case [46]. To our surprise, the necessary reformulations led to BTA algorithms that were stronger as well as easier to explain.

Independently, Lindenstrauss and Sagiv had devised some rather more complex graphs to analyze termination of logic programs [49], based on the same mathematical properties as our method. A third discovery: the Sistla-Vardi-Wolper algorithm for determinization of Büchi automata [58] resembles our closure construction. This is no coincidence, as the infinite descent condition is quite close to the condition for acceptance of an infinite string by a Büchi automaton.

Termination analysis in practice: Given the importance of the subject, it seems that surprisingly few termination analyses have been implemented. In logic programming, the Termilog analyzer [49] can be run from a web page; and the Mercury termination analysis [66] has been applied to all the programs comprising the Mercury compiler. In functional programming, analyses [1] and [46] have both been implemented as part of the AGDA proof assistant [17]. Further, [78] uses dependent types for higher-order programs, not for termination analysis but to verify run-time bounds specified by the programmer with the aid of types. A recent paper on termination analysis of imperative programs is [13].

4 Guaranteeing Termination of Program Generation

We now show how to extend the size-change principle to an analysis to guarantee termination of specialization in partial evaluation. More details can be found in [30]. The results described here are significantly better than those of Chapter 14 in [41] and [29,31]. Although the line of reasoning is similar, the following results are stronger and more precise, and a prototype implementation exists.

4.1 Online Specialization with Guaranteed Termination

Most online specializers record the values of the function parameters seen during specialization, so that current static values *vs* can be compared with earlier ones in order to detect potentially infinite value sequences. A rather simple strategy (residualize whenever any function argument value increases) is sufficient to give good results on the Ackermann example seen earlier.

More liberal conditions that guarantee termination of specialization will yield more efficient residual programs. The most liberal conditions known to the authors employ variants of the Kruskal tree condition called “homeomorphic embedding” [65]. This test, carried out during specialization, seems expensive but strong. It is evaluated on a variety of test examples in [47].

4.2 Online Specialization with Call-Free Residual Programs

The termination-detecting techniques of Section 3 can easily be extended to find a sufficient precondition for online specialization termination, as long as residual programs have no calls. The idea is to modify the size-change termination principle as follows (recall Section 2.3.3, Definition 3):

Suppose every valid infinite call sequence causes an *infinite descent in some potentially static parameter*. Then online program specialization will terminate on any static input, to yield a call-free residual program.

A program passing this test will have no infinite specialization-time call sequences. This means that an online specializer may proceed blindly, computing every potentially static value and unfolding all function calls.

This condition can be tested by a slight extension of the closure algorithm of Section 3:

- Before specialization, identify the set PS of potentially static parameters.
- Construct the closure \mathcal{S} as in Section 3.
- Decide this question: Does every idempotent graph $G \in \mathcal{S}$ have an *in situ* decreasing parameter in PS ?

Strengths:

1. No comparisons or homeomorphic embedding tests are required, so specialization is quite fast.
2. The condition succeeds on many programs, e.g.,
 - the “power” example of Section 2.2 specialized to static input \mathbf{n} , or
 - Ackermann’s function specialized to static \mathbf{m} and \mathbf{n} .
3. If a size-change terminating program has no dynamic inputs, the test above will succeed and residual programs will have the form “ $\mathbf{f}(_) = \mathbf{constant}$ ”.
4. Further, any program passing this test is free of “static loops,” a notorious cause of nonterminating partial evaluation.

Weakness: The methods fails, however, for the Ackermann example with static \mathbf{m} and dynamic \mathbf{n} , or for interpreter specialization with static program argument \mathbf{pg} . The problem is that specialized versions of such programs must contain loops.

4.3 Offline Specialization of an Interpreter

As mentioned in Section 2.2.4, the property that source syntax is “specialized away” is vital for compiling efficient target programs, and when performing self-application of the specializer to generate compilers. Our main goal is to achieve a high degree of specialization (i.e., a fast specialized program) and at the same time a *guarantee* that the specialization phase terminates.

Consider specializing the simple interpreter **interp** of Figure 2 (Section 2.2.4) to static source program \mathbf{pg} and dynamic program input \mathbf{d} . One would hope and

expect all source code to be specialized away—in particular, syntactic parameters `pg`, `e` and `ns` should be classified as “static” and so not appear in target programs.

In Figure 2 parameter `vs` is dependent on dynamic program input `d`, and so must appear in the specialized programs. Parameter `pg` always equals static input `prog` (it is only copied in all calls), so any specializer should make it static.

Parameter `e` decreases at every call site except [9], where it is *reset* (always to a subterm of `pg`). After initialization or this reset, parameter `ns` is only copied, except at call site [4], where it is *increased*.

A typical *online* specializer might see at call site [9] that `eval`’s first argument `e` has a value larger than any previously encountered, and thus would consider it dynamic. By the same reasoning, argument `ns` would be considered dynamic at call site [4] due to the evident risk of unboundedness. Alas, such classifications will yield unacceptably poor target programs.

However the binding-time analysis of an *offline* partial evaluator considers the interpreter program as a whole, and can detect that `eval`’s first argument can only range over subterms of the function bodies in the interpreted program `pg`. This means that parameter `e` can be considered as “static” without risking nontermination, and can thus be specialized away. Further, parameter `ns` will always be a part of the interpreter’s program input `pg`. Offline specialization can ensure that *no source code from pg appears in any target program* produced by specializing this interpreter.

Conclusion: control of the specializer’s termination is easier offline than “on the fly.” This effect is not easily achieved by online methods.

Parameters that are reset to bounded values. We now describe a more subtle automatic “boundedness” analysis, that reveals that the set of values `ns` ranges over during specialization is also finite¹¹. First, some terminology.

4.4 Bounded Static Variation

Definition 4. Program `p` is quasiterminating iff for any input vector \overline{in} , the following is a finite set:

$$Reach(\overline{in}) = \{(\mathbf{f}, \overline{v}) \mid \text{Computation of } \llbracket p \rrbracket[\overline{in}] \text{ calls } \mathbf{f} \text{ with argument } \overline{v}\}$$

Naturally, a terminating program is also quasiterminating.

A *necessary condition for termination of offline specialization* is that *the program is quasiterminating in its static parameters*. We define this more formally:

Definition 5. Let program `p` contain definition $\mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_n) = \text{expression}$. Parameter \mathbf{x}_i is of bounded static variation (BSV for short) if for all static inputs \overline{sin} , the following set is finite:

¹¹ Remark: this is a delicate property, holding only because this `interp` implements “static name binding.” Changing the `call` code as follows to implement dynamic name binding would make `ns` necessarily dynamic during specialization:

```
call f(e1, ..., en) : [9] eval(lkbody(f, pg), append(lkparm(f, pg), ns), ...)
```

$$StatVar(\mathbf{x}_i, \overline{sin}) = \left\{ v_i \mid \begin{array}{l} (\mathbf{f}, v_1 \dots v_n) \in Reach(\overline{in}) \text{ for some} \\ \text{inputs } \overline{in} \text{ with } \overline{sin} = \text{staticpart}(\overline{in}) \end{array} \right\}$$

Two illustrative examples both concern the program `double` of Section 2.3.3. Program `double` is terminating; and parameter `u` is of BSV, while `v` is not.

```
double(x)      = [1] dblplus(x,0)
dblplus(u,v)  = if u <= 0 then v else [2] dblplus(u-1,v+2)
```

Case 1: Input `x` is static, and

$$\begin{aligned} StatVar(\mathbf{u}, x) &= \{x, x-1, \dots, 1, 0\} \\ StatVar(\mathbf{v}, x) &= \{0, 2, \dots, 2x\} \end{aligned}$$

Both are finite for any static input `x`. The BTA can annotate both `u` and `v` as static because `dblplus` will only be called with finitely many value pairs $(\mathbf{u}, \mathbf{v}) = (x, 0), (x-1, 2), \dots, (0, 2x)$, where x is the initial value of `x`.

Case 2: Input `x` is dynamic, and

$$\begin{aligned} StatVar(\mathbf{u}, \varepsilon) &= \{0, 1, 2, \dots\} \\ StatVar(\mathbf{v}, \varepsilon) &= \{0, 2, 4, \dots\} \end{aligned}$$

(Here ε is the empty list of static program inputs.) Parameter `u` must be dynamic because it depends on `x` at call site [1]. Further, BTA *must not* annotate `v` as static because if it did so, then `dblplus` would be specialized infinitely, to: $\mathbf{v} = 0, \mathbf{v} = 2, \mathbf{v} = 4, \dots$

4.5 “Must-Decrease” and “May-Increase” Properties

Detecting violations of the BSV condition requires, in addition to the *must-decrease* parameter size properties on which the size-change approach is based, *may-increase* properties as well. It is not difficult to devise abstract interpretation or constraint solving analyses to detect either modality of size-change behavior, see [41] Section 14.3, or [10,39].

A simple extension of the size-change graph formalism uses *two-layered* size-change graphs $G = (G^\uparrow, G^\downarrow)$ where (as before) G^\downarrow approximates “must-decrease” properties on which the size-change approach is based, and G^\uparrow safely approximates “may-increase” size relations by arcs $\mathbf{x} \xrightarrow{\uparrow} \mathbf{y}$. An example: the `double` program seen before has two-layered size-change graphs:

$$G_1 = \left\{ \begin{array}{l} G_1^\uparrow : \\ \hline G_1^\downarrow : \end{array} \right. \begin{array}{c} \boxed{\begin{array}{c} \mathbf{x} \xrightarrow{=} \mathbf{u} \\ \mathbf{v} \\ \hline \mathbf{x} \xrightarrow{=} \mathbf{u} \\ \mathbf{v} \end{array}} \end{array} \quad G_2 = \left\{ \begin{array}{l} G_2^\uparrow : \\ \hline G_2^\downarrow : \end{array} \right. \begin{array}{c} \boxed{\begin{array}{c} \mathbf{u} \xrightarrow{\downarrow} \mathbf{u} \\ \mathbf{v} \xrightarrow{\uparrow} \mathbf{v} \\ \hline \mathbf{u} \xrightarrow{\downarrow} \mathbf{u} \\ \mathbf{v} \quad \mathbf{v} \end{array}} \end{array} \quad \begin{array}{l} \text{“May-increase”} \\ \text{“Must-decrease”} \end{array}$$

4.6 Constraints on Binding-Time Analysis

The purpose of binding-time analysis is safely to annotate a program. The central task is to find a so-called *division* $\beta : \text{ParameterNames}^{12} \rightarrow \{S, D, U\}$ that classifies every function parameter as “static,” “dynamic,” or “as yet undecided.” The desired division should have no U values and be “as static as possible” while ensuring that specialization will terminate in all cases. To this end define $\beta \sqsupseteq \beta'$ to hold iff $\beta'(\mathbf{x}) \neq U$ implies $\beta(\mathbf{x}) = \beta'(\mathbf{x})$ for all parameters \mathbf{x} .

Constraints on a division β :

1. $\beta \sqsupseteq \beta_0$ where β_0 is the *initial division*, mapping each program input parameter to its given binding time and all other parameters to U .
2. $\beta(\mathbf{x}) = D$ if \mathbf{x} is not of BSV.
3. $\beta(\mathbf{x}) = D$ if the value of \mathbf{x} depends on the value of some \mathbf{y} with $\beta(\mathbf{y}) = D$.
4. $\beta(\mathbf{x}) = D$ if \mathbf{x} depends on the result of a residual call.

We now proceed to develop two principles that can be used to assign $\beta(\mathbf{x}) := S$, both using ideas from size-change termination.

4.7 Second and Better Try: Bounded Domination

A new principle for termination of offline program specialization:

Let β satisfy the Section 4.6 constraints and $\beta(\mathbf{x}) = U$.¹³ If no valid infinite call sequence causes \mathbf{x} to increase infinitely, then \mathbf{x} is of BSV.

This analysis goes considerably farther than the “call-free” method of Section 4.2. Further, it is relatively easy to implement using familiar graph algorithms. First, define $\mathbf{x} \equiv \mathbf{w}$ iff \mathbf{x} depends on \mathbf{w} and \mathbf{w} depends on \mathbf{x} , and define the equivalence class: $[\mathbf{x}] = \{\mathbf{w} \mid \mathbf{x} \equiv \mathbf{w}\}$.

- A. Compute the closure \mathcal{S} of program \mathbf{p} ’s size-change graphs (now two-layered).
- B. Identify, as a candidate for upgrading, any \mathbf{x} such that $\beta(\mathbf{x}) = U$, and $\beta(\mathbf{y}) = S$ whenever \mathbf{x} depends on $\mathbf{y} \notin [\mathbf{x}]$.
- C. Reclassify parameters $\mathbf{z} \in [\mathbf{x}]$ by $\beta(\mathbf{z}) := S$ if no idempotent $G \in \mathcal{S}$ contains $\mathbf{w} \xrightarrow{\uparrow} \mathbf{w}$ for any $\mathbf{w} \in [\mathbf{x}]$.

For **double**, C allows changing initial $\beta = [\mathbf{x} \mid \neg S, \mathbf{u} \mid \neg U, \mathbf{v} \mid \neg U]$ into

$$\beta = [\mathbf{x} \mid \neg S, \mathbf{u} \mid \neg S, \mathbf{v} \mid \neg U]$$

Ackermann specialization terminates by similar reasoning. Now consider the interpreter example of Figure 2 with initial

$$\beta = [\text{prog} \mid \neg S, \mathbf{d} \mid \neg D, \mathbf{e} \mid \neg U, \mathbf{ns} \mid \neg U, \mathbf{vs} \mid \neg U, \mathbf{pg} \mid \neg U, \mathbf{f} \mid \neg U, \mathbf{x} \mid \neg U]$$

¹² We assume without loss of generality all functions have distinct parameter names, and that the program contains no calls to its initial function.

¹³ Note that \mathbf{x} must be potentially static, by Constraint 3.

`pg` is clearly of BSV because it is copied in all calls and never changed (increased or decreased). Further, call sites 2, 3 and 5–10 only pass values to `e` from substructures of `e` or `pg`, so the values of `e` are always a substructure of the static input. Thus `e` is of BSV by the Bounded Domination principle and can be annotated “static” (even though at call site [9] its value can become larger from one call to the next.) This and constraint 3 of Section 4.6 gives:

$$\beta = [\text{prog} \vdash \neg\mathcal{S}, d \vdash \neg\mathcal{D}, e \vdash \neg\mathcal{S}, \text{ns} \vdash \neg\mathcal{U}, \text{vs} \vdash \neg\mathcal{D}, \text{pg} \vdash \neg\mathcal{S}, f \vdash \neg\mathcal{S}, x \vdash \neg\mathcal{S}]$$

On the other hand, call site [4] poses a problem: `ns` can increase, which (so far) will cause it to be annotated “dynamic.” This is more conservative than necessary, as the name list `ns` only takes on finitely many values when interpreting any fixed program `pg`.

4.8 Third and Still Better Try: Bounded Anchoring

A still more general principle allows potentially static parameters that *increase*:

Suppose $\beta(x) = U$ and $\beta(y) = S$ where β satisfies the constraints of Section 4.6. If every valid infinite call sequence cs that infinitely increases x also infinitely *decreases* y , then x is of BSV.

The set of known BSV parameters can iteratively be extended by this principle, starting with ones given by Bounded Domination¹⁴. Again, it is relatively easy to implement:

- A. Compute the closure \mathcal{S} of program p ’s size-change graphs.
- B. Identify, as a candidate for upgrading, any x such that $\beta(x) = U$, and $\beta(y) = S$ whenever x depends on $y \notin [x]$.
- C. Reclassify all parameters $z \in [x]$ by $\beta(z) := S$ if every idempotent $G \in \mathcal{S}$ containing $w \xrightarrow{\uparrow} w$ for some $w \in [x]$ *also* contains $y \xrightarrow{\downarrow} y$ for some y with $\beta(y) = S$.

For the `double` example, C allows changing $\beta = [x \vdash \neg\mathcal{S}, u \vdash \neg\mathcal{S}, v \vdash \neg\mathcal{U}]$ into

$$\beta = [x \vdash \neg\mathcal{S}, u \vdash \neg\mathcal{S}, v \vdash \neg\mathcal{S}]$$

In the interpreter example, the previous analysis shows that `e` and `pg` are of BSV. The remaining parameter `ns` can increase (call 4), but a call sequence $\dots[4]\dots$ with an *in situ* increase of `ns` also has an *in situ* decrease in `e`. This implies `ns` cannot increase unboundedly and so is also of BSV. Conclusion (as desired):

$$\beta = [\text{prog} \vdash \neg\mathcal{S}, d \vdash \neg\mathcal{D}, e \vdash \neg\mathcal{S}, \text{ns} \vdash \neg\mathcal{S}, \text{vs} \vdash \neg\mathcal{D}, \text{pg} \vdash \neg\mathcal{S}, f \vdash \neg\mathcal{S}, x \vdash \neg\mathcal{S}]$$

The only dynamic parameter of function `eval` is `vs`, so target programs obtained by specialization will be free of all source program syntax.

¹⁴ In fact, the Bounded Domination principle can be seen as the special case of Bounded Anchoring where the set of call sequences that infinitely increase x is empty.

4.9 Specialization Point Insertion

The discussion above was rather quick and did not cover Section 4.6, Constraint 4, which concerns the function unfolding policy to use. The justification of Constraint 4 is that a residual call result is not available at specialization time.

Infinite specialization-time unfolding can be prevented by doing no unfolding at all, but by Constraint 4 this could force other parameters to be dynamic. (For the interpreter example, it would force functions `lkboddy` etc. and in turn also `e` and `ns` to be dynamic, which is definitely not desirable.)

A more liberal unfolding policy can be based upon a *specialization-point insertion* analysis to mark a limited set of call sites as specialization points, not to be unfolded – as few as possible, just enough to prevent infinite loop unrolling. This is not explained here, since the paper is already long enough. More details, and correctness proofs, can be found in [29] and the forthcoming [30].

5 Conclusion and Directions for Future Work

The problem of termination of generated programs and program generators must be solved before fully automatic and reliable program generation for a broad application range becomes a reality. Achieving this goal is *necessary*, if we hope ever to elevate software engineering from its current state (a highly-developed handiwork) into a successful branch of engineering, able to solve a wide range of new problems by systematic, well-automated and well-founded methods.

We have described recent progress towards taming these rather intricate problems in the context of partial evaluation. While the large lines are becoming clearer, there is still much to do to bring these ideas to the level of practical, day-to-day usability. We conclude with a long list of goals and challenges.

Termination analysis:

- Apply termination analysis to multi-stage languages, real-time systems [71], strictness analysis, automatic theorem proving (type theory).
- Develop a termination analysis for a realistic programming language, e.g., C, Java or C#.
- Develop termination-guaranteeing BTAs for
 - a functional language—e.g., Scheme.
 - an imperative language—e.g., C.
 - more widely used programming languages, e.g., Java, C#.
- Develop a still stricter BTA to identify programs whose specialized versions will always terminate.
- Find ways to combine termination analysis with
 - *abstract interpretation*
 - *other static analyses*. For example, the size-change analysis depends critically on the fact that data is well-founded (e.g., natural numbers or lists), but the more common integer type is *not* well-founded.

One approach is to apply abstract interpretation to the type of integers to recognize non-negative parameters, and somehow combine this information with size-change analysis.

Partial evaluation:

- Perform efficient, reliable, predictable specialization of realistic programming languages, e.g., C, Java and C#.
- Find annotations to ensure preservation of effects.

Static program analyses:

- Devise an *overlap* analysis to discover when a function can be called repeatedly with the same arguments. Memoization of such programs can yield superlinear speedups, but costs a high time and space overhead.
- Devise a way automatically to estimate or bound a program's running time, as a function of its input size or value.

Acknowledgments. We would like to thank Niels H. Christensen, Olivier Danvy, Julia L. Lawall, Jens Peter Secher, Walid Taha and anonymous reviewers for valuable comments and suggestions for improvements of this paper.

References

1. Andreas Abel and Thorsten Altenkirch. A semantical analysis of structural recursion. In *Abstracts of the Fourth International Workshop on Termination WST'99*, pages 24–25. unpublished, May 1999.
2. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.
3. Thomas Arts and Jürgen Giesl. Proving innermost termination automatically. In *Proceedings Rewriting Techniques and Applications RTA'97*, volume 1232 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 1997.
4. Andrew Berlin and Daniel Weise. Compiling scientific code using partial evaluation. *IEEE Computer*, 23(12):25–37, 1990.
5. Lars Birkedal and Morten Welinder. Hand-writing program generator generators. In M. Hermenegildo and J. Penjam, editors, *Proceedings of the 6th International Symposium on Programming Language Implementation and Logic Programming (PLILP '94)*, pages 198–214. Springer-Verlag, September 1994.
6. Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17:3–34, 1991.
7. Anders Bondorf and Jesper Jørgensen. Efficient analyses for realistic off-line partial evaluation: extended version. Technical Report 93/4, DIKU, University of Copenhagen, Denmark, 1993.
8. Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.
9. Jiazhen Cai, P. Facon, Fritz Henglein, Robert Paige, and Edmond Schonberg. Type analysis and data structure selection. In *Constructing Programs From Specifications*, pages 325–347. North-Holland, 1991.
10. Wei-Ngan Chin and Siau-Cheng Khoo. Calculating sized types. *Higher-Order and Symbolic Computation*, 14(2/3):261–300, 2002.
11. Wei-Ngan Chin, Siau-Cheng Khoo, and Tat-Wee Lee. Synchronisation analysis to stop tupling. In *Programming Languages and Systems (ESOP'98)*, pages 75–89, Lisbon, 1998. Springer LNCS 1381.
12. Niels H. Christensen, Robert Glück, and Søren Laursen. Binding-time analysis in partial evaluation: One size does *not* fit all. In D. Bjørner, M. Broy, and A. V. Zamulin, editors, *Perspectives of System Informatics. Proceedings*, volume 1755 of *Lecture Notes in Computer Science*, pages 80–92. Springer-Verlag, 2000.

13. Michael A. Colón and Henny B. Sipma. Practical methods for proving program termination. In *Conference on Computer-Aided Verification (CAV)*, Lecture Notes in Computer Science, pages ??–?? Springer-Verlag, 2002.
14. Charles Consel. A tour of Schism: a partial evaluation system for higher-order applicative languages. In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 66–77, 1993.
15. Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *ACM Symposium on Principles of Programming Languages*, pages 493–501. ACM Press, 1993.
16. Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In *ACM Symposium on Principles of Programming Languages*, pages 145–156, 1996.
17. Catarina Coquand. The interactive theorem prover Agda. <http://www.cs.chalmers.se/~catarina/agda/>, 2001.
18. James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering*, pages 439–448. IEEE Press, 2000.
19. Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *4th POPL, Los Angeles, CA*, pages 238–252, Jan. 1977.
20. Olivier Danvy, Robert Glück, and Peter Thiemann, editors. *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
21. Manuvir Das. *Partial Evaluation using Dependence Graphs*. PhD thesis, University of Wisconsin-Madison, February 1998.
22. Manuvir Das and Thomas Reps. BTA termination using CFL-reachability. Technical Report 1329, Computer Science Department, University of Wisconsin-Madison, 1996.
23. Yoshihiko Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999. Reprinted from *Systems · Computers · Controls* 2(5), 1971.
24. Yoshihiko Futamura. Partial evaluation of computation process, revisited. *Higher-Order and Symbolic Computation*, 12(4):377–380, 1999.
25. John Gallagher and Maurice Bruynooghe. Some low-level source transformations for logic programs. In M. Bruynooghe, editor, *Proceedings of the Second Workshop on Meta-Programming in Logic, April 1990, Leuven, Belgium*, pages 229–246. Department of Computer Science, KU Leuven, Belgium, 1990.
26. John P. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98. ACM Press, 1993.
27. Steve Ganz, Amr Sabry, and Walid Taha. Macros as Multi-Stage computations: Type-Safe, generative, binding macros in MacroML. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP-01)*, volume 36, 10 of *ACM SIGPLAN notices*, pages 74–85, New York, September 3–5 2001. ACM Press.
28. Arne Glenstrup, Henning Makhholm, and Jens Peter Secher. C-Mix — specialization of C programs. In Hatcliff et al. [35], pages 108–154.
29. Arne John Glenstrup. Terminator II: Stopping partial evaluation of fully recursive programs. Master's thesis, DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen Ø, June 1999.

30. Arne John Glenstrup. Partial evaluation, termination analysis, and specialisation-point insertion. In *preparation*, 2002.
31. Arne John Glenstrup and Neil D. Jones. BTA algorithms to ensure termination of off-line partial evaluation. In *Perspectives of System Informatics: Proceedings of the Andrei Ershov Second International Memorial Conference*, Lecture Notes in Computer Science. Springer-Verlag, June 1996.
32. Robert Glück, Ryo Nakashige, and Robert Zöchling. Binding-time analysis applied to mathematical algorithms. In J. Doležal and J. Fidler, editors, *System Modelling and Optimization*, pages 137–146. Chapman & Hall, 1995.
33. Robert Glück and Morten Heine Sørensen. A roadmap to metacomputation by supercompilation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 137–160. Springer-Verlag, 1996.
34. Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1–2):147–199, 2000.
35. John Hatcliff, Torben Mogensen, and Peter Thiemann, editors. Partial Evaluation: Practice and Theory. Proceedings of the 1998 DIKU International Summerschool, volume 1706. Springer-Verlag, 1999.
36. Carsten Kehler Holst and John Launchbury. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming, Skye, Scotland*, pages 210–218. Glasgow University, 1991.
37. Paul Hudak. Building domain specific embedded languages. *ACM Computing Surveys*, 28A:(electronic), December 1996.
38. John Hughes. Type specialisation for the λ -calculus; or a new paradigm for partial evaluation based on type inference. In Danvy et al. [20], pages 183–215.
39. John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *ACM Symposium on Principles of Programming Languages*, pages 410–423. ACM Press, 1996.
40. Neil D. Jones. What *Not* to do when writing an interpreter for specialisation. In Danvy et al. [20], pages 216–237.
41. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall. Download accessible from www.diku.dk/users/neil, 1993.
42. Neil D. Jones and Flemming Nielson. Abstract interpretation: a semantics-based tool for program analysis. In *Handbook of Logic in Computer Science*. Oxford University Press, 1994. 527–629.
43. Richard B. Kieburtz, Laura McKinney, Jeffrey Bell, James Hook, Alex Kotov, Jeffrey Lewis, Dino Oliva, Tim Sheard, Ira Smith, and Lisa Walton. A software engineering experiment in software component generation. In *18th International Conference in Software Engineering*, pages 542–553, 1996.
44. John Launchbury. *Projection Factorisations in Partial Evaluation*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1991.
45. Julia L. Lawall and Peter Thiemann. Sound specialization in the presence of computational effects. In M. Abadi and T. Ito, editors, *Proceedings of the 3rd International Symposium on Theoretical Aspects of Computer Software (TACS'97)*, number 1281 in *Lecture Notes in Computer Science*, pages 165–190, September 1997.

46. Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *ACM Symposium on Principles of Programming Languages*, volume 28, pages 81–92. ACM Press, January 2001.
47. Michael Leuschel. On the power of homeomorphic embedding for online termination. In *Static Analysis. Proceedings*, volume 1503, pages 230–245. Springer-Verlag, September 1998.
48. Michael Leuschel and Maurice Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2002.
49. Naomi Lindenstrauss, Yehoshua Sagiv, and Alexander Serebrenik. Termilog: A system for checking termination of queries to logic programs. In Orna Grumberg, editor, *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, Jun 22–25, 1997*, volume 1254 of *Lecture Notes in Computer Science*, pages 444–447. Springer, 1997.
50. Y.Ä. Liu. Efficiency by incrementalization: an introduction. *Journal of Higher-Order and Symbolic Computation*, 13(4):289–313, 2000.
51. John W. Lloyd and John C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11:217–242, 1991.
52. Dylan McNamee, Jonathan Walpole, Calton Pu, Crispin Cowan, Charles Krasic, Ashvin Goel, Perry Wagle, Charles Consel, Gilles Muller, and Renaud Marlet. Specialization tools and techniques for systematic optimization of system software. *ACM Transactions on Computer Systems*, 19(2):217–251, 2001.
53. Torben Mogensen. Partially static structures in a self-applicable partial evaluator. In D. Björner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 325–347. North-Holland, 1988.
54. Eugenio Moggi, Walid Taha, Zine-El-Abidine Benaïssa, and Tim Sheard. An idealized MetaML: Simpler, and more expressive. In *European Symposium on Programming*, volume 1576, pages 193–207, 1999.
55. Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291–1000, USA. Available online from <ftp://cse.ogi.edu/pub/tech-reports/README.html>.
56. Rosza Péter. *Rekursíve Funkciók (Recursive Functions)*. Akadémiai Kiadó, Budapest (Academic Press, New York), 1951 (1976).
57. Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek. ‘C and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21(2):324–369, March 1999.
58. Aravinda Prasad Sistla, Moshe Y. Vardi, and Pierre Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987.
59. Konstantinos F. Sagonas, Terrance Swift, and David Scott Warren. XSB as an efficient deductive database engine. In Richard T. Snodgrass and Marianne Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24–27, 1994*, pages 442–453. ACM Press, 1994.
60. Danny De Schreye, Robert Glück, Jesper Jørgensen, Michael Leuschel, Bern Martens, and Morten H. B. Sørensen. Conjunctive partial deduction: Foundations, control, algorithms, and experiments. *Journal of Logic Programming*, 41(2&3):231–277, 1999.
61. Ulrik Schultz. Partial evaluation for class-based object-oriented languages. In *PADO*, pages 173–197, 2001.

62. Peter Sestoft. Bibliography on partial evaluation and mixed computation (bibtex format, online <ftp://ftp.diku.dk/diku/semantics/partial-evaluation/>). Technical report, DIKU (Computer Science, University of Copenhagen), 2001.
63. Tim Sheard. Using MetaML: A staged programming language. *Lecture Notes in Computer Science*, 1608:207–??, 1999.
64. Litong Song and Yoshihiko Futamura. A new termination approach for specialization. In [70], pages 72–91, 2000.
65. Morten Heine Sørensen and Robert Glück. An algorithm of generalization in positive supercompilation. In J.W. Lloyd, editor, *Logic Programming: Proceedings of the 1995 International Symposium*, pages 465–479. MIT Press, 1995.
66. Chris Speirs, Zoltan Somogyi, and Harald Søndergaard. Termination analysis for Mercury. In Pascal Van Hentenryck, editor, *Static Analysis, Proceedings of the 4th International Symposium, SAS '97, Paris, France, Sep 8–19, 1997*, volume 1302 of *Lecture Notes in Computer Science*, pages 160–171. Springer, 1997.
67. Michael Sperber and Peter Thiemann. Generation of LR parsers by partial evaluation. *ACM Transactions on Programming Languages and Systems*, 22(2):224–264, March 2000.
68. Walid Taha. A sound reduction semantics for untyped CBN multi-stage computation. or, the theory of MetaML is non-trivial. *ACM SIGPLAN Notices*, 34(11):34–43, November 1999. Extended abstract.
69. Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [55].
70. Walid Taha, editor. *Semantics, Applications, and Implementation of Program Generation*, volume 1924 of *Lecture Notes in Computer Science*, Montréal, 2000. Springer-Verlag.
71. Walid Taha, Paul Hudak, and Zhanyong Wan. Directions in functional programming for real(-time) applications. In *the International Workshop on Embedded Software (ES '01)*, volume 221 of *Lecture Notes in Computer Science*, pages 185–203, Lake Tahoe, 2001. Springer-Verlag.
72. Walid Taha, Henning Makholm, and John Hughes. Tag elimination and Jones-optimality. In *PADO*, pages 257–275, 2001. <http://cs-www.cs.yale.edu/homes/taha/publications/preprints/pado00.dvi>.
73. Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1–2):211–242, October 2000.
74. Peter Thiemann. A unified framework for binding-time analysis. In M. Bidoit and M. Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development, Lille, France, April 1997. (Lecture Notes in Computer Science, vol. 1214)*, pages 742–756. Springer-Verlag, 1997.
75. Peter Thiemann. Aspects of the pgg system: Specialization for standard scheme. In Hatcliff et al. [35], pages 412–432.
76. Valentin F. Turchin. A supercompiler system based on the language Refal. *SIGPLAN Notices*, 14(2):46–54, February 1979.
77. Philip Wadler. Deforestation: Transforming programs to eliminate trees. In H. Ganzinger, editor, *ESOP'88. 2nd European Symposium on Programming, Nancy, France, March 1988. (Lecture Notes in Computer Science, vol. 300)*, pages 344–358. Springer-Verlag, 1988.
78. Hongwei Xi. Dependent types for program termination verification. volume 15, pages 91–132, 2002.

Generative Programming for Embedded Systems

Janos Sztipanovits and Gabor Karsai

Institute for Software Integrated Systems
Vanderbilt University, P.O. Box 1829 Sta. B. Nashville, TN 37235, USA
{[sztipaj.gabor](mailto:sztipaj.gabor@vuse.vanderbilt.edu)}@vuse.vanderbilt.edu

Abstract. Embedded systems represent fundamentally new challenges for software design, which render conventional approaches to software composition ineffective. Starting with the unique challenges of building embedded systems, this paper discusses key issues of model-based technology for embedded systems. The discussion uses Model-Integrated Computing (MIC) as an example for model-based software development. In MIC, domain-specific, multiple view models are used in all phases of the development process. Models explicitly represent the embedded software and the environment it operates in, and capture the requirements of the application, simultaneously. Models are *descriptive*, in the sense that they allow the formal analysis, verification and validation of the embedded system at design time. Models are also *generative*, in the sense that they carry enough information for automatically generating embedded systems from them using the techniques of program generators.

1 Introduction

Perhaps the biggest impact of the “IT explosion” in the last decade has been the emerging role of computing and software as “universal system integrator.” Systems are formed from interacting components. The new trend is that an increasing number of components and interactions in real-life systems, which were previously physical, are becoming *computational*. From large-scale systems, such as manufacturing processes or command and control (C^2) systems, to small systems, such as automobiles and simple appliances, interaction and coordination of physical components increasingly involves digital information processing and communication. For example, the automotive industry is currently testing “brake-by-wire” systems, where the currently dominant hydraulic and mechanical brake systems will be replaced by position sensors observing the brake pedal, electrical actuators exerting braking force, and a distributed, embedded computer system, which computes the optimum force distribution according to the driving conditions. Two important consequences of this change are the following:

- **Increased fusion of software into application domains.** In many application domains, computing has become the focal point of complexity and the primary source of new functionality. For example, over 90% of innovations in the automotive industry come from embedded computing [33]. The increased significance of computing means that unless unique characteristics of the application domain are reflected directly in the software development paradigms, application engineering considerations must be mapped manually onto general-

- purpose software engineering concepts and tools, which is tedious and error-prone for both the domain experts and the software experts. The difficulty of this manual mapping process creates the need for sharply tailored capabilities, such as domain-specific languages, generators, and composition platforms that enable building of applications from components.
- **Physicality in software design.** In embedded computing applications, the role of the embedded software is to configure and control the operation of programmable computing devices so as to meet physical requirements at their sensor-actuator interfaces. This deep integration of computing with physical systems implies that essential physical characteristics of systems (such as latency, noise, power consumption) are strongly influenced — or simply determined by — software. Consequently, software requirements become multi-faceted, i.e., computational platforms and software must satisfy functional and physical requirements *simultaneously*.

The goal of this paper is to discuss challenges and opportunities of generative programming for embedded software development. We use the term “generative programming” in a broad sense: systems or components of systems are produced automatically from abstract terms [28]. The examples for possible solutions are based on our experience gained from the *Model-Integrated Computing (MIC)* effort at the Institute of Software Integrated Systems (ISIS) at Vanderbilt University [1].

The outline of the paper is the following: In Section 2, we examine the challenges of composing complex systems from components and describe a model-based extension of composition platforms. Section 3 summarizes the challenges and discusses the MIC approach in modeling languages and model building. In Section 4, we provide an overview of generator technologies in the MIC framework. Section 5 summarizes some of the relevant approaches and compares them with MIC.

2 Significance of Generative Programming

Composition and component-based design are key tools in modern software engineering for managing complexity. The concept of component-based design is straightforward: systems are built by composing software components with precisely defined interfaces using standardized interconnection mechanisms. “Plug-and-play” construction is supported by an underlying composition framework, such as CORBA or COM+, which facilitates the component interactions by providing standard services such as a request broker, interface repository, location service, and others. Unfortunately, this solution tends to work either for building systems with relatively coarse-grained components using small amounts of “glue code”, or for very small systems with a few components because of the following two problems:

1. Component interfaces in conventional standards-based composition frameworks only capture the signature, but not the semantics of components. Consequently, design integrity for the composed system may quickly be lost and inconsistencies may emerge by simply combining components based on compatibility of interface signatures.
2. To achieve flexibility, components are frequently designed to be customizable to different application contexts via parameterization or via the selection of alternative

implementations. In large systems, these choices represent complex, interacting assumptions about operating conditions, which may lead to brittleness.

The success of building applications based on coarse-grained components is the result of strong restrictions on component interactions: the dominant components (such as databases or web browsers) preserve the overall design integrity. In case of small systems, the system designers can preserve design integrity without extensive tool support through heroic manual effort.

Although software composition is rarely easy, the problems of software composition for embedded systems are particularly hard. Physical processes surround embedded computers, which receive their inputs from sensors and send their outputs to actuators. When viewed from their sensor and actuator interfaces, embedded computing devices act like physical processes with dynamics, noise, fault, size, power and other physical characteristics. *The role of the embedded software is to “configure” the computing device in order to meet its physical requirements* [2]. The mapping of logical behavior into physical behavior is influenced by the detailed physical characteristics of the devices involved, including their physical architecture, instruction execution speed, bus bandwidth, power dissipation, etc. In addition, modern processor architectures introduce complex interactions between the software and essential physical characteristics of the underlying devices.

It is not surprising that using current software technology logical/functional composability does not imply physical composability. In fact, *physical properties are not composable*, rather, they appear as cross-cutting constraints in the development process. The effects of such cross-cutting constraints can be devastating for the design. Meeting specifications in one part of the system may destroy performance in others, and, additionally, many of the problems will surface at system integration time rather than during unit development and testing. Consequently, we need to change our approach to the design of embedded software: productivity increases must come from tools that directly address the design of the whole system with its many different physical and logical aspects.

There are several comprehensive approaches, such as Model-Integrated Computing (MIC) [1], Aspect-Oriented Programming [3], Intentional Programming [4], GenVoca Architecture [5], that attempt to answer problems of composing complex systems. In our discussion, we focus on MIC and point out some of the similarities and differences with the other approaches.

Figure 1 shows an overview of the MIC architecture. The applications and infrastructure software (left side) are defined by application models and platform models. The difference between MIC for embedded and non-embedded systems is particularly significant regarding the scope and composition of models. In order to make the physical properties of the embedded system computable and analyzable, models capturing only the logical characteristics of applications and infrastructure software are insufficient. The models must include physical properties of the platforms and the mapping between the application and platform models. The scope of modeling and the required level of abstraction are highly domain-specific. We cannot expect that the same types of models are used to design controllers for break-by-wire system in cars (where safety, timing and cost are the critical properties) and to design mobile phones (where besides cost, power, security, and feature richness are the most important factors).

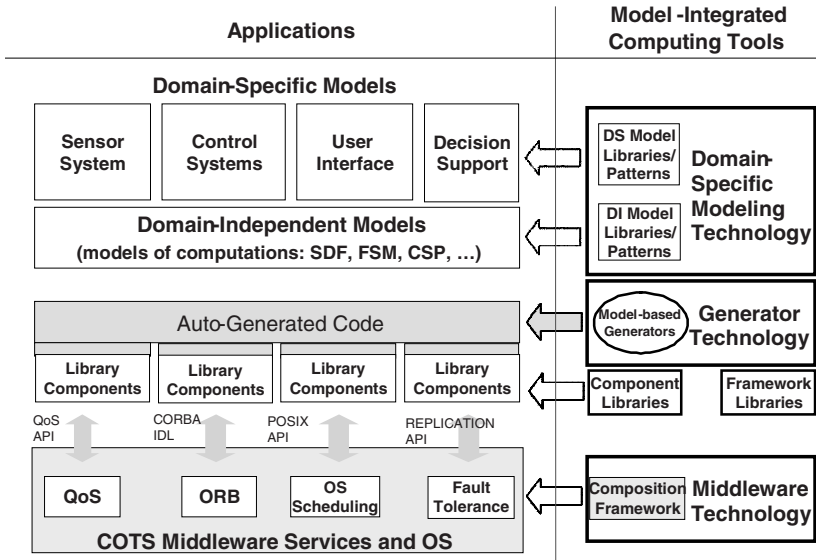


Fig. 1. Model-integrated approach to software composition

Model-based approaches are not practical without extensive tool support. The right side of Figure 1 shows the MIC tool components. Support for modeling includes tools for building domain-specific modeling environments, tools for analyzing and synthesizing models, and model translators that help to integrate large, heterogeneous tool environments. Section 3 discusses the main components of the MIC modeling framework.

While models define and characterize the embedded system applications, the applications themselves are formed by a set of customized components and the underlying hardware/software platform (see Figure 1). There are numerous hardware/software platforms that are commonly used for building embedded system applications. For example, the Time-Triggered Architecture (TTA) [6] and the Real-time CORBA Event Service [7] provide composition platforms for embedded software and systems with very different properties. While TTA supports a time triggered, synchronous model of computation, the Real-time CORBA Event Service follows an event-triggered, fully asynchronous model. The platforms have a very important role in model-based design: they enable the use of idealized assumptions (such as synchrony, loss-less communication, guaranteed communication bandwidth) about the interaction and behavior of components, which largely simplifies the modeling task. In fact, these idealized and simplified abstractions (or layers of abstractions) between applications and platforms are the basis for platform-based design [8].

Building an integrated application on a platform is a complex task. It includes the consistent parameterization/customization of the selected components and of the hardware/software platform via the provided API-s and creation of any additional new code for connecting the components to the platform. In the MIC framework these steps must be tied directly to the models, otherwise the connection between model

properties and the modeled system will be lost. This need makes the model-based generators and generator technology a fundamental component of the MIC tool architecture. Section 4 examines different approaches for building model-based generators.

3 Modeling and Model Building

The two fundamental problems we consider in this section are the composition of domain-specific modeling languages and the composition of models.

3.1 Domain Specific Modeling Languages

Domain-specific languages (DSLs) have significant impact on the design process [9]. In embedded systems, where computation and communication always occur in the context of a physical domain, DSLs offer an effective way to structure the information about the system to be designed along the “natural dimensions” of the applications. The chief difficulty with DSLs is the cost of developing solid semantic foundations and tools to support their use. MIC resolves this difficulty using a *meta-modeling approach* [14]. Using standard denotational semantics style (see e.g. [10]), a *modeling language* is defined as a five-tuple of concrete syntax (C), abstract syntax (A), semantic domain (S) and semantic and syntactic mappings (M_s and M_c):

$$L = \langle C, A, S, M_s, M_c \rangle$$

The C concrete syntax defines the form of representation, such as visual, textual, or mixed. The A abstract syntax defines the *concepts, relationships, and integrity constraints* available in the language. Thus, the abstract syntax determines all the (syntactically) correct “sentences” (in this case models) that can be built. (It is important to note the A abstract syntax includes semantic elements as well. The integrity constraints, which define well-formedness rules for the models, are frequently called “static semantics”.) The S semantic domain is usually defined by some mathematical formalism in terms of which the meaning of the models is explained. The $M_c : A \rightarrow C$ mapping assigns syntactic constructs (visual, textual, or both) to the elements of the abstract syntax. The $M_s : A \rightarrow S$ semantic mapping relates syntactic concepts to those of the semantic domain.

Rapid composition of *domain-specific modeling languages (DSMLs)* requires tool support. The tools need representation formalism for the syntactic elements (C and A), the semantic domain, and the syntactic and semantic mapping. The languages used for this purpose are called *meta-languages* and the models describing a DSML are called *meta-models*. Since a meta-modeling language can also be considered a domain-specific modeling language (with the domain being that of “modeling languages”), it is desirable that the meta-modeling language be powerful enough to describe itself, in a meta-circular manner. This corresponds to the *four-layer meta-model language architecture* (see Figure 2) used in UML [11] or in CDIF [12].

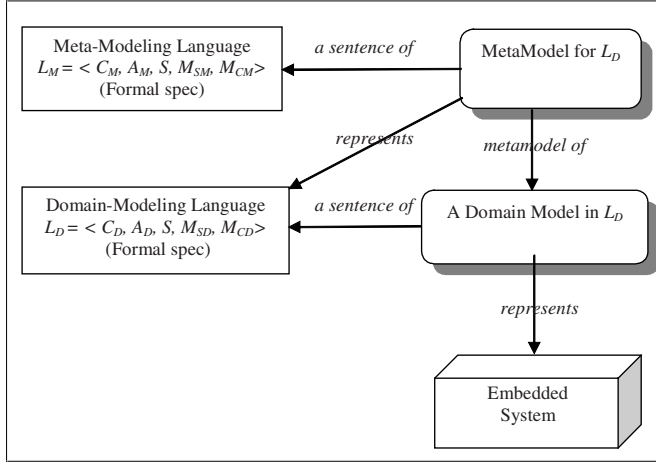


Fig. 2. Meta-modeling language architecture

3.2 Tools for Domain-Specific Modeling

The ISIS toolset for Model Integrated Computing uses the *same* generic, meta-programmable modeling environment (Generic Modeling Environment – GME) for meta-modeling and domain modeling [15]. The GME tool architecture supports meta-circularity: a meta-meta-model (defined in the meta-language) configures the environment to support meta-modeling. We have adopted UML class diagrams (with stereotypes) [11] and the Object Constraint Language (OCL) [13] as the meta-language for modeling the A abstract syntax of DSMLs. This choice was made for the following practical reasons: (a) UML/OCL is an OMG standard that enjoys widespread use in the industry, (b) tools supporting UML are widely available, and (c) familiarity with UML class diagrams helps to mitigate the conceptual difficulty of the meta-modeling language architecture. It is important to note that adopting UML for meta-modeling does not imply any commitment to use UML as a domain modeling language, though it can certainly be used where appropriate. Further details on meta-modeling can be found in [14].

Consider a simple example for the abstract syntax of a DSML for signal flow modeling (SF). The A_{SF} abstract syntax of SF is shown in Figure 3a as a meta-model expressed in terms of a UML class diagram (we have omitted the integrity constraints). The core concepts of this language are Compounds, Primitives, Ports, and Signals. Primitives form the basic signal processing blocks (e.g., Filters, FFT, IFFT, Correlation, etc.). Ports define the I/O interfaces of these blocks, and Signals represent the signal-flow between the blocks. Compounds are processing blocks that can be decomposed into other Compounds, and/or Primitives. An abstract Base concept combines Compounds and Primitives to represent an abstract signal-processing block. This abstract syntax is complemented with the $M_C : A \rightarrow C$ mapping to obtain a simple DSML for signal processing. (The specification of this mapping is part of the meta-programmable Graphical Modeling Environment system

[15] and not shown here.) Figure 3b shows a simple a hierarchical application model, which is an instance of the meta-model in Figure 3a.

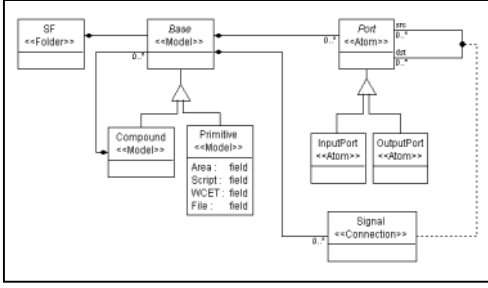


Fig. 3a. Meta-model for SF

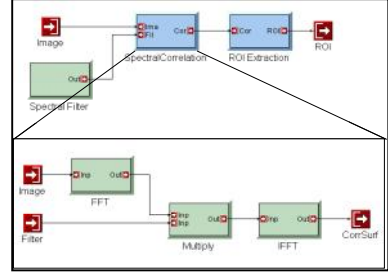


Fig. 3b. Simple application model

In practice, the specification of meta-models can be quite a complex undertaking. It requires a thorough understanding of the S semantic domain and formulation of the well-formedness rules such that the semantic mapping of each well-formed model leads to a consistent semantic model. For example, if SF has synchronous dataflow (SDF) semantics [16], one of the well-formedness rules must prohibit the connection of two different output ports to the same input port:

```
Self.InputPorts() →forall(ip | ip.src() →forall(x1,x2 | x1=x2))
```




However, if we use for SF dynamic dataflow (DDF) semantics [16], the same constraint may not necessarily apply (depending on the actual semantics). (We will discuss the assignment of semantics to SF in Section 4.)

In our experience, DSMLs that are designed to solve real-life engineering problems tend to become complex. In these domains, crucial requirements for robust, domain-specific modeling include careful formulation of meta-models, keeping them consistent with the associated semantic domains, and checking if the created models are well-formed.

3.3 Composing Meta-models

An obvious way to decrease the difficulty of defining DSMLs is to compose them from sub-languages. This is particularly important in application domains such as embedded systems, which frequently require many modeling views. Compositional construction of DSMLs requires the $L_n = L_1 || L_2 \dots || L_k$ composition of meta-models from constituent DSMLs. While the composition of orthogonal (independent) sub-languages is a simple task, construction of DSMLs with non-orthogonal views is a complex problem. Non-orthogonality means that component DSMLs may share concepts and well formed-ness rules may span over the individual modeling views.

Table 1. Meta-model composition operators

Operator	Symbol	Informal semantics
Equivalence		Complete equivalence of two classes
Implementation Inheritance		Child inherits all of the parent's attributes and those containment associations where parent functions as container.
Interface Inheritance		Child inherits all associations except containment associations where parent functions as container.

The current design of ISIS's composable meta-modeling environment in GME leaves the component meta-models intact and creates meta-models that are further composable [17]. The composition is accomplished by introducing three (new) operators for combining meta-models (see Table 1). Application of these operators generates a new meta-model that conforms to the underlying semantics of UML. Decomposition of the UML inheritance operation allows finer control over meta-model composition (details are discussed in [17]). Unfortunately, meta-model composition is not complete by executing the composition operators. If the component meta-models are not orthogonal, it is possible that the resulting meta-model is inconsistent, which means that conflicting integrity constraints are created during the composition process. This means that the meta-modeling toolset needs to be extended with a validation tool (such as PVS [36]), which automatically checks the consistency of the well-formedness rules. (This tool is not part of the current version of GME.)

3.4 Modeling and Model Synthesis

As we pointed out above, modeling and model-based approaches [18][19] play central role in embedded software and system development, where mathematical modeling of physical characteristics is essential. The fundamental promise of model-based approaches is that experimental system verification will largely be replaced by model-based verification. This is extremely important, since testing is very expensive and cannot be exhaustive in real-life systems. Even by taking advantage DSMLs and advanced model-builders, creating high-fidelity models is expensive. Developing efficient methods for model-building is therefore an important research goal. Below, we briefly discuss two important approaches: compositional modeling and model synthesis.

3.4.1 Compositional Modeling

Building complex models by composing components $M_n = M_1 || M_2 \dots || M_k$ is a common, highly desirable technique for efficient modeling. In bottom-up

composition, simpler components are integrated to obtain more complex components. The condition for composability in bottom-up composition is that if a property P_k holds for component M_k , this property will be preserved after integrating M_k with other components. Unfortunately, in embedded systems, many physical properties (such as time dependent properties) are not composable [6]. Therefore, DMSLs that are formal enough to be analyzable and analysis tools that can verify essential properties of designs are crucial in model-based system/software development.

3.4.2 Model Synthesis

Model synthesis in a compositional modeling framework can be formulated as a search problem: given a set of $\{M_p, M_q, \dots, M_k\}$ model components (which may represent different views of the system and may be parameterized), and a set of composition operators, how to select an $M_d = M_i \parallel M_j \dots \parallel M_l$ design (with the required set of parameters) such that a set of $\{P_{1d}, P_{2d}, \dots, P_{kd}\}$ properties for M_d are satisfied? Fully automated synthesis is an extremely hard problem both conceptually and computationally. By narrowing the scope of the synthesis task, however, we can formulate solvable problems. For example, by using patterns [37] and introducing alternative design choices for component templates in generic designs, we can construct a design space using hierarchically layered alternatives. This approach is quite natural in top-down engineering design processes, which makes the construction of a design space using hierarchically layered alternatives relatively simple [20].

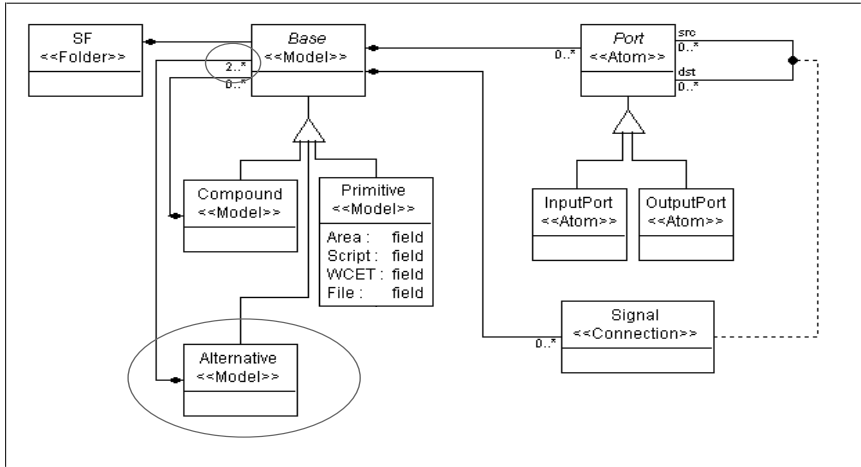


Fig. 4. Meta-model of SF extended with the “Alternative” construct

To enable the representation of design spaces, we need to expand DSMLs with the ability to represent design alternatives explicitly. As an example, Figure 4 shows the meta-model of SF extended with the concept of *Alternatives*. We selected the abstract Base concept for Alternative implementation, and introduced a containment relationship to enable hierarchical composition. An Alternative, in the composed SF meta-model context, can now be defined as a processing block with rigorously defined interface, which contains two or more (notice the cardinality of the containment relation highlighted in the figure) alternative implementations. The

implementations can be Compounds, Primitives, or other Alternatives, with matching interfaces. As we mentioned above, composition of a design requires finding implementation alternatives, which satisfy a set of design properties. The complexity of this task depends largely on the computability of selected design properties from the component properties. A detailed analysis of this problem and a constraint-based design-space pruning technique is described in [21].

4 Generators and Model Transformations

The last step of the model-based development process is to generate the application on the underlying hardware/software platform (see Figure 1) using the verified domain-specific models. In embedded systems where model-based verification is crucial to keep the cost and complexity of experimental system verification tolerable, maintaining a direct link between the verified models and the application is mandatory. The role of the *model-based generators* in the MIC framework is to generate the “glue” required to compose the integrated application from library components, consistently parameterize the components, and customize the composition platform.

4.1 Model-Based Generators

Since generators always work with some selected representation of models, they can be modeled as a mapping between the abstract syntax of their L_i input and L_o output languages: $G: A_i \rightarrow A_o$. Complexity of generators depends largely on the semantic relationship between the input and output languages. Our previous example, SF , is a declarative hierarchical module interconnection language, L_{SF} , which describes the *structure* of signal processing systems. In this sense, we can assign a *structural semantics* to L_{SF} by providing the semantic mapping between its abstract syntax and a semantic domain, which is represented by set-relational mathematics. (A closely related example for this can be found in [22].)

At this point, L_{SF} does not have *behavioral semantics*. Behavioral semantics can be assigned to L_{SF} by defining a mapping between L_{SF} and another modeling language with well-defined operational semantics, such as synchronous dataflow (SDF). For example, we may adopt the SDF behavioral semantics as defined in [23]. Of course, the precise definition of the SDF semantics has also required the definition of the structure of a dataflow graph, which can be expressed by a language L_{SDF} with abstract syntax A_{SDF} . Figure 5 shows a simple meta-model for homogeneous SDF [16], where each node firing consumes and generates exactly one data token at the input and output ports, respectively. We define the operational semantics of L_{SF} by the $G: A_{SF} \rightarrow A_{SDF}$ mapping. In fact, G is also the specification of the generator, which maps $l_{SF} \in L_{SF}$ signal processing system models into $l_{SDF} \in L_{SDF}$ synchronous dataflow models that can be executed by a synchronous dataflow virtual machine.

semantics, with high-level “instructions”. If one omits the technical details of creating the output product in some form (e.g. text files), the main task of a generator is reduced to create a “target tree” from an “input tree”. The “input tree” is a data structure that corresponds to the input abstract syntax tree of compilers, and the “target tree” corresponds to the “output tree” of compilers from which the code is directly “printed”.¹ Naturally, in the simplest of cases the output product can be produced directly from the input tree.

In the most general form, a generator performs the following operations.

1. Construct the input tree. This step is often implicit in model-based systems, as the models *are* the input tree, and the modeling environment directly manipulates a tree-like representation.
2. Traverse the input tree, possibly in multiple passes, and construct an output tree. In this step the generator should visit various the objects in the input tree, recognize patterns of objects, instantiate portions of the target tree, calculate attributes of output objects from attributes of input objects, etc.
3. “Print out” the product. This step creates the result of the generation in the required form: a network of objects, a text file, a sequence of commands issued to a hardware device, etc.

Following the above approach, a generator is straightforward to construct: after designing the input and output data structures (which are determined by the meta-models of the input and target languages already), one has to design the appropriate code sequences for traversal and constructions. The implementation can follow an object-oriented approach: the traversal code can be embedded as methods of the input objects, and by directly coding the traversals and the target construction one can easily implement the generation algorithms. Similarly, the output “printing” can be embedded as methods of the target objects, and by programming the traversal of the *output* one can realize the output-producing algorithms. This direct implementation is simple and works well for situations where the transformations are easy to capture in a procedural form. The disadvantage is that the generator code is hard to maintain and hard to verify.

4.2.2 Pattern-Based Design

The scheme described above can also be implemented in a more structured way, by using the *Visitor* design pattern [24]. The main task of a generator involves the *traversal* of the input tree and *taking actions* at specific points during the traversal. This is clearly in the purview of the Visitor pattern, which offers a common solution for coding the above generic algorithm. In this pattern, a *visitor* object implements the actions to be performed at various nodes in the at tree, while the tree nodes contain code that *accepts* the visitor object and calls the appropriate, node-specific operation on it (while passing itself to the operation as a parameter). The Visitor pattern allows the concise and maintainable implementation of generators, both for the transformation and the printing phases.

While the implementation of a generator following the Visitor pattern is straightforward, it can be improved significantly by using some for of automation. In

¹ We use the term “tree” here, although these data structures are graphs in the most general case. However, even in those cases, a spanning tree of the graph can be found, which “dominates” the structure.

previous work on design tool integration [29], we have developed a technique for the structured capturing of the “traversal/action” code of generators. The approach was based on the observation that traversal sequences and actions to be taken at specific points in the input tree are separate concerns, and that the traversal can be specified using higher-level constructs (than procedural code). A language was designed that allowed the specification of traversal paths and the capturing of actions to be executed at specific nodes. Generators written using this approach were very compact and readable, and have been successfully applied in various projects. The approach is similar to Adaptive Programming [30], but it is more focused on the needs of generators.

4.2.3 Meta-generators

The approach based on the Visitor pattern has a serious shortcoming: most of the logic of the generator is still realized as procedural code, and therefore it is hard to verify or to reason about. A better technique would allow the precise mathematical modeling of the generator’s workings, and the generation of the code of the generator from that model. This process is called *meta-generation*.

A “model of a generator” is much simpler than that of a compiler and it can be defined operationally: it is an abstract description of what the generator does. Following the generic description of a generator above, the main task of a generator can be modeled in terms of (1) the traversal sequences and (2) the transformation actions the generator takes during its operation. The approach described in the previous section allowed the specification of (2) in pure procedural terms, but it can also be specified declaratively using graph-transformation rules.

Graph grammars and graph rewriting [31] offer a structured, formal, and mathematically precise method of representing how a generator constructs the output tree from the input tree. One elementary rewriting operation performed by a translator is called a *transform*. A transform is a specification of a mapping between a portion of the input graph and a portion of the output graph. Note that the meta-models of the input and the output of a generator is a compact description of all the possible input and output structures. If $G_{in} = (C_{in}, A_{in})$ and $G_{out} = (C_{out}, A_{out})$ denote the input and the output meta-models consisting of classes and associations, a transform can be described using the following elements:

- $g_{in} = (c_{in}, a_{in})$: subgraph formed from a subset $c_{in} \subseteq C_{in}$ of the input classes and a subset $a_{in} \subseteq A_{in}$ of the input associations.
- $F : G_{in} \rightarrow \{T, F\}$: a Boolean condition, called *filter*, over G_{in} .
- $g_{out} = (c_{out}, a_{out})$: subgraph formed from a subset $c_{out} \subseteq C_{out}$ of the output classes and a subset $a_{out} \subseteq A_{out}$ of the output associations.
- $M : g_{in} \rightarrow g_{out}$ a mapping where $g_{in} \subseteq G_{in}$, $g_{out} \subseteq G_{out}$, and $F(g_{in}) = T$.

A transform is a specific rewrite rule that converts a sub-graph of the input into a sub-graph of the output. The input sub-graph must also satisfy the filter. The mapping should also specify how the attributes of the output objects and links should be calculated from the attributes of the input objects and links.

While graph transformations are descriptive, they are computationally expensive. Matching the left hand side of a rewriting rule against an input graph involves searching for a sub-graph, which can be of exponential complexity. However, in generators one can almost always avoid the (global) search by specifying the traversal and order in which the transformation rules should be applied, thus the search can be reduced to a (local) matching. This latter one can be accomplished by introducing “pivot nodes”, which are bound by the higher-level, traversal strategy, so the left hand side of the mapping is partially bound already when the rule is fired.

To summarize, a generator can be specified in terms of (1) a graph traversal, which describes in what order the nodes of the input tree should be visited and (2) a set of transformation rules. The implementation of the above scheme is subject of active research. Early experiments [32] indicate the viability of the meta-generator approach.

5 Related Work

It has been increasingly recognized that conventional programming languages are not rich enough to provide efficient support for the composition of complex systems. It is therefore essential to increase the level of abstraction for representing designs and to expand composition from today’s hierarchical, modular composition to multi-faceted, generative composition [25]. Besides model-integrated computing approaches, several other important efforts work toward the same or similar goal. Below we mention three of these directions – Aspect-Oriented Programming (AOP), Intentional Programming (IP) and GenVoca generators – with the limited purpose of examining their approach to the challenges listed in Section 2.

5.1 Aspect-Oriented Programming (AOP)

The goal of AOP is to introduce a new decomposition concept in languages, *aspects*, which crosscut the conventional hierarchical, functional decomposition. AOP provides programmers with the opportunity to express separate concerns independently, and facilitates the merging (weaving) of components in an integrated implementation [3]. Aspect orientation fits well with the need of managing crosscutting constraints in embedded systems. Physical requirements in embedded systems, such as timing or synchrony, can be guaranteed by assigning them to a specific module, but they are the result of implementing their interaction in a particular manner. Changing these requirements may involve widespread changes to the functional components of the system, which makes component-based design and implementation using only functional composition very complicated.

AOP and MIC have strong similarity in addressing multiple-view system design explicitly. Both AOP and MIC allows the separation of design concerns in different aspects and allows capturing and managing interdependence among them. Composition of integrated systems is completed by weaving technology in AOP and model synthesis and generator technology in MIC. The main difference is in the level abstraction used. AOP research focuses on imperative languages and creates aspect-oriented versions such AspectJ [26], while MIC targets DSMLs. Consequently, MIC

is better suited for modeling, verifying and generating large, heterogeneous systems using larger components, while AOP provides better run-time performance due to the use of compiler technology in generating executable systems.

5.2 GenVoca

GenVoca is a generator technology that performs automated composition using precisely defined layers of abstractions in object-oriented languages [5]. The concept is based on the definition and explicit representation of designs layers, where each layer refines the layer above. Design layers have standardized interfaces with alternative implementations. Layers are implemented using DSLs, which are implemented as extensions of existing languages. GenVoca generators convert these composition specifications into the source code of the host language. GenVoca is supported by an extensive toolsuite called the Jakarta Tool Suite (JTS), which provides a common infrastructure for extending standard languages with domain-specific constructs [27].

Regarding the level of abstraction used in describing designs, GenVoca resides between AOP and MIC. GenVoca technology still preserves the advantage of embedded DSLs: the generators output needs to go only to the level of the host language. Similarly to AOP, it results highly efficient code due to the GenVoca-based optimization of component structure and the compiler technology of the host language environment. GenVoca strongly differs from both AOP and MIC in terms of supported decomposition strategy: at this point, GenVoca does not focus on multiple aspect composition (although extension in this direction seems feasible). Interestingly, the design-space exploration techniques used in MIC [21] and the composition validation technique in GenVoca (e.g. [28]) have strong similarities. Both of these techniques are based on a design-space definition using hierarchically layered alternatives and prune the potentially very large space using constraints (in GenVoca the goal is validation, in MIC the goal is to find configurations that satisfies the constraints).

5.3 Intentional Programming (IP)

IP is a bold experiment to transform programming from the conventional, “language focused” activity to a domain-specific, intention-based activity [4]. IP re-factors the conventional programming paradigm into intentional specification and transformers. Intentional specifications encode abstractions in graph data structures (active source) – without assigning a concrete syntax for representing them. Using various transformers manipulating the active source, the intentions can be visualized in different forms, and more importantly, can be assembled into complex programs (represented as intentions) by generators (or meta-programs).

There are many interesting parallels between IP and the other generative programming approaches discussed earlier. For example, in MIC the models (which are the expression of domain-specific constructs, designs) are represented in model databases in a format, which is independent from the concrete syntax used during modeling. Model transformation has a similarly central role in MIC as transformers in IP: the MIC generators, synthesis tools are all directly manipulating the content of

model databases for visualizing, analyzing, and composing models at different phases of the system development. Both in MIC and IP, efficient technology for defining and implementing transformers *is* a crucial issue.

6 Conclusion

DSMLs, model-based generators, and composition frameworks are important elements of software/system technology for embedded computing. In fact, the prevalence of crosscutting constraints in embedded software makes the design process intractable without the extensive use of generative programming. However, the widespread use of these technologies requires drastic decrease in the cost of supporting tools. We believe that this can be achieved by applying the same ideas a level higher: compositional construction of DSMLs via meta-modeling, development of meta-generators, and composable tool environments built on affordable model transformation technology.

It is quite remarkable to observe the intellectual similarity among MIC, AOP, GenVoca, and IP. Although emerging in quite different communities, answering diverse and different needs, and using very different techniques, they still advocate a core set of new concepts in software and system development: (1) extension of the currently dominant hierarchical, modular composition to multiple-aspect composition, (2) the extensive use and support for domain specific abstractions, and (3) the fundamental role of generative programming and generators in all phases of the system development. In addition, there is a strong trend in the component-based software development and middleware community toward the increased use of modeling and model-based composition in system building [4][35]. It will be interesting to observe whether or not these different approaches will have the strength and will reinforce each other enough to redefine programming during the next decade.

Acknowledgments. The authors would like to thank Don Batory, Gregor Kiczales, Doug Schmidt, and Charles Simonyi for stimulating discussions on the topic. This work was supported by the DARPA MoBIES contract, F30602-00-1-0580.

References

- [1] J. Sztipanovits and G. Karsai: “Model-Integrated Computing,” *IEEE Computer*, April, 1997 (1997) 110–112
- [2] Sztipanovits, J., Karsai, G.: “Embedded Software: Opportunities and Challenges”, in *Embedded Software*, Lecture Notes in Computer Science (LNCS 2211), pp. 403–415, Springer, 2001
- [3] Kiczales, G., Lamping, J., Lopes, C.V., Maeda, C., Mendhekar, A., Murphy, G.: “Aspect-Oriented Programming,” ECOOP’97, , LNCS 1241, Springer. (1997)
- [4] Simonyi, C.: “Intentional Programming: Asymptotic Fun?” Position Paper, SDP Workshop Vanderbilt University December 13–14, 2001. <http://isis.vanderbilt.edu/sdp>

- [5] Batory, D., Geraci, B.J.: "Compositon, Validation and Subjectivity in GenVoca Generators," IEEE Transactions on SE, pp. 67–82, February, 1997.
- [6] Kopetz, H.: *Real-Time Systems: Design Principles for Distributed Embedded Applications* Kluwer, 1997.
- [7] Harrison T., Schmidt, D.C., Levine, D.L.: "The Design and Performance of a Real-time CORBA Event Service," Proceedings of the OOPSLA Conference, October, 1997.
- [8] Sangiovanni-Vincentelli, A.: "Defining Platform-based Design," EEDesign, February, 2002
- [9] P. Hudak: Keynote address at the Usenix DSL Conference, 1997.
<http://www.cs.yale.edu/HTML/YALE/CS/HyPlans/hudak-paul/hudak-dir/dsl/index.htm>
- [10] T. Clark, A. Evans, S. Kent, P. Sammut: "The MMF Approach to Engineering Object-Oriented Design Languages," Workshop on Language Descriptions, Tools and Applications (LDTA2001), April, 2001.
- [11] OMG Unified Modeling Language Specification, Version 1.3. June, 1999
(<http://www.rational.com/media/uml/>).
- [12] CDIF Meta Model documentation. <http://www.metamodel.com/cdif-metamodel.html>
- [13] Object Constraint Language Specification, ver. 1.1, Rational Software Corporation, et al., Sept. 1997. (1997).
- [14] Karsai G., Nordstrom G., Ledecz A., Sztipanovits J.: Specifying Graphical Modeling Systems Using Constraint-based Metamodels, IEEE Symposium on Computer Aided Control System Design, Conference CD-Rom, Anchorage, Alaska, September 25, 2000.
- [15] Generic Modeling Environment (GME 2000) Documentation
<http://www.isis.vanderbilt.edu/projects/gme/Doc.html>
- [16] Girault, A., Lee, B., Lee, E.A.: "Hierarchical Finite State Machines with Multiple Concurrency Models," Technical Memorandum UCB/ERL M97/57, Berkeley, Aug, 17, 1997.
- [17] Ledecz A., Nordstrom G., Karsai G., Volgyesi P., Maroti M.: "On Metamodel Composition," IEEE CCA 2001, CD-Rom, Mexico City, Mexico, September 5, 2001.
- [18] "Sifakis, J.: "Modeling Real-Time Systems – Challenges and Work Directions," EMSOFT 2001, LNCS 2211, Springer. (2001) 373–389
- [19] Lee, E.A., Xiong, Y.: "System-Level Types for Component-Based Design," EMSOFT 2001, LNCS 2211, Springer. (2001) 237–253
- [20] Butts, K., Bostic, D., Chutinan, A., Cook, J., Milam, B., Wand, Y.: "Usage Scenarios for an Automated Model Compiler," EMSOFT 2001, LNCS 2211, Springer. (2001) 66-79
- [21] Neema, S., "Design Space Representation and Management for Embedded Systems Synthesis," Technical Report, ISIS-01-203, February 2001. http://www.isis.vanderbilt.edu/publications/archive/Neema_S_2_0_2003_Design_Spa.pdf
- [22] Rice, M.D., Seidman, S.B.: "A formal model for module interconnection languages," Software Engineering, IEEE Transactions on , Volume: 20 Issue: 1 , Jan. 1994 pp: 88–101
- [23] Lee, E.A. and Sangiovanni-Vincentelli, A: "A Denotational Framework for Comparing Models of Computations," Technical Memorandum, UCB/ERL M97/11, January 30, 1997.
- [24] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns*, Addison-Wesley, 1995.
- [25] Porter, A., Sztipanovits, J. (Ed.): *New Visions for Software Design and Productivity: Research and Applications*. Workshop Report of the Interagency Working Group for Information Technology Research and Development (ITRD) Software Design and Productivity (SDP) Coordinating Group. Vanderbilt University December 13–14, 2001.
<http://isis.vanderbilt.edu/sdp>
- [26] AspectJ: <http://aspectj.org>
- [27] Batory,D., Lofaso, B.,Smaragdakis,Y.: "JTS: Tools for Implementing Domain-Specific Languages," 5th International Conference on Software Reuse, Victoria, Canada, June 1998.
- [28] Czarniecki, K., Eisenecker, U.W.: *Generative Programming*, Addison-Wesley, 2000

- [29] Karsai G., Gray J.: Component Generation Technology for Semantic Tool Integration, Proceedings of the IEEE Aerospace 2000, CD-Rom Reference 10.0303, Big Sky, MT, March, 2000.
- [30] Lieberherr, K.: "Adaptive Object-Oriented Software", Brooks/Cole Pub Co, 1995.
- [31] Rozenberg, G. (ed.), "Handbook on Graph Grammars and Computing by Graph Transformation: Foundations", Vol. 1–2. *World Scientific*, Singapore, 1997
- [32] Tihamer Levendovszky, Gabor Karsai, Miklos Maroti, Akos Ledeczki, Hassan Charaf: Model Reuse with Metamodel-Based Transformations. ICSR 2002: 166–178
- [33] Heiner, G.: Automotive applications, Proceedings of the Joint Workshop on Advanced Real-time Systems, Austrian Research Centers, Vienna, March 26, 2001
- [34] Object Management Group: "Model Driven Architecture" <http://www.omg.org/mda/>
- [35] Gokhale, A. Schmidt, D.C., Natarajan, B., and Nanbor Wang: "Applying Model-Integrated Computing to Component Middleware and Enterprise Applications," *The Communications of the ACM* special issue on Enterprise Components, Service and Business Rules, Vol. 45, No. 10, October, 2002
- [36] The PVS Specification and Verification Systems, <http://pvs.csl.sri.com/>
- [37] Schmidt, D., Stal, M., Rohnert, H., Buschman, F.: Pattern-Oriented Software Architecture, Wiley, 2000

Self Reflection for Adaptive Programming

Giuseppe Attardi and Antonio Cisternino

Dipartimento di Informatica
corso Italia 40, I-56125 Pisa, Italy
{attardi,cisterni}@di.unipi.it

Abstract. Applications in an evolving computing environment should be designed to cope with varying data. Object-oriented programming, polymorphisms and parametric types often do not provide the required flexibility, which can be achieved with the use of metadata attached or extracted from objects by means of reflection. We present a general mechanism to support reflection in C++, exploiting template metaprogramming techniques. Metaclass objects are present at runtime and can be inspected to access information about objects, in particular about their fields. Metaobjects describing fields can be extended, through a mechanism similar to custom attributes in C#. The mechanism is self-reflective, allowing for metaclass objects to be described in turn. This allows storing and retrieving metaclasses from external storage and allows programs to understand and manipulate objects built by other programs. We present two case studies of the technique: building configuration objects and creating object-oriented interfaces to relational database tables.

1 Introduction

Modern applications often involve an open and evolving computing environment and are highly dynamic: neither the elements participating in the architecture nor the objects being handled are fixed or known at implementation time. A pervasive computing environment consists of elements of varying number (nodes, clients, servers), of varying type (devices) and handles data items of varying types. It is not surprising that interpreted scripting languages, such as JavaScript, Python, Perl, PHP, VBScript have become so popular, since they adapt more easily to changes in structures (additive extensions may even require no change at all).

Traditional applications instead rely on data structures designed before hand and fixed at compile time: for instance data base applications rely on a well-defined database schema. The code of the application is written around such data and each component of the application knows what to expect as input and what to produce as output. Programming languages embrace and support this approach by providing compile time analysis and type checking. Database systems rely on data of known structure even if a degree of flexibility is possible through the use of a declarative query language that interprets the database schema.

However, in order to integrate heterogeneous data sources, database middleware systems, such as database gateways or mediators, are needed. Such middleware systems rely on developers to port and manually install to all sites in the system the

application-specific data types and operators necessary to implement the global schema used by the system. Since new applications and data are added to the systems in the course of time, the global schema must be changed to reflect them. This solution is impractical and does not scale well. Consider for instance the case of a database to which new geographic data are added. Processing queries on the new data might require for instance the ability to perform equality tests on geographic locations, which the original system does not know how to handle.

This approach can be called the *structured approach*, where data definition precedes applications. Data and applications are so intertwined that any change in data structures requires a change in the application.

The *semi structured approach* instead exploits self describing data, whose meaning is interpreted at run time by each application. For instance XML data is described by means of an XSD schema and programs can understand them from information in the schema. Web Services are described in WSDL notation [3], which provides information about the types of data and the methods exposed by the service.

The most typical approach to handling semi structured systems is to use interpreters: one for the data definition language and one for the programming language. For instance XSD schemas can be read from Perl. In some cases the two languages may coincide: e.g. Lisp expressions for Lisp, Python initializers for Python. The AnyFramework [28] provides a semantic data representation mechanism for objects in C++, quite similar to the structures used in Python, where objects are made of key-value pairs. This form of representation provides a high degree of flexibility, but lacks essential features like methods and the efficiency of native objects.

While object-oriented systems might seem helpful in providing extensibility, this can only occur within the range of a predefined hierarchy: unfortunately molding new classes into existing inheritance hierarchies is often difficult or impossible and it is subject to limitations in expressiveness and performance due to the need of numerous explicit cast operations. Parametric types or templates provide a better alternative solution.

The solution we advocate here is based on using reflection and annotating objects with metadata. While other solutions require an embedded or special purpose language, reflection does not involve external tools from the programming language, it relieves the application programmer from low level interface coding, and it enables a variety of domain optimization by the component developer.

Unfortunately reflection is not generally available in most programming languages: notable exceptions are Java, C# [27], CLOS [2], Smalltalk, as well as scripting languages like Perl and Python.

A few proposals have been made to extend C++ with support for reflection. In [17, 18, 19] keywords are added to the language for specifying the location of meta-information. A preprocessor generates C++ source code containing the appropriate classes that reflects program's types. In [16] a similar approach is presented which avoids the use of special keywords but still uses a preprocessor.

OpenC++ instead extends the C++ compiler providing support for reflection during program compilation [20].

A form of introspection is provided in XParam [24], a serialization framework for C++ that allows passing objects between applications. Introspection is achieved with programmer assistance: the programmer must supply "registration code" that describes the relevant parts of the classes dealt with XParam.

We present a technique for implementing reflection in C++ by means of template metaprogramming, which exploits the ability to execute code at compile time and to access type information. This relieves the programmer from supplying detailed introspection information. The technique does not involve a separate preprocessor, uses only standard C++ features and works with any recent C++ compiler.

Providing the ability to dynamically add functions or methods to objects within our approach is more difficult, and three solutions are possible: providing an interpreter on each active node of the architecture; exploiting a common virtual machine or an intermediate language; exploiting code generation techniques.

While reflection is normally available and used by programs at run time, reflection support may also be provided to metaprograms at compile time [1]. This allows generating a more efficient version of the program or library, optimized to the specific parameters in the application [15].

The technique presented here enriches objects with metadata, some automatically generated and some supplied by the programmer, which can then be extracted and exploited to perform application specific tasks.

We present two case studies of the use of reflection: building configurable objects and building an object interface to a relational database table. The latter example illustrates the use of custom attributes: objects to be stored in the table are annotated with custom information, expressing special properties, like the type of indexing or the size for a database column.

2 The Reflection Framework

We present a framework that provides full reflection for C++, can deal with both simple and aggregate types (reference and pointer types) and supports both introspection and intercession. The framework requires minimal assistance from the programmer in supplying metadata and provides the following features:

- *Self reflection*: metadata are represented by C++ objects which are themselves described through reflection. This is essential for producing self describing objects to be communicated or stored and retrieved, even by different programs.
- *Custom attributes*: a feature similar to C# custom attributes [27] that allows programmers to extend the metadata associated with types, fields, methods and other elements of the reflection framework.
- *Exact runtime type* information is available for all types.
- *Extensible basic types* (variable grain reflection): the basic types upon which reflection is built are not limited to primitive types and can defined within the framework.
- *Serialization*: an automatic serialization schema is available for C++ types annotated with metadata.

Reflection is under programmer control: for instance metadata need only be supplied for those types involved in reflection, provided that they form a closed set with respect to aggregation and inheritance relations. Our reflection model also satisfies the following properties:

- Preservation of type identity
- Metadata descriptors are isomorphic to the type system of the language.

Preserving *type identity* means that given two objects o_1 and o_2 of the same type, then $typeof(o_1) = typeof(o_2)$, where *typeof* is the reflective operator that returns the descriptor of the type of its argument. This is essential in particular for the serialization of objects.

The solution presented here is more general than the one discussed in [1], where reflection was designed to support serialization of flat object structures and objects could only be read back from a program that included the same class definitions. The new schema allows serializing the metaclasses themselves together with the objects. Hence another program can read and access such objects through the mechanisms of reflection, without knowing their types in advance. This is useful for instance for building interpreters: e.g. an SQL interpreter that must be capable of interpreting data from any database.

3 Expressing Metadata

Reflection involves creating a metaclass object containing metadata information about a class. Metadata annotations are supplied by the programmer within the class definition itself, using macros that trigger template metaprogramming to perform their tasks. The following is an example of a class definition enriched with meta-information:

```
class DocInfo {
    char const*   name;
    char const*   title;
    int           date;

    META(DocInfo,
        (FIELD(name, (MaxLength(256),
                     IndexType(Index::primary))),
         FIELD(title, MaxLength(2048)),
         FIELD(date, IndexType(Index::key))));
};
```

META is a macro that exploits template metaprogramming for creating the metaclass for a class. The macro FIELD is used to specify which of the member variables are to be represented in the metaclass and to define custom attributes for them.

Custom attributes [1] in this example are used to supply properties of fields for use in a database interface. The custom attribute MaxLength(256) specifies 256 as the maximum length for the field; IndexType(Index::primary) designates the field as the primary key. For the field name both attributes are specified by means of an overloaded comma operator. In SQL such properties would be expressed like this:

```
create table DocInfo (
    name          varchar(256),
    title         varchar(2048),
    date          INT,
    PRIMARY KEY (name)
);
```

Supplying such annotations is the only effort required to a programmer for obtaining reflection facilities on a certain class. For instance the type for each attribute needs not to be stated, as required for instance in Xparam, since it is deduced from the class definition, using template metaprogramming.

4 MetaObjects

Introspection and intercession capabilities are made available to C++ programs through metaclass objects present at runtime.

The mechanism used to create meta-objects exploits C++ template metaprogramming, driving the compiler to compute the appropriate metadata.

4.1 C++ Template Metaprogramming

The *template* mechanism of C++ supports generic programming by allowing defining parameterized classes and functions. Templates together with other C++ features form a Turing-complete, compile-time sublanguage of C++. C++ can thus be considered a two-level language [5] since programs may contain both static code, evaluated at compile time, and dynamic code, executed at runtime. Template meta-programs [5] are the part of a C++ source that is executed during compilation. A meta-program can access information about types not generally available to ordinary programs – with the exception of the limited facilities provided by the Run Time Type Identification (RTTI) [6].

Template metaprogramming exploits the computation performed by the type checker in order to execute code at compile time. This technique is used mostly for code selection and code generation at compile time. Specific applications are code configuration, especially in libraries, and code optimization [5]. Partial evaluation [23] is a general technique for program optimization, which produces code specialized for a particular combination of the arguments to a function. Partial evaluation is a proper complement to generic programming since it enables developing generic libraries specialized for each type used in an application.

4.2 Reflecting Attributes

A class is described by an instance of class `MetaClass`, declared by means of the macro `META`. For the example of class `DocInfo` it produces the following code:

```
class DocInfo {
    ...
    typedef DocInfo _CLASS_;
    static vector<Field> _createFields() {
        return (createField("name", &((_CLASS_*)0)->attr1,
                           (MaxLength(2048),
                            IndexType(IndexType::unique))),
                ...
        );
    }
}
```

```
virtual MetaClass* _getMetaClass() {
    return &DocInfo::_metaClass; }
static MetaClass _metaClass;
};
```

Method `_createFields()` builds a vector with the description of the fields of the class. In the generated code, an overloaded operator, `()` is used for adding an element to a vector of `Attribute*`.

Function `createField()` creates an object of class `Field` using template metaprogramming:

```
template <class T>
inline Field createField(char const *name, T* attr,
                        vector<Attribute*> customAttributes)
{
    MetaClass* mc = TypeOf<T>::get();
    return Field(mc, name, (int)attr, customAttributes);
}
```

Method `_getMetaClass()` retrieves the metaclass of an object at runtime. It is a virtual method so that it returns the metaclass of the currently referred object, not the one for the type of the referring expression. This method is only available for `struct` and `class` types: for simple types or other type constructors the meta-function `TypeOf<T>` is used, as defined later.

Before the metaclass can be used, it must be initialized, using the macro `REGISTER(DocInfo)`, which calls its constructor with the class name and its vector of fields:

```
MetaClass
DocInfo::_metaClass("DocInfo", DocInfo::_createFields());
```

Custom attributes, similar to those present in C# [27], can be created by adding items in the vector of attributes for a field. In the example they are instances of class `MaxLength` and `IndexType`.

4.3 Creating Instances of Metaclass Objects

A metaclass provides sufficient information to manipulate the state of objects of the corresponding class, however we cannot make real instances of a class just from a metaclass object: the full layout of the object including the *vtable* and methods is hard to reproduce, and moreover dynamic loading and invocation of methods fields remains a problem. For the time being we allow the creation of surrogate objects, made with similar fields as the originals, which can be accessed and modified. For instance, given the metaclass for `DocInfo`, we can create a surrogate by means of class `AnyObject`:

```
AnyObject any(DocInfo::_metaClass);
any.field<char*>(0) = "value for attribute name";
any.field<int>(2) = 5;
```

and we can then access and modify its individual fields.

5 Full Self Reflection

The mechanism described so far allows providing objects with metaclasses. Using this mechanism we showed [1] how to build an interface to a relational database table: a row of a table is represented as a flat object, consisting of basic types; metaobjects provided support for serialization so that objects could be stored and fetched from the table.

The following problems still remained to be addressed:

- ☐ Dealing with composite objects
- ☐ Dealing with basic types non defined by classes
- ☐ Achieving self-reflection, i.e. allowing the metaobjects to be reflectively described, so that they themselves can be analyzed. Reflection on metaobjects allows achieving completely self-describing objects, which can be serialized, transferred and reinterpreted without prior knowledge of them.

Supporting *full self reflection* requires creating a closed set of reflective types. A *reflective type* is a type whose structure can be exposed through reflection, accessing its metatype. A set of reflective types R is *closed* if for each type t in R the metatype of t and of any constituent type of t belong to R .

We discuss how to construct such a closed set of reflective types and how extensive coverage we can achieve of all C++ types.

The C++ type system [22] includes either fundamental or compound types. Fundamental types include the *void type*, *integral types* (bool, char, int, etc.) and *floating types* (float, double, long double). We consider compound types constructed in the following ways¹:

$T[]$	arrays of objects of type T
T^*	pointers to void or objects of type T
$T\&$	references to objects of type T
<i>enumerations</i>	a set of named constant values
<i>unions</i>	containing objects of different types at different times
<i>classes</i>	containing a sequence of objects and a set of types, enumerations and functions for manipulating those objects
$C<T_1, \dots, T_n>$	a template type, obtained by instantiating type C with parameter types T_1, \dots, T_n .

The first step in building a closed set of reflective types is to provide metaclasses for all fundamental types. We showed earlier how to build metaclasses for class types.

Arrays unfortunately are not a fully-fledged type in C++; they are a low level concept, closely related to the pointer concept: array types for instance cannot be used in template patterns. Therefore we had to resort to use vectors instead of arrays, in particular for building metaclasses.

Metaclasses for pointer types are generated automatically as needed.

Metaclasses are built as classes, so they can be directly described, provided we have a metaclass for `std::vector`, the class used for representing multiple fields and attributes. `Std::vector` is part of the STL, hence it is not a primitive type, but it

¹ At the moment we do not handle functions and pointers to members.

is defined as a template class. However to avoid modifying a class from the standard library, we hand code the construction of metaclasses for vectors.

5.1 Accessing Metaclass Objects

Given a set of reflective types, a metaclass can be accessed in two ways: either from an object or from the type.

The first mechanism gets resolved at run time through dynamic binding and applies only to instances of a reflective class: it consists in invoking method `_getMetaClass()`, as discussed earlier.

The second mechanism gets resolved at compile time and provides access to the metaclass for any type, either fundamental or composite, through the meta-function `TypeOf<T>`. The actual metaclass is obtained as `TypeOf<T>::get()`.

In reflective classes, the metaclass is stored as static field of the class. For these types `TypeOf<T>` is defined as follows:

```
template <class T>
struct TypeOf {
    MetaClass* get() { return &T::_metaClass; }
};
```

For fundamental types a full specialization of the template `TypeOf<T>` is used, for instance:

```
template <>
struct TypeOf<int> {
    MetaClass* get() { return &_metaClass; }
    static MetaClass _metaClass;
};
TypeOf<int>::_metaClass("int", FundamentalType, sizeof(int));
```

Type constructors such as pointers and references are dealt by pattern matching through partial specialization and by generating the corresponding metaclasses with a special constructor for `MetaClass`:

```
template <class T>
struct TypeOf<T*> {
    MetaClass* get() { return &_metaClass; }
    static MetaClass _metaClass;
};
template <class T>
MetaClass
    TypeOf<T*>::_metaClass(TypeOf<T>::get(), PointerType);
```

As mention earlier, we need metaclasses for vectors in order to bootstrap self-reflection, hence a specialization for `std::vector` is present:

```
template <class T>
struct TypeOf<std::vector<T> > {
    MetaClass* get() { return &_metaClass; }
    static MetaClass _metaClass;
};
```

```
template <class T>
MetaClass
    TypeOf<std::vector<T> >::_metaClass (TypeOf<T>::get(),
                                           VectorType);
```

In practice vectors are handled as if they were a basic type. Other types could be dealt like this, effectively extending the set of basic types onto which reflection is built, hence allowing the programmer to control the granularity of types describing the reflection system. In general providing the meta-function `TypeOf<T>` turns type `T` into a basic reflective type.

Enumerations and unions can be handled through a metaclass constructor similar to the `META` macro used for `class` and `struct` types.

6 Metaclass Objects and Bootstrapping

Metaclass objects, i.e. instances of class `MetaClass`, are used to describe types. Various parts of a class are represented by other meta-objects. For instance, the classes for describing attributes, methods and constructors are respectively: `Field`, `Method` and `Constructor`.

Various aspects of objects are described through attributes. For instance class `Field` contains the offset of the field in the memory layout of an object. In addition to predefined attributes a mechanism is available for adding information to metadata: the programmer can define a class, derived from class `Attribute`, to represent the desired information, and use instances of such class to supply the appropriate metadata for the field.

A metaclass should itself be described in terms of another metaclass, its meta-metaclass. The reflection system must be capable of self-reflection avoiding an infinite regression. Only one meta-metaclass exists: it corresponds to class `MetaClass` and is described by an instance of itself.

The only requirement for bootstrapping the self-reflection is that all metaclasses used for describing metaclasses be known and that their metaclass is defined in turn by means of the macro `META`. Beyond the fundamental types, it is only necessary to hand-code the metaclass for `std::vector`.

7 Introspection and Intercession

Reflection support involves two aspects: introspection and intercession. Our reflection system allows introspection: a program may inspect the structures of its own types. The programmer may also add metadata and access them through metaclasses.

Intercession is partially supported in our scheme. A programmer may use the metaclass to access and modify the values of class member variables. Given an object the program may call `_getMetaClass()` and obtain information about fields from their `Field` descriptors. In particular the offset of a class member variable in the memory layout of the object was computed and stored in such descriptor by `createField()`.

Dynamic method invocation is harder to handle: C++ method invocation is compiler and machine dependent. The `Method` class exposes a method named `Invoke` that takes one parameter as input: a vector of pointers. When the method is called a method call should be performed on the specified parameters. However C++ does not provide a portable mean to build a C argument list and call a function on it: a notable attempt is the *avcall* mechanism [26]. The only portable solution is to perform actual method invocation through a case statement on the number of the arguments and dispatching to the code performing a method call with such number of arguments.

A second problem is which type to use for the vector of arguments in the method object. Since the C++ type hierarchy does not provide a common root the only reasonable choice would be to use the `void*` type. This solution only supports invocation of methods with arguments whose size is less or equal to the size of pointers.

8 Customized Serialization

Serialization must be performed in such a way as to preserve identity of types: for instance if an object whose class is already available is being loaded, no metaclass for it must be generated, but the current one should be used.

The system therefore maintains a *metaclass registry* that associates to a fully qualified type name a pointer to the corresponding metaclass, for all currently loaded classes. In dynamic type loading schemes this would help preserving the identity of the metaclass.

Serialization of objects is accomplished by means of reflection, examining information stored in metaclasses and involves two steps:

1. writing the associated class information
2. writing the data associated with the object.

Since class descriptions are provided by metaclasses, the first step involves identifying the class of the object and serializing it. This is a recursive process that ends when the metaclass is one of the built-in metaclasses or the metaclass has been already serialized. This is achieved by maintaining a mapping M from instances and classes to handles, where a handle is just an ID assigned to each object. Serialization of object data (including metaobjects) is done by:

1. checking whether the object has been already serialized, in this case its *ID* is written; otherwise
2. assigning it a new *ID*, registering the *ID* in the mapping M , and serializing each of its field.

The process of deserialization uses the current metaclass dictionary extending it with the metaclasses for the objects being read. The process is as follows:

1. The table of types is loaded and a new table that maps type IDs into metaclass pointers (or null if the type is not linked into the reading program)

2. Metaclasses are deserialized and the ones not yet loaded are loaded into a table (for these types only the metaclass will be available). The location in memory of these classes is stored in the mapping table
3. Objects are deserialized according the metaclass map (preserving the identity of types).

Bootstrapping the serialization mechanism requires that metaclasses be available for all fundamental types as well for all metaclass object types (`MetaClass`, `Field`, `Attribute`, etc.).

In cases when the classes of objects being read are known and loaded, an effective optimization is to skip all metadata information and just store object data. This is the solution described in [1].

When deserializing an object whose class is not available, we make use of the class `AnyObject`. The metaclass for the object is read and supplied to the `AnyObject` constructor for creating a surrogate of the object. This mechanism overcomes the lack in C++ of a mechanism for dynamic loading of classes.

9 Case Study: Configuration Objects

In [21], Eric S. Raymond illustrates the benefits of Python, showing the use of *metaclass hacking* in order to manage configuration files for his *fetchmail* utility. Typically configuration files are free-format text files, edited manually with a text editor. As the configuration becomes more complex, it is convenient to provide a GUI tool to visualize and update the configuration. The basic idea for the solution is to define classes representing configuration objects: a configuration object is dumped to a file in the format of a Python initializer, like this:

```
fetchmailrc = {
    'poll_interval':300,
    'syslog':FALSE,
    # List of server entries
    'servers': [
    # Entry for site 'imap.ccil.org':
    {
        "pollname":"imap.ccil.org",
        "protocol":"IMAP",
        'port':0,
    }
    ... ]
... }
```

Simply evaluating the expression read back from the file would assign to a variable a dictionary object, made of key-value pairs. Turning this object into a linked tree of objects requires a program that knows the structure of the objects involved and uses “that knowledge to grovel through the initializer creating matching objects”, from their constituent key-value pairs. The straightforward solution is to code this program by hand, but this has the drawback that the program would have to be changed whenever new class members are added over time as the configuration language grew new features. An alternative solution is to exploit metaclass programming.

A solution based on C++ metaprograms is the following:

```
class fetchmailrc {
    int                poll_interval;
    bool               syslog;
    vector<server>      servers;

    META(fetchmail,
          (FIELD(poll_interval, 0),
           FIELD(syslog, 0),
           FIELD(servers, 0)));
};

class server {
    char*              pollname;
    char*              protocol;
    int                port;

    META(server,
          (FIELD(pollname, MaxLength(128)),
           FIELD(protocol, MaxLength(128)),
           FIELD(port, 0)));
};
```

Storing and reading configuration objects into files is accomplished by code like this:

```
PersistentList<fetchmailrc> config(rc_file);
config.write(myConfig);           // stores myConfig into rc_file
...
Cursor<fetchmailrc> cursor(config);
myConfig = cursor.get();          // reads back a fetchmailrc
object from file
```

With respect to Python, the reflective C++ solution provides both reading and writing of objects and is also more efficient since it avoids the creation of an intermediate generic representation of data as sets of key-value pairs.

Both the *fetchmail* program and the graphic utility for editing configuration files may use this mechanism for manipulating configuration objects. For viewing and modifying such objects, each class has a method that pops up a GUI edit panel.

10 Case Study: A Relational Object Table

A relational table consists of rows of objects of the same kind. A generic object oriented interface library to a relational table must be capable of handling objects of any class, to be stored in table rows. Therefore the library must know the structure of the objects in order to perform serialization when storing them. In a generic library such information about objects is obtained from the type parameters supplied to it.

For fetching or updating objects from a table, a traditional database interface library like ADO or ODBC provides methods for accessing the individual fields of a row: the programmer must know and specify the type of each field being accessed

and he is also responsible of storing values of the correct type into each field of an object. Table schema definition and table usage are independent operations, of which the compiler is totally unaware: when dealing with instructions performing operations on a table, the compiler has not available any type information on the table.

Exploiting reflection instead, we can build an interface library that provides a higher-level interface to programmers, relieving them from the burden of reconstructing an object fetched from a table or supplying detailed information about the class of the object.

The interface, inspired by GigaBase [29], allows storing and retrieving objects from disk. The library exploits metadata to serialize the objects into the storage provided by Berkeley Database [30]. The library handles objects of any reflective class. In particular a reflective class may include objects of other reflective classes or references to such objects. In the former case the included object is stored within the same table, while in the latter case a separate relational table is used. The library provides several kind of indexing facilities on fields of an object, including inverted indexing or fulltext indexing.

The template class `Table<T>` implements a table for objects of class `T`. Through custom attributes supplied within class `T` various storage properties for each field can be expressed. For instance, class `DocInfo` presented in section 3 shows the use of some custom attributes: the maximum length allowed for a field and the kind of indexing required for it.

This interface has been used in implementing IXE, a fully featured, high performance class library for building customized, full-text search engines. The ability of specialize the library for a given type allows reducing overheads in accessing data at runtime with a corresponding boost in performances.

A program can load the data into the table as follows:

```
Table<DocInfo> table(table_file);
DocInfo aDocInfo(...);
table.insert(aDocInfo);
```

A query on the table can be performed as follows:

```
Query query(query_string);
QueryCursor<DocInfo> cursor(table, query);
while (cursor.MoveNext()) {
    DocInfo docInfo = cursor.get();
    // use docInfo
}
```

Differently from traditional interfaces to database systems [31, 32, 33], the cursor here returns a real object, built from a database row data using reflection. The cursor is capable of accepting complex boolean queries, involving full-text searches on full-text columns and other typical SQL conditions on other columns.

11 Comparison with Other Systems

The reflection support for C++ has been designed to rely only on standard language features, without requiring external tools, such as pre or post processors. Nevertheless

it provides most of the capabilities of reflection supported by frameworks such as Java or Microsoft .NET and C#.

Custom metadata are similar to custom attributes in C# [27]: they can be used to annotate classes with application specific attributes. Minimal burden is required on the programmer for specifying metadata: both Java and C# generate such data through the compiler.

The introspection facilities of reflection are similar to those in Java and C#. Intercession is more limited in our solution since dynamic method invocation and dynamic loading of types are not supported. Dynamic loading cannot be implemented in standard C++ and is quite compiler and of the operating system dependent.

A restricted form of dynamic method invocation might be provided for methods that take only pointers parameters.

The mechanism of dynamic loading of libraries could be exploited for implementing a dynamic class loading facility for C++. The main issue here is to guarantee that each metaclass gets registered only once. This requirement may be hard to fulfill when a class depends on other classes linked in external libraries.

Access restrictions to member variables are neither enforced nor enforceable in C++. Private members are more vulnerable though since information about them is exposed in their metaclass.

12 Conclusions

We presented a general technique based on template metaprogramming for providing reflection in C++. Metaprogramming allows accessing type information from the compiler, which is normally unavailable with other techniques based on source-to-source transformations.

We addressed the problem of representing metadata information as well, so that it can be stored together with data, and programs can read it back and understand the data, without prior built-in knowledge of their structure.

Full self-reflection has been achieved for a closed set of reflective types that covers most of the C++ types.

General-purpose serialization is achieved by accessing and examining metadata. Special versions of serialization achieve quite higher performance than the mechanisms present in Java or C#, as shown by the solution used in the IXE library.

We illustrated the use reflection in building configuration objects and in the implementation of a generic component for storing objects in a relational table. The component can be specialized to any class of objects. Such component has been used in developing the IXE search engine library. The IXE library has proven effective in building several customized search engines and has superior performance to similar commercial products.

References

1. Attardi, G., Cisternino, A.: Reflection support by means of template metaprogramming. Proceedings of Third International Conference on Generative and Component-Based Software Engineering, Lecture Notes in Computer Science, Vol. 2186. Springer-Verlag, Berlin (2001) 178–187
2. Attardi, G.: Metaobject Programming in CLOS. In Object-Oriented Programming – The CLOS Perspective, A. Paepke (ed.), Cambridge, MA (1993)
3. W3C: Web Services Description Language. <http://www.w3.org/TR/wsdl>.
4. Gabriel, R.G., Bobrow, D.G., White, J.L.: CLOS in Context – The Shape of the Design Space. In Object Oriented Programming – The CLOS perspective. The MIT Press, Cambridge, MA (1993) 29–61
5. Czarnecki, K., Eisenacker, U.W.: Generative Programming – Methods, Tools, and Applications. Addison Wesley, Reading, MA (2000)
6. Stroustrup, B.: The Design and Evolution of C++. Addison Wesley, Reading, MA (1994)
7. Sleepycat Software, *The Berkeley Database*, <http://www.sleepycat.com>.
8. MySQL, *MySQL*, <http://www.mysql.com>.
9. Microsoft, ActiveX Data Objects, <http://msdn.microsoft.com/library/psdk/dasdk/adot9elu.htm>.
10. K.A. Knizhnik, The GigaBASE Object-Relational database system, <http://www.ispras.ru/~knizhnik>.
11. Sun Microsystems, Java Database Connectivity, <http://java.sun.com/>.
12. Petter Urkedal, *Tools for Template Metaprogramming*, <http://matfys.lth.se/~petter/src/more/metad>.
13. R. Sunderraman, Oracle8™ Programming: a primer, Addison-Wesley, MA, (2000)
14. Plauger, P. J. , Stepanov, A., Lee, M., Musser, D.: The Standard Template Library. Prentice-Hall (2000)
15. Malenfant, J., Jaques, M., Demers, F.N.: A tutorial on behavioral reflection and its implementation. Proceedings of the Reflection 96 Conference, Kiczales, G. (ed.), San Francisco, California, USA, April (1996) 1–20
16. Chuang, T.R., Kuo, Y. S., Wang, C.M.: Non-Intrusive Object Introspection in C++: Architecture and Application. Proceedings of the 20th Int. Conference on Software Engineering, IEEE Computer Society Press (1998) 312–321
17. Madany, P.W., Islam, N., Kougiouris, P., Campbell, R.H.: Reification and reflection in C++: An operating systems perspective. Technical Report UIUCDCS-R-92-1736, Dept. of Computer Science, University of Illinois at Urbana-Champaign, March (1992)
18. Ishikawa, Y., Hori, A., Sato, M., Matsuda, M., Nolte, J., Tezuka, H., Konaka, H., Maeda, M., Kubota, K.: Design and Implementation of metalevel architecture in C++ – MPC++ approach. Proceedings of the Reflection 96 Conference, Kiczales, G. (ed.), San Francisco, California, USA, April (1996) 153–166
19. Gowing, B., Cahill, V.: Meta-Object Protocols for C++: The Iguana Approach. Proc. Reflection '96, San Francisco, California (1996) 137–152
20. Chiba, S.: A metaobject protocol for C++. Conference Proceedings of Object-Oriented Programming Systems, Languages and Applications, ACM Press (1995) 285–299
21. Raymond, E.S.: Why Python? Linux Journal, May (2001) <http://www.linuxjournal.com/article.php?sid=3882>.
22. Ellis, M. A., Stroustrup, B.: The Annotated C++ Reference Manual. Addison-Wesley, MA (1990)
23. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall International, International Series in Computer Science, June (1993)
24. Maor, R., Brand, M.: XParam - General-Purpose Object Serialization Framework for C++. <http://xparam.sourceforge.net>.

25. Grosso, W.: Java RMI Serialization. O'Reilly & Associates (2001)
http://www.onjava.com/pub/a/onjava/excerpt/JavaRMI_10.
26. Haible, B.: VACALL Manual. <http://www.mit.edu/afs/sipb/project/clisp/new/clisp-1999-01-08/ffcall/avcall/avcall.html>.
27. Archer, T.: Inside C#. Microsoft Press, Redmond, WA (2001)
28. Mätzel, K.U., Bischofberger, W.: The Any Framework A Pragmatic Approach to Flexibility. Second Usenix Conference on Object-Oriented Technology and Systems, Toronto, June (1996)
29. Knizhnik, K.A.: The GigaBASE Object-Relational database system.
<http://www.ispras.ru/~knizhnik>.
30. Sleepycat Software, The Berkeley Database, <http://www.sleepycat.com>.
31. MySQL, MySQL, <http://www.mysql.com>.
32. Microsoft, ActiveX Data Objects,
<http://msdn.microsoft.com/library/psdk/dasdk/adot9elu.htm>.
33. Sun Microsystems, Java Database Connectivity, <http://java.sun.com>.

DataScript – A Specification and Scripting Language for Binary Data

Godmar Back

Stanford University
gback@stanford.edu

Abstract. DataScript is a language to describe and manipulate binary data formats as types. DataScript consists of two components: a constraint-based specification language that uses DataScript types to describe the physical layout of data and a language binding that provides a simple programming interface to script binary data. A DataScript compiler generates Java libraries that are linked with DataScript scripts. DataScript specifications can be used to describe formats in a programmatic way, eliminating the vagaries and ambiguities often associated with prosaic format descriptions. The libraries generated by the DataScript compiler free the programmer from the tedious task of coding input and output routines. More importantly, they assure correctness and safety by validating both the input read and the output generated. We show examples that demonstrate that DataScript is simple, yet powerful enough to describe many commonly used formats. Similar to how scripting languages such as Perl allow the manipulation of text files, the libraries generated by the DataScript compiler can be used to quickly write scripts that safely manipulate binary files.

1 Introduction

Currently, binary data formats are typically described in English prose in design documents and standards. This method is inefficient, prone to ambiguities and misunderstandings, and does not lend itself to easy reuse and sharing of these metadata specifications between applications. As a result, some applications may produce incompatible output that cannot be read by other applications, or worse, causes other applications to crash. Casual scripting of binary data is also cumbersome. While there are many tools to validate and manipulate high-level textual data formats such as XML, there are no tools for binary formats. Currently, a programmer has to choose a language, convert the prosaic description into language-specific data structures such as C structures; then he must implement input routines to read the data into memory, and output routines to write the data back to external storage. The programmer has to account for alignment and byte-order issues, which are notorious for their potential for bugs when such programs are ported to different architectures.

Even where libraries or tools to read a given file format are available, a programmer has to learn those libraries' APIs first, and there may be limitations

in what they can do. The necessary feature to perform the task at hand may not be provided, because the developers of the tool or library did not foresee a need for it. Where libraries are not available, programmers tend to “whip up” input and output routines that often only provide partial implementations, do not validate their input, or produce nonconforming output that causes problems when it is processed in the future.

Many binary formats include constraints that guarantee the consistency of the data, such as parameter ranges, length constraints, checksums and others. Like the layout of the data itself, these constraints are also typically expressed in prose rather than in a form suitable for automatic translation into programs.

DataScript’s specification language solves these problems by describing binary data formats as types. The specification language has the following properties:

- **Simple and intuitive.** DataScript is designed to be a simple solution to what is conceptually a simple problem. DataScript types are easily readable; we kept the number of abstractions needed to understand a DataScript specification to a minimum.
- **Architecture-independent.** Unlike C header files, DataScript types are described in a architecture-independent way. The developer is responsible for specifying the size and byte order of any and all data elements—since DataScript is a language to describe the physical layout of data, no attempt is made to hide such details from the programmer.
- **Constraint-based.** Each component of a DataScript type can have constraints associated with it. These constraints must be fulfilled if a given piece of binary data corresponds to a type. They can be used to discriminate between types, or they can be used to express invariants that hold if the data is consistent.
- **Declarative.** DataScript types are purely declarative, i.e., the specification of a type does not include any side effects. Purely declarative descriptions allow the same format description to be used for input (parsing binary data) and output (generating binary data according to a given format).
- **Practical.** DataScript provides direct support for commonly used idioms. For instance, some file formats store length and offset parameters in headers. Such dependencies can be directly expressed in DataScript.

2 The DataScript Language

A typical situation in which binary data are read and processed is the loading of Java class files by a class loader in a Java Virtual Machine (JVM) [4]. Java is often used for mobile code, which a virtual machine may load from untrusted sources, hence the class loader must verify the well-formedness of the class file. It must be prepared to handle corrupted or malformed class files gracefully.

The loader reads a class file’s bytes into memory, parses them and creates and populates data structures that correspond to the data stored in the file. This

process involves identifying constants that are stored as multibyte integers in the file, reading parameters such as lengths and offsets, reading arrays of simple and variable-record types, reading flags and indices and checking that they are valid and within range, and the verification of constraints that ensure the internal consistency of a class file.

The set of rules that govern the internal structure and consistency of a class file are described in the DataScript specification shown in Fig. 1. A class file starts with magic integers that describe its version, which are followed by the constant pool, which is a linear array of variable-sized structures of `union` type “ConstantPoolInfo”. A variable record or union type can take on different alternatives, which must be discriminated. In this case, the choice taken depends on the value of a “tag” field contained in each of the options, which include class entries, utf8 (string) entries, and others not shown in the figure.

Fields in a composite DataScript type can have constraints associated with them. For instance, the “magic” constant at the beginning of `ClassFile` must always be equal to 0xCAFEBAFE. However, DataScript allows for the expression of more complex constraints: for instance, the Java virtual machine specification requires that the “super_class” field in a class file is either zero or the index of a constant pool entry with a tag value of “CONSTANT_Class”. In general, a field’s constraints can be an arbitrary boolean predicate. Predicates that are used in multiple places, such as “clazz”, can be defined using the `constraint` keyword. “clazz” uses the `is` operator, which checks whether the union representing the specified constant pool entry represents an external reference to another class.

The DataScript compiler takes this specification and generates a set of Java classes and interfaces that can read and write Java class files. On input, the generated code checks that all constraints are satisfied. When reading union types, it uses constraints to discriminate between the different choices of a union, similar to how a top-down parser uses look-ahead tokens to decide on the next nonterminal during parsing. On output, it performs the same set of checks, which ensures the well-formedness of the generated output.

2.1 DataScript Types

In this section, we provide a more formal description of DataScript’s types. DataScript types are defined recursively as follows. A DataScript type is either

- a primitive type, or
- a set type, which can be either an enumerated type or a bitmask type, or
- a linear array of a type, or
- a composite type, which can either be a record type or variant-record type.

We describe each of these types in turn.

Primitive Types. Primitive types form the basic blocks upon which more complex types can be built. They include bit fields, 8-bit integers (bytes), 16-bit, 32-bit, 64-bit, and 128-bit integers. All primitive types are interpreted as

```

const uint8    CONSTANT_Utf8    = 1;
const uint8    CONSTANT_Class   = 7;      // ...

const uint16   ACC_PUBLIC       = 0x0001;
const uint16   ACC_ABSTRACT    = 0x0400;  // ...

ClassFile {
    uint32      magic = 0xCAFEBAFE;
    uint16      minor_version = 3;
    uint16      major_version = 45;
    uint16      cp_count;
    ConstantPoolInfo constant_pool[1..cp_count];

    bitmask uint16 ClassFlags {
        ACC_PUBLIC, ACC_FINAL, ACC_ABSTRACT, ACC_INTERFACE, ACC_SUPER
    } access_flags;
    uint16      this_class : clazz(this_class);
    uint16      super_class : super_class == 0 || clazz(super_class);
    uint16      interfaces_count;
    {
        uint16 ifidx : clazz(ifidx);
    } interfaces[interfaces_count];
    uint16      fields_count;
    FieldInfo   fields[fields_count];
    uint16      methods_count;
    MethodInfo  methods[methods_count];
    uint16      attributes_count;
    AttributeInfo attributes[attributes_count];

    constraint clazz(uint16 x) { constant_pool[x] is cp_class; }
};

union ConstantPoolInfo {
    {
        uint8 tag = CONSTANT_Class;
        uint16 name_index;
    } cp_class;

    Utf8 {
        uint8 tag = CONSTANT_Utf8;
        uint16 length;
        uint8 bytes[length];
    } cp_utf8;
    // ... other choices ...
};

```

Fig. 1. DataScript description of Java class files. The complete description is 237 lines long.

integers, which can be either signed or unsigned. The size and signedness is encoded in the keyword, i.e., `uint32` is a 32-bit unsigned integer, while `int16` is a signed 16-bit integer.

The byte order for multibyte integers can be specified by an optional attribute prefix, which can be either `little` or `big`. By default, a primitive type inherits the byte order attribute of its enclosing composite type. If the enclosing composite type has no byte order attribute, big endian or network byte order is assumed (as is the case for Java class files).

Set Type. DataScript supports two types of set types: enumerated types (`enum`) and bitmask types (`bitmask`). A set type forms a subset of an underlying primitive type, which specifies the byte order and signedness used to store and interpret its values. This example was taken from the DataScript specification of IEEE 802.11b [2] packets:

```
enum bit:4 ControlPacketType {
    CTRL_PS_POLL      = 1010b,      // Power Save (PS) Poll
    CTRL_RTS          = 1011b, ... // Request To Send
};
```

Composite Types. DataScript uses a C-like style to describe composite types. DataScript's composite types are similar to C structs and unions, and the intuitions match. However, unlike in C, the `struct` keyword can be omitted, because it describes the default composition mode. Unlike C structs and unions, however, there is no implementation-dependent alignment, packing, or padding. As such, the size of a union may vary depending on which choice is taken.

DataScript unions are always discriminated. The first choice in a union whose constraints match the input is taken, hence the textual ordering in the specification is important. DataScript does not require that the constraints associated with the different choices of a union be disjoint, which allows for a default or fall-back case that is often used in specifications to allow room for future extensions.

Composite types can be lexically nested, which provides each type with its own lexical scope. Scoping provides a namespace for each type, it does not provide encapsulation. If a type is referenced from outside its confining type, a full qualification can be used, such as `ClassFile.ClassFlags`. Anonymous composite types can be used to define a fields whose types are never referenced, such as “cp_class” in Fig. 1.

Array Types. DataScript arrays are linear arrays that use integer indices. Like record types, there is no alignment or padding between elements. An array's lower and upper bounds are given as two expressions: [`lower`.. `upper`], which includes `lower`, but excludes `upper`. If `lower` is omitted, it defaults to zero.

The expressions used to specify the lower and upper bounds must be constructed such that their values can be determined at the time the array is read.

Typically, an array's length will depend in some form on fields that were previously read. Hence, frequently used idioms such as a length field followed by an array of the specified length can be easily expressed.

2.2 DataScript Constraints

Each field in a composite type can have a constraint associated with it, which is specified as an arbitrary boolean predicate separated from the field name by a colon. Constant fields form a special case of constraints that can be written like an initializer in C and Java. Thus, the field definition `uint32 magic = 0xCAFEBAFE` is a shorthand for `uint32 magic : magic == 0xCAFEBAFE`.

Constraints are used in three cases: first, constraints that are attached to fields in a union type are used to discriminate that type. Second, constraints are used to express consistency requirements of a record type's fields. In a record type, a specification designer can choose the field to which a predicate is attached, with the restriction that a predicate can only refer to fields that lexically occur *before* the field to which the constraint is attached. This restriction was introduced to facilitate straightforward translation and to encourage readable DataScript specifications.

Third, constraints can be attached to array elements to limit the number of items in an array when its length is not known beforehand. Arrays with unspecified length grow until an element's constraints would be violated or until there is no more input.

Because DataScript constraints cannot have side effects, the DataScript compiler is free to generate code that evaluates constraints as soon as their constituents are known; the generated code can also evaluate a constraint predicate multiple times if needed.

DataScript predicates are boolean expressions. DataScript borrows its set of operators, associativity, and precedence rules from Java and C in the hope that doing so will facilitate adoption by tapping into programmers' existing skill sets. It includes arithmetic operations such as integer addition, multiplication, etc.; logical and arithmetic bit operations, relational and comparison operators, and so on.

DataScript requires additional operators. In many formats, the interpretation of a piece of data depends on descriptors that occurred earlier in the file. If the descriptor was modeled as a union type, the constraints used to discriminate the data will need to inquire which alternative of the descriptor was chosen. The `is` operator is a boolean operator that can be used to test whether an instance of a union type took on a specified choice.

Another common situation is for a field to express the length of an array not as the number of elements, but as its total size in bytes. The `sizeof` operator can be used to compute the number of elements. `sizeof` returns a field's or type's size in bytes, similar to the same-named operator in C. If `sizeof` is applied to a field, it will evaluate to the actual size of that field after it is read. If `sizeof` is applied to a composite type, the type must have a fixed size. A record type has a fixed size if all its elements have fixed sizes; a union type's size is fixed if

```

ClassFile {
    constraint compare_utf8(uint16 idx, string str) {
        constant_pool[idx] is cp_utf8;
        constant_pool[idx].cp_utf8.bytes.compare_to_string(str);
    } // rest omitted ...
}

union AttributeInfo {
    Code {
        uint16 name_idx: ClassFile.compare_utf8(name_idx, "Code");
        uint32 length;
        uint16 max_stack;
        uint16 max_locals;
        uint32 code_length;
        uint8 code[code_length];
        uint16 exception_table_length;
        {
            uint16 start_pc;
            uint16 end_pc;
            uint16 handler_pc;
            uint16 catch_t: catch_t == 0 || ClassFile.clazz(catch_t);
        } exception_table[exception_table_length];
        uint16 attributes_count;
        AttributeInfo attributes[attributes_count];
    } code : sizeof(code) == sizeof(code.name_idx)
        + sizeof(code.length) + code.length;

    LineNumberTable {
        uint16 name_idx: ClassFile.compare_utf8(name_idx, "LineNumberTable");
        uint32 length;
        uint16 line_number_table_length;
        {
            uint16 start_pc: start_pc < Code_attribute.code_length;
            uint16 line_number;
        } line_number_table[line_number_table_length];
    } lnr_table: sizeof(lnr_table) == sizeof(lnr_table.length)
        + sizeof(lnr_table.name_idx) + lnr_table.length;
};

```

Fig. 2. Recursive description of attributes in Java class files.

all its elements have the same size, and an array type has a fixed size if all its elements have the same size and the number of elements is a constant. Figure 2 provides an example for the `sizeof` operator: the “length” field in a `Code` type must contain the length of the attribute, not including the number of bytes used for the “name_idx” and the “length” fields.

DataScript provides a `forall` operator that can be attached to arrays, and whose use is necessary when constraints cannot be attached to an array element because they refer to the array itself, whose definition occurs lexically after the element’s definition. For instance, the following construction can be used to ensure that an array is sorted:

```
uint32 a[1] : forall i in a : i < 1-1 ? a[i] < a[i+1] : true;
```

The index check is necessary because any expression that causes out-of-bound array accesses will evaluate to false.

Constraints that are used in multiple places can be defined once and reused as part of predicates. A constraint defined in this way can only be applied to instances that are contained inside an instance of the type defining the constraint. For example, the constraint “compare_utf8” shown in Figure 2 can be used from inside `AttributeInfo` if the instance is contained in a `ClassFile` instance shown in Figure 1. Because both `ClassFile` and `AttributeInfo` are top-level definitions, this decision cannot be made statically, but must be made at run time. If at run time the instance is not contained, the expression containing the reference will evaluate to false. The DataScript compiler will reject type references if the specification provides no possible way for the referring type to be contained inside the referred type. This property can be checked by considering reachability in the graph that results from the “contained-in” relation between types.

2.3 Type Parameters

Each composite type can have a list of parameters associated with it. Parameters can be arbitrary DataScript types, either primitive or composite. Consider the example in Figure 3, which shows how type parameters can be used to implement a padding type that can be inserted to ensure that alignment constraints are not violated. `Padding` takes two parameters: the size of the unaligned structure that

```
Padding(uint32 size, uint32 align) {
    uint8 padding[(align - (size % align)) % align];
};

SomeDirectoryEntry {
    uint16 namesize;
    uint8 fname[namesize];
    Padding(sizeof fname, 4) pad;           // align to 4-byte boundary
    ...
};
```

Fig. 3. Use of type parameters for padding.

needs alignment, and the boundary at which it should be aligned. A variable-length byte array whose length is computed from these parameters serves to insert appropriate padding.

2.4 Labels

Many data formats are not sequential, but contain headers or directories whose entries store offsets into the file at which other structures start. To support such constructs, DataScript allows fields to be labeled. Unlike C labels, DataScript labels are integer expressions. When they are evaluated at run time, the resulting value is used as an offset at which the labeled field is located in the data.

Figure 4 shows part of the DataScript specification for an ELF [9] object file. ELF object files contain a number of sections with symbolic and linking information necessary to load a program or library into memory and to execute it. These sections are described in a section header table. However, the location of the section header table is not fixed; instead, the ELF specification states that the offset at which the section header table starts is given by the field “e_shoff”, which is stored at a known offset in the ELF header at the beginning of the file. The DataScript specification expresses this by labeling the “sh_header” field with the expression “e_shoff”.

Labels are offsets relative to the beginning of the record or union type in whose scope they occur. They can only depend on fields that occur lexically before them. On some occasions, it is necessary for a label to refer to a different structure. In these circumstances, a label base can be specified before the label that refers to the structure relative to which the label is to be interpreted. Figure 5 provides an example.

```
Elf32_File {
  Elf_Identification {
    uint32 magic = 0x7f454c46;
    ...
  } e_ident;

  ...

  uint32 e_shoff;
  uint32 e_flags;
  uint16 e_ehsize;
  uint16 e_phentsize;
  uint16 e_phnum;
  uint16 e_shentsize;
  uint16 e_shnum;
  uint16 e_shtrndx;

  ...
  e_shoff:
  Elf32_SectionHeader {
    uint32 sh_name;
    uint32 sh_type;
    uint32 sh_flags;
    uint32 sh_addr;
    uint32 sh_offset;
    uint32 sh_size;
    uint32 sh_link;
    uint32 sh_info;
    uint32 sh_addralign;
    uint32 sh_entsize;
  } hdrs[e_shnum];

  ElfSection(hdrs[s$index]) s[e_shnum];
};
```

Fig. 4. Use of labels in ELF object file description.

```

ElfSection (Elf32_File.Elf32_SectionHeader h) {
Elf32_File:: h.sh_offset:
    union {
        { } null : h.sh_type == SHT_NULL;
        StringTable(h) strtab : h.sh_type == SHT_STRTAB;
        SymbolTable(h) symtab : h.sh_type == SHT_SYMTAB;
        SymbolTable(h) dynsym : h.sh_type == SHT_DYNSYM;
        RelocationTable(h) rel : h.sh_type == SHT_REL;
        ...
    } section;
};

SymbolTable(Elf32_File.Elf32_SectionHeader h) {
    Elf32_Sym {
        uint32    st_name;
        uint32    st_value;
        uint32    st_size;
        uint8     st_info;
        uint8     st_other;
        uint16    st_shndx;
    } entry[h.sh_size / sizeof Elf32_Sym];
};

```

Fig. 5. Use of a label base to express offsets that are relative to another structure.

Figures 4 and 5 also demonstrate a more involved use of type parameters. Since each section is described by its header, the definition for the `ElfSection` type is parameterized by the `Elf32.SectionHeader` type. The type definition then uses fields such as “sh_offset” and “sh_type” to compute the offset and to discriminate the section type. The parameter “h” passed to the type can be passed on to other types. For instance, the `SymbolTable` type uses the “sh_size” field to compute the number of entries in the table if the section contains a symbol table.

3 Java Language Binding

We chose Java for our reference implementation because its object-oriented nature provides for a straightforward mapping of DataScript types to Java classes, because it provides architecture-independent primitive types, and because it has rich runtime support for reading from and writing to data streams. The DataScript compiler generates a Java class for each composite type, which includes accessor methods for each field. Each DataScript type’s class provides a constructor that can construct an instance from a seekable input stream.

3.1 Prototype Implementation

Our current prototype code generator does not perform any optimizations. When parsing input, it simply sets a mark in the input stream and reads the necessary numbers of bytes for the next field, performs byte-order swapping if necessary, and initializes the field. It then checks the constraints associated with that field. If the constraint is violated, an exception is thrown. A catch handler resets the stream to the earlier set mark and aborts parsing the structure. For union types, a try/catch block is created for each choice. If a constraint for one choice is violated, the next choice is tried after the stream is reset until either a matching choice is found or there are no choices left, in which case the constructor is aborted.

3.2 Using Visitors

The DataScript code generator also creates the necessary interfaces to implement the visitor design pattern [1]. This generated visitor interface contains methods for each composite and primitive type in the specification. A default visitor implementation is generated that provides a depth-first traversal of a type. For a record type, it visits all fields in sequence. For a union type, it visits only the incarnated choice.

4 Related Work

PACKETTYPES. McCann and Chandra present a packet specification language called PACKETTYPES that serves as a type system for network packet formats [6]. It provides features that are similar to DataScript's: composite types, union types, and constraints. In addition, it provides a way for a type to overlay fields in other types, as is needed for protocol composition. PACKETTYPES's focus is on performing one operation well: matching a network packet received from the network to a specified protocol. For this reason, it generates stubs in C. Unlike DataScript, PACKETTYPES does not address generating data, it also does not provide an visitor interface for writing scripts the way DataScript's Java binding does. PACKETTYPES also does not support different byte orders, labels for non-sequential types such as ELF, nor type parameters.

OMG's Meta Object Facility. The OMG's Meta Object Facility or MOF [8] is a framework for standardizing the exchange of metadata. It defines a framework with which to describe metadata; a Java language binding [3] in the form of interfaces has been proposed recently. However, it does not address the physical layout of the actual data nor does it provide a way to automatically generate code to read and write data.

XML. XML is an extensible markup language used to store structured information as marked-up text while providing support for syntactic and limited semantic checking. In a world in which everybody used XML to exchange data, there would be no need for DataScript, yet this scenario is unlikely not only because of the large number of existing non-XML formats, but because XML has shortcomings, such as lack of space efficiency.

Interface Definition Languages (IDL). IDL languages such as RPC-XDR [5] or CORBA-IDL [7] provide a way to exchange data between applications. They typically consist of a IDL part that is independent of the physical layout of the data, and a physical layer specification such as IIOP that describes the physical layout of the data. By contrast, DataScript does not prescribe how applications lay out data, it merely provides a language to describe how they do.

5 Conclusion

We have introduced DataScript, a specification language for describing physical data layouts, and we presented a Java language binding for DataScript specifications. DataScript specifications allow casual scripting with data whose physical layout have been described in DataScript. DataScript is a simple-to-understand constraint-based language. We believe that such a specification language should become a standard part of the tool and skill set of developers, in the same way that compiler-compilers such as yacc or the use of regular expressions in languages such as Perl have become tools that are in everyday use.

References

1. GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.
2. IEEE STANDARDS BOARD. *IEEE Std 802.11-1997*, 1997.
3. IYENGAR, S. *Java Metadata Interface (JMI Specification)*, Nov. 2001. Version 1.0 Update Public Review Draft Specification.
4. LINDHOLM, T., AND YELLIN, F. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Jan. 1997.
5. LYON, B. Sun remote procedure call specification. Tech. rep., Sun Microsystems, 1984.
6. MCCANN, P. J., AND CHANDRA, S. Packet Types: abstract specification of network protocol messages. In *Proceedings of the SIGCOMM '00 Conf.* (Stockholm, Sweden, Aug. 2000).
7. OBJECT MANAGEMENT GROUP. *The Common Object Request Broker: Architecture and Specification*, June 1999. Revision 2.3. OMG Document formal/98-12-01. Part of the CORBA 2.3 specification.
8. OBJECT MANAGEMENT GROUP. *Meta Object Facility (MOF) Specification*, Mar. 2000. Revision 1.3 OMG Document 00-04-03.
9. TOOLS INTERFACE STANDARDS (TIS) COMMITTEE. *ELF: Executable and Linkable Format: Portable Formats Specification, Version 1.2*, May 1995.

Memoization in Type-Directed Partial Evaluation

Vincent Balat¹ and Olivier Danvy²

¹ PPS, Université Paris VII – Denis Diderot
Case 7014, 2 place Jussieu, F-75251 Paris Cedex 05, France
(balat@pps.jussieu.fr)

² BRICS***
Department of Computer Science, University of Aarhus
Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark
(danvy@brics.dk)

Abstract. We use a code generator—type-directed partial evaluation—to verify conversions between isomorphic types, or more precisely to verify that a composite function is the identity function at some complicated type. A typed functional language such as ML provides a natural support to express the functions and type-directed partial evaluation provides a convenient setting to obtain the normal form of their composition. However, off-the-shelf type-directed partial evaluation turns out to yield gigantic normal forms.

We identify that this gigantism is due to redundancies, and that these redundancies originate in the handling of sums, which uses delimited continuations. We successfully eliminate these redundancies by extending type-directed partial evaluation with memoization capabilities. The result only works for pure functional programs, but it provides an unexpected use of code generation and it yields orders-of-magnitude improvements both in time and in space for type isomorphisms.

1 Introduction

1.1 Background: Reduction-Based vs. Reduction-Free Normalization

Say that we consider binary trees modulo associativity. Binary trees are easily coded as a data type in Standard ML [36,37,45]:

```
datatype 'a bt = LEAF of 'a
              | NODE of 'a bt * 'a bt
```

(In this declaration, `bt` is the name of the data type. It is parameterized with a type variable `'a` to express that the data type is polymorphic: we can represent

*** Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation.

trees of integers, trees of reals, trees of lists, etc. `LEAF` and `NODE` are binary-tree constructors.)

The following conversion rule (written with an infix \Leftrightarrow) defines associativity:

$$\forall t_1, t_2, t_3 : 'a \text{ bt}, \text{NODE } (t_1, \text{NODE } (t_2, t_3)) \Leftrightarrow \text{NODE } (\text{NODE } (t_1, t_2), t_3).$$

Two binary trees are equal modulo associativity if they can be converted to each other using \Leftrightarrow .

How do we represent binary trees modulo associativity in practice? One option is to use the data type above and test for equality using the conversion rule. A more efficient version, however, exists. It is based on the idea of *orienting* the conversion rule into a rewriting rule. For example, we could orient it as follows:

$$\forall t_1, t_2, t_3 : 'a \text{ bt}, \text{NODE } (t_1, \text{NODE } (t_2, t_3)) \Leftarrow \text{NODE } (\text{NODE } (t_1, t_2), t_3).$$

This rewriting rule is nice because repeatedly applying it (1) terminates and (2) yields a unique normal form. (A normal form here is a binary tree for which the rewriting rule cannot be applied.) Representing a binary tree modulo associativity is thus best done with its normal form because it is more efficient to test for equality.

In the present case, the data type of binary trees in normal form can be coded as follows:

```
datatype 'a bt_nf = LEAF_nf of 'a
                  | NODE_nf of 'a * 'a bt_nf
```

The constructor `NODE_nf` guarantees that the rewriting rule cannot be applied.

Since this data type is isomorphic to the data type of non-empty lists, we can represent normalized binary trees as ML lists. The question then is how to normalize binary trees. In ML terms, this amounts to writing a function

```
normalize : 'a bt -> 'a list
```

that flattens its argument into a normal form.

The traditional, reduction-based, approach is to traverse the source tree and repeatedly apply the rewrite rule:

```
fun reduction_based_normalize (LEAF x)
  = x :: nil
| reduction_based_normalize (NODE (LEAF x, t))
  = x :: (reduction_based_normalize t)
| reduction_based_normalize (NODE (NODE (t1, t2), t3))
  = reduction_based_normalize (NODE (t1, NODE (t2, t3)))
```

An alternative, reduction-free, approach to normalization, however, exists: it amounts to interpreting the binary tree in a non-standard model and inverting this interpretation. In the present case, we choose the non-standard model to be the function space

```
'a list -> 'a list
```

We map leaves into a function that adds an element to its argument, we map nodes into function composition, and we invert the interpretation function by applying values to the empty list:

```
fun reduction_free_normalize t
= let fun eval (LEAF x)
      = (fn a => x :: a)
      | eval (NODE (t1, t2))
      = (eval t1) o (eval t2)
  fun reify value
    = value nil
  in reify (eval t) end
```

This seemingly daunting function can be simplified as follows: rather than returning a function, the argument of this function can be specified as one more argument to `eval`, and `reify` can be inlined:

```
fun reduction_free_normalize_simpler t
= let fun eval (LEAF x) a
      = x :: a
      | eval (NODE (t1, t2)) a
      = eval t1 (eval t2 a)
  in eval t nil end
```

The result is the familiar `flatten` function with an accumulator.

This way of normalizing binary trees is said to be reduction free because it does not explicitly apply the rewriting rule. Because it normalizes a term by inverting an evaluation function (into a non-standard model), reduction-free normalization is also referred to as *normalization by evaluation*. The flattening example above is folklore in the normalization-by-evaluation community.

Normalization by evaluation has been variously studied in logic, proof theory, and category theory [2,3,8,9,10,13] and in partial evaluation [14,16]. Type-directed partial evaluation, which we present next, has been investigated both practically [5,15,17,18,29,31,38] and foundationally [24,25,47].

1.2 Type-Directed Partial Evaluation

Type-directed partial evaluation is a practical instance of normalization by evaluation and is used for specializing functional programs. The evaluation function it inverts is the standard evaluation of functional programs. Consequently, a type-directed partial evaluator maps *values* to a textual representation of their normal form, in contrast to a traditional syntax-directed partial evaluator, which maps the *textual representation* of a source program to the textual representation of the corresponding specialized program.

In the present work, we consider a pure version of type-directed partial evaluation for ML with the following types (*a* is atomic):

$$t ::= a \mid t_1 \rightarrow t_2 \mid t_1 \times t_2 \mid t_1 + t_2$$

For example, let us consider the following ML function, which exponentiates its argument x by recursively halving its argument n , using the auxiliary function `binary`. Depending on the parity of its argument, `binary` applies `odd` or `even` to each intermediate result. The functions `quot` and `rem` respectively compute the quotient and the remainder of two integers; they are found in the `Int` library. The opportunity for specialization here is that the first argument of `exponentiate` (and thus the argument of `binary`) is known statically.

```
fun exponentiate n (odd, even) x
  = let fun binary 0
        = x
        | binary n
        = let val r = binary (Int.quot (n, 2))
          in if Int.rem (n, 2) = 0
             then even r
             else odd r
          end
    in binary n end
```

A syntax-directed partial evaluator maps the textual representation of `exponentiate 20` to the textual representation of its specialized version (the overhead of the interpretation of 20 has been completely eliminated):

```
fn (p1, p2) => fn x3 => let val r4 = p1 x3
                        val r5 = p2 r4
                        val r6 = p1 r5
                        val r7 = p2 r6
                    in p2 r7 end
```

In contrast, a type-directed partial evaluator maps the value of `exponentiate 20` (together with a representation of its type) to the textual representation of its specialized version. (In fact, the residual function above is the actual output of our type-directed partial evaluator.)

1.3 Motivation

Recently, we have realized that a proof-theoretical application of type-directed partial evaluation was affected by the size and redundancy of the generated code. Unsurprisingly, we have diagnosed the problem to arise because of sums, which are handled with continuations and therefore duplicate contexts.

For example, at type $(a \rightarrow b) \rightarrow (c \rightarrow a) \rightarrow c \rightarrow b$, the term `fn f => fn g => fn x => f (g x)` is normalized into the following residual term:

```
fn x0 => fn x1 => fn x2 => let val r3 = x1 x2
                        in x0 r3 end
```

At type $(a \rightarrow b) \rightarrow (bool \rightarrow a) \rightarrow bool \rightarrow b$, however, it is normalized into the following other residual term, where the application of `x0` occurs in both conditional branches:¹

¹ The boolean type is a trivial sum.

```

fn x0 => fn x1 => fn x2 => if x2
  then let val r3 = x1 true
        in x0 r3 end
  else let val r5 = x1 false
        in x0 r5 end

```

In both cases, the residual term is in normal form: it contains no function applications or conditional expressions that could be simplified away. It is also fully eta-expanded.

Normalization at boolean type is handled by duplicating contexts (the application of `x0` in the example just above). This duplication is known to yield redundant residual terms in a pure setting. For example, normalizing `fn f => fn g => fn x => f (g x) (g x)` at type $(bool \rightarrow bool \rightarrow a) \rightarrow (b \rightarrow bool) \rightarrow b \rightarrow a$ yields the following residual term:

```

fn x0 => fn x1 => fn x2 => let val r3 = x1 x2
  in if r3
    then let val r4 = x0 true
          val r5 = x1 x2
          in if r5
            then r4 true
            else r4 false
          end
    else let val r8 = x0 false
          val r9 = x1 x2
          in if r9
            then r8 true
            else r8 false
          end
    end
  end
end

```

This residual term is redundant in a pure setting because in each branch of the outer conditional expression, we know the result of `x1 x2` and therefore there is no need to recompute it and test it. The residual term could thus be simplified into the following one:

```

fn x0 => fn x1 => fn x2 => let val r3 = x1 x2
  in if r3
    then let val r4 = x0 true
          in r4 true end
    else let val r6 = x0 false
          in r6 false end
    end
  end
end

```

In the proof-theoretic setting considered here (see Section 2), such a simplification is crucial.

1.4 Contribution and Overview

We solve the above redundancy by introducing a memoization mechanism in type-directed partial evaluation.

The rest of this article is organized as follows: Section 2 describes the proof-theoretical setting of our work; Section 3 reviews type-directed partial evaluation; and Section 4 presents the memoization mechanism.

2 Type Isomorphisms

Two data types are said to be isomorphic if it is possible to convert data between them without loss of information. More formally, two types σ and τ are isomorphic if there exists a function f of type $\sigma \rightarrow \tau$ and a function g of type $\tau \rightarrow \sigma$, such that $f \circ g$ is the identity function over τ and $g \circ f$ is the identity function over σ .

Type isomorphisms provide a way not to worry about unessential details in the representation of data. They are used in functional programming to provide a means to search functions by types [20,21,22,39,40,41,42] and to match modules by specifications [7,19].

Searching for converters between particularly complex isomorphic types raises the problem of normalizing composite functions, in order to verify whether they are the identity function or not. Normalization by evaluation provides an elegant solution: we simply write the functions in ML and we residualize their composition.

The work presented in this paper takes its inspiration from a recent joint work by Balat, Di Cosmo, and Fiore [6]. This work addresses the relations between the problem of type isomorphisms and a well-known arithmetical problem, called “Tarski’s high school algebra problem” [23].

2.1 Tarski’s High School Algebra Problem

Tarski asked whether the arithmetic identities taught in high school (namely: commutativity, associativity, distributivity and rules for the neutral elements and exponentiation) are complete to prove all the equations that are valid for the natural numbers. His student Martin answered this question affirmatively under the condition that one restricts the language of arithmetic expressions to the operations of product and exponentiation and the constant 1.

For arithmetic expressions with sum, product, exponentiation, and the constant 1, however, the answer is negative, witness an equation due to Wilkie that holds true in \mathbb{N} but that is not provable with the usual arithmetic identities [46]. Furthermore, Gurevič has shown that in that case, equalities are not finitely axiomatizable [30]. To this end, he exhibited an infinite number of equalities in \mathbb{N} such that for every finite set of axioms, one of them can be shown not to follow.

2.2 Tarski’s High School Algebra Problem, Type-Theoretically

If one replaces sums, product, and exponentiation respectively by the sum, product, and arrow type constructors, and if one replaces the constants 0 and 1 respectively by the empty and unit types, one can restate Tarski’s question as one

about the isomorphisms between types built with these constructors. For types built without sum and empty types, Soloviev, and then Bruce, Di Cosmo, and Longo have shown that exactly the same axioms are obtained [11,43].

Continuing the parallel with arithmetic, Balat, Di Cosmo, and Fiore have studied the case of isomorphisms of types with empty and sum types [6]. They have generalized Gurevič's equations for the case of equalities in \mathbb{N} without constants as follows:

$$(A^u + B_n^u)^v \cdot (C_n^v + D_n^v)^u = (A^v + B_n^v)^u \cdot (C_n^u + D_n^u)^v \quad (n \geq 3 \text{ odd})$$

where

$$A = y + x$$

$$B_n = y^{n-1} + xy^{n-2} + x^2y^{n-3} + \dots + x^{n-2}y + x^{n-1} = \sum_{i=0}^{n-1} x^i y^{n-i-1}$$

$$C_n = y^n + x^n$$

$$D_n = y^{2n-2} + x^2y^{2n-4} + x^4y^{2n-6} \dots + x^{2n-4}y^2 + x^{2n-2} = \sum_{i=0}^{n-1} x^{2i} y^{2n-2i-2}$$

Balat, Di Cosmo, and Fiore have proven that these equalities hold in the world of type isomorphisms as well. They did so by exhibiting a family of functions and their inverses. Figure 1 shows a fragment of one of these functions, written in Standard ML, when $n = 3$. The type of this term fragment is displayed at the top of the figure. It corresponds to $(A^u + B_3^u)^v \cdot \dots \rightarrow (A^v + B_3^v)^u \cdot \dots$, where 'a corresponds to v , 'b corresponds to u , 'c corresponds to y , 'd corresponds to x , and furthermore `sum`, `*`, and `->` are type constructors for sums, products, and functions (i.e., exponentiations).²

For such large and interlaced functions, whether intuited or automatically produced, it is rather daunting to show that composing them with their inverse yields the identity function. A normalization tool that handles sums is needed. In the presence of sums, however, normalization is known to be a non-trivial affair [1], chiefly because of commuting conversions [27]. Type-directed partial evaluation does handle sums, but the redundancy pointed out in Section 1.3 is a major impediment.

3 Type-Directed Partial Evaluation

Type-directed partial evaluation is defined as a pair of functions for each type constructor. The first function, `reify`, maps a value into a representation of its normal form. The second function, `reflect`, maps the representation of a normal form into the corresponding value. Reification and reflection are already well described in the literature [8,10,16,25,28,29,31,38,47] and therefore, rather than

² In ML's type language, the type constructors for products and functions are infix, and the type constructor for sums is postfix.

```

('a -> ('b -> ('c,'d) sum, 'b -> ('d * 'd,('d * 'c,'c * 'c) sum) sum) sum) * ...
->
('b -> ('a -> ('c,'d) sum, 'a -> ('d * 'd,('d * 'c,'c * 'c) sum) sum) sum) * ...

```

```

fn (p1, p2)
=> (fn x3
  => (case p2 x3
    of (LEFT s5)
      => LEFT (fn x7
        => (case p1 x7
          of (LEFT s9)
            => (case s9 x3
              of (LEFT s12)
                => LEFT s12
              | (RIGHT s13)
                => RIGHT s13)
          | (RIGHT s10)
            => (case s5 x7
              of (LEFT (p17, (p19, p20)))
                => (case s10 x3
                  of (LEFT (p24, p25))
                    => RIGHT p25
                  | (RIGHT s23)
                    => (case s23
                      of (LEFT (p28, p29))
                        => LEFT p29
                      | (RIGHT (p30, p31))
                        => RIGHT p20))
                | (RIGHT (p32, (p34, p35)))
                  => (case s10 x3
                    of (LEFT (p39, p40))
                      => LEFT p35
                    | (RIGHT s38)
                      => (case s38
                        of (LEFT (p43, p44))
                          => RIGHT p43
                        | (RIGHT (p45, p46))
                          => LEFT p46))))))
              | (RIGHT s6) => RIGHT (fn x47 => ...)),
  fn x111
  => (case p1 x111
    of (LEFT s113) => LEFT (fn x115 => ...)
    | (RIGHT s114) => RIGHT (fn x179 => ...)))

```

Fig. 1. Isomorphism function for $n = 3$ (fragment)

repeating these descriptions, let us instead focus on the one equation of interest: reflection at sum type.

$$\uparrow_{t_1+t_2} e = \text{shift } \kappa \text{ in } \text{case}(e, x.\text{reset}(\kappa(\text{in}_1(\uparrow_{t_1} x))), y.\text{reset}(\kappa(\text{in}_2(\uparrow_{t_2} y))))$$

where x and y are fresh

The control operator shift abstracts the evaluation context of $\uparrow_{t_1+t_2} e$ and relocates it in each branch of a residual conditional expression. The control operator reset delimits the extent of any subsequent control abstraction in the conditional branches. The effect of this context duplication has been illustrated in Section 1.3.

4 Memoization

4.1 What

Our aim is to avoid dead branches in the residual code by integrating the two following transformations in our type-directed partial evaluator:

$$\begin{aligned} & \text{case}(e, x_1.M_1 \left[\text{case}(e, y_1.N_1, y_2.N_2)/z \right], x_2.M_2) \\ & \longrightarrow \text{case}(e, x_1.M_1 \left[N_1 \left[x_1/y_1 \right]/z \right], x_2.M_2) \\ & \text{case}(e, x_1.M_1, x_2.M_2 \left[\text{case}(e, y_1.N_1, y_2.N_2)/z \right]) \\ & \longrightarrow \text{case}(e, x_1.M_1, x_2.M_2 \left[N_2 \left[x_2/y_2 \right]/z \right]) \end{aligned}$$

These transformations are easily derivable from the η rule for sum types:

$$\text{case}(t, x_1.h(\text{in}_1 x_1), x_2.h(\text{in}_2 x_2)) = h t$$

For example, taking $h = \lambda x. \text{case}(x, x_1.M_1 \left[\text{case}(x, y_1.N_1, y_2.N_2)/z \right], x_2.M_2)$ and β -reducing yields the first transformation.

4.2 How

The residual program is an abstract-syntax tree. This abstract-syntax tree is constructed depth first, left to right. Our key idea is to maintain a global stack accounting for conditional branches in the path from the root of the residual program to the current point of construction.

The global stack can be implemented with a global reference and sequential push and pop operations as the construction proceeds. It seems plausible that the correctness of this state-based version of type-directed partial evaluation can be approached by adding a state monad to Filinski's formalization [25]. We are currently looking into this issue [4].

The stack associates a flag (Left or Right) and a variable to an expression as specified below:

$$\uparrow_{t_1+t_2} e = \begin{cases} \text{in}_1(\uparrow_{t_1} z) & \text{if } e \text{ is globally associated to (Left, } z) \\ \text{in}_2(\uparrow_{t_2} z) & \text{if } e \text{ is globally associated to (Right, } z) \\ \text{shift } \kappa \text{ in } \underline{\text{case}}(e, & \text{otherwise} \\ \quad \quad \quad x.\text{reset}(\kappa(\text{in}_1(\uparrow_{t_1} x))), & \\ \quad \quad \quad y.\text{reset}(\kappa(\text{in}_2(\uparrow_{t_2} y)))) & \\ \text{where } x \text{ and } y \text{ are fresh} & \end{cases}$$

If e is not associated to anything in the stack, then we associate it to (Left, x) when processing the consequent and to (Right, x) when processing the alternative.

4.3 Application

In the present case, memoization pays off: as illustrated in Figure 2, the output of type-directed partial evaluation is between one and two orders of magnitude smaller, for a residualization time that is also between one and two orders of magnitude smaller. (We also observed that the time ML takes for inferring the types of the isomorphism functions offsets the time taken by type-directed partial evaluation, even in the absence of memoization.)

4.4 Common Sub-expression Elimination

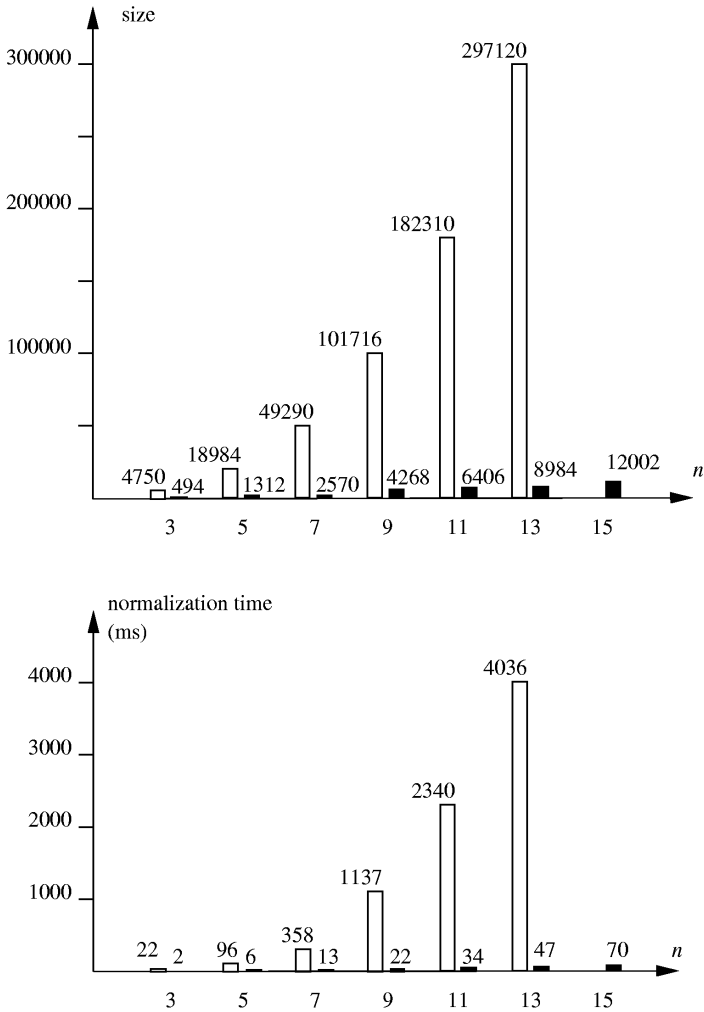
Furthermore, we are now in position to integrate common sub-expression elimination by reflecting at function type into memo functions [32,35]. These memo functions are indexed by the global stack to ensure their coherence, since a reflected function can be applied in conditional branches with distinct lexical scope. (In the absence of sums, the memo functions can be used uniformly.)

To illustrate common sub-expression elimination, let us come back to the last example of Section 1.3, `fn f => fn g => fn x => f (g x) (g x)`. Without memo functions, residualizing it at type $(a \rightarrow a \rightarrow b) \rightarrow (c \rightarrow a) \rightarrow c \rightarrow b$ yields the following residual term where the application `x1 x2` occurs twice:

```
fn x0 => fn x1 => fn x2 => let val r3 = x1 x2
                           val r4 = x0 r3
                           val r5 = x1 x2
                           in r4 r5 end
```

In contrast, memo functions make it possible to obtain the following residual term, where the result `r3` is used twice and thus the application `x1 x2` occurs only once:

```
fn x0 => fn x1 => fn x2 => let val r3 = x1 x2
                           val r4 = x0 r3
                           in r4 r3 end
```



The two graphs visualize the size of the residual abstract syntax trees (the number of their nodes) and their normalization time (in milliseconds, on a 4-processor Sparc station running SunOS 5.7 using SML/NJ Version 110.0.6) for the isomorphism functions described in Section 2 and Figure 1, for $n = 3, 5, 7, 9, 11$, and 13 . The white bars account for standard type-directed partial evaluation, and the black bars account for type-directed partial evaluation with memoization.

Fig. 2. Benchmarks

5 Related Work

Memoisation is a standard component of polyvariant partial evaluators that yield mutually recursive residual programs [12,33,44]. Using a traditional syntax-directed partial evaluator, however, is not a realistic option here because our source programs are higher-order and thus require a frightful number of binding-time improvements.

We are not aware of any similar work on type isomorphisms.

Finally, and as illustrated in Section 1.1, type-directed partial evaluation is only one instance of normalization by evaluation. We are not aware of any other use of memoization in other instances.

6 Conclusion and Issues

We have extended type-directed partial evaluation of pure functional programs with memoization capabilities. Primarily, memoization makes it possible to keep track of the dynamic result of tests in conditional branches, as in Futamura's Generalized Partial Computation [26]. Secondly, memoization makes it possible to integrate a form of common sub-expression elimination in type-directed partial evaluation. Getting back to our initial motivation, memoization makes it practical to use type-directed partial evaluation to verify type isomorphisms in the presence of sums.

Acknowledgments. This article sprang from the two authors's participation to the 30th spring school on theoretical computer science (<http://www.pps.jussieu.fr/~ecole>) held in Agay, France, in March 2002, and it has benefited from discussions with Andrzej Filinski and Samuel Lindley. Thanks are also due to Mads Sig Ager, Peter Thiemann, and the reviewers for helpful comments, and to Henning Korsholm Rohde for automating the generation of the isomorphism functions.

This work is supported by the ESPRIT Working Group APPSEM (<http://www.md.chalmers.se/Cs/Research/Semantics/APPSEM>).

References

1. Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Philip Scott. Normalization by evaluation for typed lambda calculus with coproducts. In Joseph Halpern, editor, *Proceedings of the Sixteenth Annual IEEE Symposium on Logic in Computer Science*, pages 203–210, Boston, Massachusetts, June 2001. IEEE Computer Society Press.
2. Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction-free normalization proof. In David H. Pitt, David E. Rydeheard, and Peter Johnstone, editors, *Category Theory and Computer Science*, number 953 in Lecture Notes in Computer Science, pages 182–199, Cambridge, UK, August 1995. Springer-Verlag.

3. Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Reduction-free normalisation for a polymorphic system. In Edmund M. Clarke, editor, *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science*, pages 98–106, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.
4. Vincent Balat. PhD thesis, PPS, Université Paris VII – Denis Diderot, Paris, France, 2002. Forthcoming.
5. Vincent Balat and Olivier Danvy. Strong normalization by type-directed partial evaluation and run-time code generation. In Xavier Leroy and Atsushi Ohori, editors, *Proceedings of the Second International Workshop on Types in Compilation*, number 1473 in Lecture Notes in Computer Science, pages 240–252, Kyoto, Japan, March 1998. Springer-Verlag.
6. Vincent Balat, Roberto Di Cosmo, and Marcelo Fiore. Remarks on isomorphisms in typed lambda calculi with empty and sum types. In Gordon D. Plotkin, editor, *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science*, Copenhagen, Denmark, July 2002. IEEE Computer Society Press. To appear.
7. Gilles Barthe and Olivier Pons. Type isomorphisms and proof reuse in dependent type theory. In Furio Honsell and Marino Miculan, editors, *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001*, number 2030 in Lecture Notes in Computer Science, pages 57–71, Genova, Italy, April 2001. Springer-Verlag.
8. Ulrich Berger. Program extraction from normalization proofs. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 91–106, Utrecht, The Netherlands, March 1993. Springer-Verlag.
9. Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by evaluation. In Bernhard Möller and John V. Tucker, editors, *Prospects for hardware foundations (NADA)*, number 1546 in Lecture Notes in Computer Science, pages 117–137. Springer-Verlag, 1998.
10. Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ -calculus. In Gilles Kahn, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.
11. Kim Bruce, Roberto Di Cosmo, and Giuseppe Longo. Provable isomorphisms of types. *Mathematical Structures in Computer Science*, 2(2):231–247, 1992.
12. Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.
13. Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7:75–94, 1997.
14. Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, January 1996. ACM Press.
15. Olivier Danvy. Online type-directed partial evaluation. In Masahiko Sato and Yoshihito Toyama, editors, *Proceedings of the Third Fuji International Symposium on Functional and Logic Programming*, pages 271–295, Kyoto, Japan, April 1998. World Scientific. Extended version available as the technical report BRICS RS-97-53.

16. Olivier Danvy. Type-directed partial evaluation. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation – Practice and Theory; Proceedings of the 1998 DIKU Summer School*, number 1706 in Lecture Notes in Computer Science, pages 367–411, Copenhagen, Denmark, July 1998. Springer-Verlag.
17. Olivier Danvy, Morten Rhiger, and Kristoffer Rose. Normalization by evaluation with typed abstract syntax. *Journal of Functional Programming*, 11(6):673–680, 2001.
18. Olivier Danvy and René Vestergaard. Semantics-based compiling: A case study in type-directed partial evaluation. In Kuchen and Swierstra [34], pages 182–197. Extended version available as the technical report BRICS-RS-96-13.
19. David Delahaye, Roberto Di Cosmo, and Benjamin Werner. Recherche dans une bibliothèque de preuves Coq en utilisant le type et modulo isomorphismes. In *PRC/GDR de programmation, Pôle Preuves et Spécifications Algébriques*, November 1997.
20. Roberto Di Cosmo. Type isomorphisms in a type assignment framework. In Andrew W. Appel, editor, *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 200–210, Albuquerque, New Mexico, January 1992. ACM Press.
21. Roberto Di Cosmo. Deciding type isomorphisms in a type assignment framework. *Journal of Functional Programming*, 3(3):485–525, 1993.
22. Roberto Di Cosmo. *Isomorphisms of types: from λ -calculus to information retrieval and language design*. Birkhauser, 1995. ISBN-0-8176-3763-X.
23. John Doner and Alfred Tarski. An extended arithmetic of ordinal numbers. *Fundamenta Mathematica*, 65:95–127, 1969. See also <http://www.pps.jussieu.fr/~dicosmo/Tarski/>.
24. Andrzej Filinski. A semantic account of type-directed partial evaluation. In Gopalan Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming*, number 1702 in Lecture Notes in Computer Science, pages 378–395, Paris, France, September 1999. Springer-Verlag. Extended version available as the technical report BRICS RS-99-17.
25. Andrzej Filinski. Normalization by evaluation for the computational lambda-calculus. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001*, number 2044 in Lecture Notes in Computer Science, pages 151–165, Kraków, Poland, May 2001. Springer-Verlag.
26. Yoshihiko Futamura, Kenroku Nogi, and Akihiko Takano. Essence of generalized partial computation. *Theoretical Computer Science*, 90(1):61–79, 1991.
27. Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.
28. Bernd Grobauer. *Topics in Semantics-based Program Manipulation*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, July 2001. BRICS DS-01-6.
29. Bernd Grobauer and Zhe Yang. The second Futamura projection for type-directed partial evaluation. *Higher-Order and Symbolic Computation*, 14(2/3):173–219, 2001.
30. R. Gurevič. Equational theory of positive numbers with exponentiation. *Proceedings of the American Mathematical Society*, 94(1):135–141, May 1985.

31. Simon Helsen and Peter Thiemann. Two flavors of offline partial evaluation. In Jieh Hsiang and Atsushi Ohori, editors, *Advances in Computing Science - ASIAN'98*, number 1538 in Lecture Notes in Computer Science, pages 188–205, Manila, The Philippines, December 1998. Springer-Verlag.
32. John Hughes. Lazy memo-functions. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 129–146, Nancy, France, September 1985. Springer-Verlag.
33. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, London, UK, 1993. Available online at <http://www.dina.kvl.dk/~sestoft/pebook/>.
34. Herbert Kuchen and Doaitse Swierstra, editors. *Eighth International Symposium on Programming Language Implementation and Logic Programming*, number 1140 in Lecture Notes in Computer Science, Aachen, Germany, September 1996. Springer-Verlag.
35. Donald Michie. ‘Memo’ functions and machine learning. *Nature*, 218:19–22, April 1968.
36. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.
37. Larry C. Paulson. *ML for the Working Programmer (2nd edition)*. Cambridge University Press, 1996.
38. Morten Rhiger. *Higher-Order Program Generation*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, July 2001. BRICS DS-01-4.
39. Mikael Rittri. Retrieving library identifiers by equational matching of types. In Mark E. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction*, number 449 in Lecture Notes in Computer Science, pages 603–617, Kaiserslautern, Germany, July 1990. Springer-Verlag.
40. Mikael Rittri. *Searching program libraries by type and proving compiler correctness by bisimulation*. PhD thesis, University of Göteborg, Göteborg, Sweden, 1990.
41. Mikael Rittri. Using types as search keys in function libraries. *Journal of Functional Programming*, 1(1):71–89, 1991.
42. Colin Runciman and Ian Toyn. Retrieving re-usable software components by polymorphic type. *Journal of Functional Programming*, 1(2):191–211, 1991.
43. Sergei V. Soloviev. The category of finite sets and cartesian closed categories. *Journal of Soviet Mathematics*, 22(3):1387–1400, 1983.
44. Peter Thiemann. Implementing memoization for partial evaluation. In Kuchen and Swierstra [34], pages 198–212.
45. Jeffrey D. Ullman. *Elements of ML Programming (ML 97 edition)*. Prentice-Hall, 1998.
46. Alex J. Wilkie. On exponentiation – a solution to Tarski’s high school algebra problem. *Quaderni di Matematica*, 2001. To appear. Mathematical Institute, University of Oxford (preprint).
47. Zhe Yang. *Language Support for Program Generation: Reasoning, Implementation, and Applications*. PhD thesis, Computer Science Department, New York University, New York, New York, August 2001.

A Protocol Stack Development Tool Using Generative Programming

Michel Barbeau and Francis Bordeleau

School of Computer Science, Carleton University, 1125 Colonel By Drive, Ottawa
(Ontario), Canada K1S 5B6,
{barbeau|francis}@scs.carleton.ca,
<http://www.scs.carleton.ca>

Abstract. Traditional protocol implementation approaches capture the structural aspects of protocols in a common base that can be used accross layers. However, they are usually not very good at capturing the behavioural aspects. Two important implementation problems result, namely, reprogramming similar behavior and configuration of crosscutting concerns. In this paper, we present an approach to solve the problems of reprogramming similar behavior and absence of systematic configuration mechanisms for crosscutting concerns in communication systems. Our approach is based on generative programming, has been implemented in C++ and has been validated with several protocols. We also sketch an approach for run-time reconfigurable protocol stacks.

1 Introduction

Accross layers, protocols have several architectural elements in common. Indeed, they are often based on a common model, which is customized in each protocol. Traditional protocol implementation approaches can capture the structural aspects of protocols in a common base that can be used accross the layers. However, they are usually not very good at capturing the behavioural aspects. As a result, the same behaviour model has to be reprogrammed and debugged from protocol to protocol. We call this problem *reprogramming similar behaviour*.

Each individual protocol implementation framework includes a set of good solutions to concerns that cross all the protocols of an implementation, e.g. message representation and event handling. However, it is actually very hard to combine a good solution for one concern from one framework with a good solution for another concern from another framework. It is unfortunate because it might be exactly the combination that would make the application at hand effective. We call this problem *absence of systematic configuration mechanisms for crosscutting concerns*.

In this paper, we present an approach to solve the problems of reprogramming similar behaviour and configuration of crosscutting concerns in communication systems. Our approach is based on generative programming and has been implemented in C++. It has been validated with several protocols, including a packet radio protocol, a wireless LAN protocol, IP, a dynamic source routing protocol,

a satellite transport protocol and a simple network management protocol. Some of our work is available on-line [1].

Traditional approaches for the implementation of communication systems and candidate approaches are reviewed in Section 2. Our approach based on generative programming is presented in Sections 3 and 4. In Section 5, we discuss a limitation of our current work and how we plan to address it in the future. We conclude with Section 6.

2 Related Work

Protocol implementation approaches can be categorized according to the placement of the code relative to the OS space. There are three approaches: kernel level, user level and a mix of both.

With the kernel-level approach, all the code is placed in the kernel space. This approach is especially efficient for network-level and link-level protocols in which the amount of data and control traffic is large. A lot of work is normally required to port a kernel-level protocol stack from one OS to another. The TCP/IP stack of both Linux [2] and BSD Unix [14] are kernel level.

With the user-level approach, the kernel of the OS is not changed. The protocol stack runs as one or several user processes. On Unix, for example, user-level protocols communicate with the OS through the socket abstraction. The deployment and debugging of communication systems are made easier. The start and interruption of the communication systems are also often easier to manage. It is also easy to port an implementation from one OS to another. This approach may cause, however, performance penalties because protocol data units have to be transferred between the user-level and kernel-level and may not work for cases in which a certain level of control of the OS is required, but not available through system calls. Hence, a mixed approach is often required. The x-kernel framework [6] and NOS TCP/IP stack [13] are both user level. Elements of mobility support in IPv6 have been implemented with a mix approach in [15].

Object-oriented approaches can help to solve the problems of reprogramming similar behaviour and configuration of crosscutting concerns. All the aforementioned implementation approaches are based on traditional programming methodologies, except for x-kernel, which is an object based framework. Use of the object-oriented approach to implement network management applications as resulted to SNMP++ [8]. A pattern language that addresses issues associated with concurrency and networking is presented in [12]. An object-oriented network programming approach is described in [11] and [10].

Other related work is the Ensemble project. It proposes an approach in which stacks of micro-protocols can be assembled to fulfill needs for group communications [3]. There is a support for dynamic adaptation [9].

In this paper, we introduce a novel protocol implementation approach based on generative programming, which can be combined with other approaches to obtain a higher degree of reuse of design and code. To the best of our knowledge,

it is the first attempt to use the generative programming approach, including feature modeling, for developing protocols.

3 Feature Modeling of Protocols

Generative programming [4] is domain engineering and modeling of a family of software as components that can be assembled automatically. Domain engineering consists of three steps: domain analysis (including feature analysis), domain design and domain implementation. This paper is about generative programming for telecommunications protocols. A model is presented consisting of software components that can be assembled to build communication systems, i.e. protocol stacks.

Domain Analysis

According to [6], a family of telecommunications protocol stacks can be modeled using the following abstractions: Session, Protocol, Participant, Map, Message, Event and Uniform Interface.

A Session is the abstraction of an end-point of a communication channel, i.e. a point-to-point channel is terminated at each end by a session. A Protocol is an abstraction that represents specific message formats and communication rules. There is a many-to-many relationship between Session and Protocol; each session is associated with a protocol and each protocol can contain several sessions. For instance, the local end-point of a host-to-host channel is abstracted as an IP session and is embedded and managed by an IP protocol. A session object is dynamically created when a new channel is opened. Protocols and sessions contain together the elements required for the interpretation of messages: coding, decoding and taking decisions. They maintain the local state of the channel: window size, number of next message expected, number of last acknowledged message, etc.

The Participant abstraction represents the identity of an application, a protocol or a session. For example, in the TCP/IP architecture the identity of an application is a port number and an IP address.

The Map abstraction models partial functions. It is used to map external keys made of addresses (e.g. a source IP address and a remote IP address pair) to an internal value (e.g. the session corresponding to the abstraction of that channel).

The general model for a protocol layer is pictured in Figure 1. Each layer is made of a protocol. It is a rather static entity (created at start and persisting until the end). A protocol contains both an active map and a passive map. The active map keeps track of sessions managed by the protocol. Each entry maps a local address and remote address pair (e.g. A1 and A2), that we call an ActiveId, to a pointer to a session which models the entry of a communication channel between a local entity that has address A1 to a remote entity that has address A2. The passive map keeps track of enabled local addresses on which the

protocol is eager to open new channels. With TCP, for instance, the passive map would contain the port numbers on which servers are listening for client-initiated sessions. The establishment of the channel is initiated by a remote entity. Such an enabled address is called a PassiveId. The enabling is normally performed by a higher-level protocol. The number of times this action has been performed is kept in the Ref count field and the Protocol field is a pointer to this higher-level protocol.

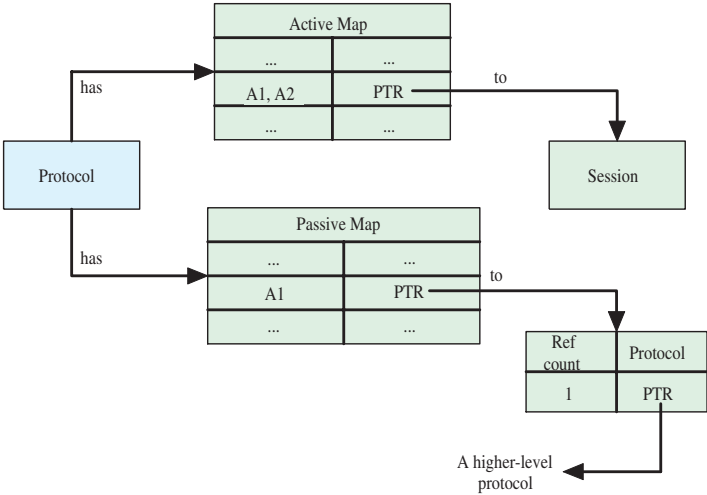


Fig. 1. General model of a protocol layer.

The Message abstraction represents protocol data units. It is a string of bytes of some length. This abstraction is used to pass data up and down in a protocol stack. It provides operations for adding/stripping headers and fragmentation/reassembly of messages.

Scheduling of events, captured by the Event abstraction, is an activity shared by several protocols, e.g. transmission of advertisements and scheduling of re-transmissions.

The notion of Uniform Interface captures the operations common to all protocols and sessions. Interaction with protocols and session can be modelled in a layer independent model using a common collection of operations. For example, every layer provides an operation for opening a channel (xOpen) and operation for pushing data through data channel (xPush). It also makes protocols and session definitions easily substitutable.

Feature Analysis

Feature analysis is the identification of the commonalities and variability among the members of a family software. A feature diagram is a result of feature anal-

ysis. A feature is a property having importance and is a point where choices can be made to define a particular instance of a concept. A feature diagram is a representation that captures the features of a concept. Figures 2 and 3 picture the feature diagrams for the Session and Protocol concepts. The root of each diagram represents a concept and leaf nodes represent its features.

The Session concept has the Message mandatory open feature (the filled circle indicates a mandatory property, open brackets indicate an open property) and the EventManager optional feature (indicated by a non-filled circle). For example, either a single buffer model, as in Linux, or directed acyclic graph of buffers, as in x-kernel, can be chosen for the Message feature. For the EventManager feature, either a delta list model of a timing wheel model of event manager can be selected.

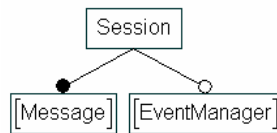


Fig. 2. The Session concept with internal features.

The Protocol concept has the Message, Session, Participant, EventManager, ActiveOpen and PassiveOpen features. The Session feature is optional because the Protocol-Session model does not fit certain cases such as high-level protocols and low-level protocols that abstract hardware devices. A possible choice for the Participant feature is a stack model of identifiers, as in x-kernel, or a linked-list of identifiers. A Protocol variant is defined using a Session variant, when applicable. The choice for the Message feature of the Protocol variant must match the value of the same feature in the Session variant.

The Active/Passive Map (with the accompanying ActiveId/PassiveId) is modelled as a sub feature of feature Active/PassiveOpen. The ActiveOpen feature is optional because as we mentioned before certain protocols are not providers of communication channels and don't fit into that model. The PassiveOpen feature is optional because enabling addresses is needed only for nodes running servers, which may not be required, for instance, by handheld computers. If the PassiveOpen feature is selected, then the ActiveOpen feature must be selected as well.

Domain Design

Domain design is accomplished using a conceptual tool called the GenVoca grammar [4]. A GenVoca grammar defines the architecture of a domain as a hierarchy of layers of abstraction. Each layer groups together features or concepts, which are independent of each other. Each layer is defined as a refinement of the layers

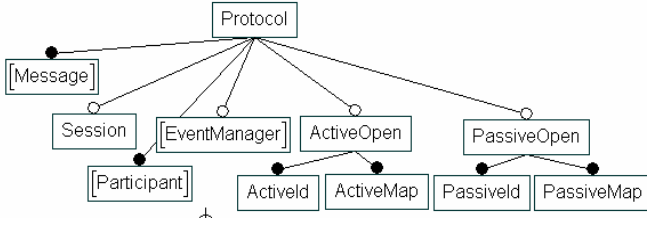


Fig. 3. The Protocol concept with internal features.

below. This hierarchy reflects the dependencies between the features and concepts, which are present in the feature analysis. In the hierarchy, we go from the more specialized (top) to the more general (bottom). Features or concepts that appear in each layer are called components and are mapped to classes, in a C++ implementation. A GenVoca grammar for protocol stacks is listed hereafter. There are seven named layers of abstraction L1 (top) to L7 (bottom):

```

Protocol: (L1)
  PassiveOpen[POWithOptA0] | ActiveOpen[Config] | ProtocolUI
POWithOptA0: (L2)
  ActiveOpen[Config]
Config: (L3)
  ProtocolUI EventManager ActiveId ActiveMap PassiveId PassiveMap
ProtocolUI: (L4)
  Protocol[ProtocolUIConfig]
ProtocolUIConfig: (L5)
  Message Session Participant
SessionUI: (L6)
  Session[SessionConfig]
SessionConfig: (L7)
  Message EventManager

```

Mapping of the feature diagrams to the GenVoca grammar is done according to Ref. [4]. The Message and EventManager features of the Session concept, are collected together in the bottom layer (L7), also called a configuration repository. This configuration repository is used to define the SessionUI layer (L6), which reflects the Session concept.

For the Protocol concept, we make a distinction between the essential features that are required to interact with a protocol and additional features that are needed to define the internal of a protocol. We came to this distinction after several iterations. It separates the protocol declaration (an interface) from the protocol definition (an implementation). To interact with a protocol, only the Message, Session and Participant features are required. They are collected together in a repository (L5). It is used to define the layer above, named ProtocolUI (L4), which is the abstraction of an interface to a protocol.

The layer labeled Config (L3) is the configuration repository for a protocol definition and includes a dependency on the ProtocolUI layer and all additional internal features. The ActiveOpen feature is mapped to the POWithOptAO layer (L2), which depends on the Config layer. The PassiveOpen feature depends on the ActiveOpen feature. This is reflected by a dependency from the L1

layer to the POWithOptAO layer. The Protocol layer (L1) groups three different alternatives that can be taken to define a protocol, namely, PassiveOpen parameterized with POWithOptAO (provides a basis for protocols that have sessions with client/server behavior), ActiveOpen parameterized with Config (provides a basis for protocols with sessions with client behavior only) or ProtocolUI (for protocols that don't embed sessions, e.g. top level protocols and certain drivers). Alternatives are separated by vertical bars.

4 Implementation

This section is about the implementation of the components uncovered in Section 3. This application is rather complex. In this paper it is possible to address a limited number of key elements. The full source code is available on-line [1].

Components with parameters are represented as C++ class templates. In other words, the GenVoca grammar is a generator of hierarchies of classes. With the GenVoca grammar translated to C++, the assembly of a base class for a new protocol class is as follows, informally:

```
PassiveOpen<ActiveOpen<Protocol<Config> > >
```

It results from the composition of a series of components. The Protocol component is configured with the Config repository, which is a structure (defined with the **struct** statement) with type members (defined with the **typedef** statement). The result is the actual parameter of an ActiveOpen component which is in turn the actual parameter of a PassiveOpen component.

In our model, such a generated type definition is used as a base for the definition of a specific protocol. In this base, all the architecture related code is present. What remain to be defined are specific message interpretation rules of a specific protocol. Here is an outline of the class template representing the protocol component.

```
template <class Generator> class Protocol {
public:
    typedef typename Generator::Config Config;
    // retrieve all the needed types from "Config"
    typedef typename Config::Session Session;
    typedef typename Config::Message Message;
    typedef typename Config::Participant Participant;
    // Constructor
    Protocol() {}
    // Destructor
    virtual ~Protocol() {}
    // Does an active open of a session
    virtual Session *xOpen(Protocol *hlp, Participant **p)
    { return NULL; }
    // Does close an active session
    virtual Outcome xClose(Session *s)
    { return OK; }
    // Other operations of the uniform interface ...
    // Layer interconnection operations
    Protocol * getmyHlp() { return myHlp; };
    void setmyLlp(Protocol * aLlp) { myLlp = aLlp; };
    Protocol * getmyLlp() { return myLlp; };
private:
```

```
// My high-level protocol
Protocol * myHlp;
// My low-level protocol
Protocol * myLlp; };
```

The Protocol class template has a parameter named **Generator** with a field named **Config**. It is also called a repository and its role is to pass to the template the choices needed to obtain a specific variant. The Protocol class template declares the operations of the uniform interface. They are declared virtual because they can be overridden in derived classes. The field **Generator::Config** is exported under the data member named **Config**, which makes it accessible to other components down in an inheritance hierarchy.

The **xOpen()** operation does an active open of a session. It takes in input a reference to a high-level protocol opening the session (**hlp**), a pair of participant addresses (**p**), **p[0]** is local and **p[1]** is remote. In output, it returns a reference to a session. The **xClose()** operation closes an active session. It takes in input a reference to an active session (**s**). It returns OK/NOK to indicate success/failure to close the session. Declarations of other similar operations are omitted. Then there are operations for connecting protocol layers together, which are required to assemble a protocol graph.

A generator takes a specification of a uniform protocol interface (which is normally realized by all protocols of a protocol stack) and returns the finished interface. It is defined as follows:

```
template < class SessionClass, class ParticipantClass = Part,
          class MessageClass = SingleBuffMessage >
class PROTOCOL_UI_GENERATOR {
    // define a short name for the complete generator
    typedef PROTOCOL_UI_GENERATOR <
        SessionClass, ParticipantClass, MessageClass >
        Generator;
    // create configuration
    struct Config {
        typedef SessionClass Session;
        typedef ParticipantClass Participant;
        typedef MessageClass Message; };
public:
    // assembly
    typedef Protocol<Generator> RET; };
```

A generator is another class template that encapsulates the rules defining how components can be assembled together. It knows what goes with what and under which constraints. It takes a specification of what needs to be assembled. It verifies its consistency and can provide default values. The generator of the uniform interface (UI) of protocols takes a protocol description and returns a finished protocol UI base class. During this generation, the actual classes of the Message, Participant and Session features are selected. The result of the assembly is returned as the RET type member. A common interface to all protocols, within a stack, is generated as follows.

```
typedef PROTOCOL_UI_GENERATOR < SessionUI >::RET ProtocolUI;
```

It is the creation of an instance of the Protocol class template, which is renamed ProtocolUI. A value (SessionUI) for the feature Session is specified. Other

features take default values. We now explain key elements of the implementation of a component in the hierarchy, namely, the `ActiveOpen` component which source is outlined hereafter:

```
template <class Generator>
class ActiveOpen: public Generator::Config::Base {
public:
    // export the name "Generator::Config" under the name "Config"
    typedef typename Generator::Config Config;
    // retrieve all the needed types from "Config"
    typedef typename Config::Base Base;
    typedef typename Config::Session Session;
    typedef typename Config::Message Message;
    typedef typename Config::Participant Participant;
    typedef typename Config::ActiveId ActiveId;
    typedef typename Config::ActiveMap ActiveMap;
    typedef typename Config::Header Header;
    typedef typename Config::EventManager EventManager;
    static const int hdrSize = sizeof(Header);
    typedef typename Config::ProtocolUI ProtocolUI;

    // active open of a session
    virtual Session *xOpen(ProtocolUI *hlp, Participant **p) {
        ActiveId key;
        Session * aSession;
        // execution of a protocol specific preamble (key construction!)
        if (xOpenPreamble(&key, p) != OK) {
            // execution of the preamble failed
            return NULL;
        }
        // is this session in the active map
        if ((aSession = activeMap.resolve((char *)&key)) != NULL) {
            // add a reference
            aSession->addRef();
            // session is in the active map, return it
            return aSession;
        }
        // session not in the active map, create it!
        aSession = createSession(&key);
        if (aSession == NULL) {
            // creation of session failed, return NULL
            return NULL;
        }
        // store the new binding in the active map
        if (activeMap.install((char *)&key, aSession) == NULL) {
            delete aSession; // recover the allocated memory
            return NULL;
        }
        // end with success
        return aSession;
    }
    // definition of other operations ... }
}
```

The `ActiveOpen` class template has one parameter named `Generator`, which supplies as sub fields of the `Config` field the `Base` base class name and choices of features. Note that the format of headers of protocol data units is passed within the configuration as the `Header` field. The `ActiveOpen` class template is responsible of keeping track of sessions that are active. It creates and deletes entries in the active map. Each entry is a pair made of a key (structure specific to each protocol) and a reference to an active session.

The definition of the `xOpen()` operation is detailed. The `xOpen()` operation creates a session. Arguments are a high-level protocol `hlp`, a reference to the

caller of the operation and a local participant and a remote participant contained in `p`. Firstly, an active id key is constructed by calling the `xOpenPreamble()` operation. The active map is consulted to determine if there is already a session associated to the key. If a session exists, then its reference count is incremented. Else, a new session `s` is created by calling the `createSession()` operation. If creation of `s` succeeds, then a new entry is created in the active map binding the key and `s` together. This operation depends on the implementation of the `xOpenPreamble()` and `createSession()` operations, which are done in a derived class.

The generator of a protocol is given below:

```
template <
    class BaseClass, class ActiveIdClass, class ActiveMapClass,
    class EnableClass, class PassiveIdClass, class PassiveMapClass,
    class HeaderClass, class EventManagerClass, class ProtocolUIClass,
    ActiveopenFlag activeopenFlag = with_activeopen,
    PassiveopenFlag passiveopenFlag = with_passiveopen >
class PROTOCOL_GENERATOR {
    // define a short name for the complete generator
    typedef PROTOCOL_GENERATOR <
        BaseClass, ActiveIdClass, ActiveMapClass,
        EnableClass, PassiveIdClass, PassiveMapClass,
        HeaderClass, EventManagerClass, ProtocolUIClass,
        activeopenFlag, passiveopenFlag > Generator;
    // parse domain specific language
    enum {
        isActiveopen = activeopenFlag==with_activeopen,
        isPassiveopen = passiveopenFlag==with_passiveopen };
    // assemble feature ActiveOpen
    typedef IF<isActiveopen,
        ActiveOpen<Generator>,
        BaseClass
    >::RET ActiveopenBase;
    // assemble feature PassiveOpen
    typedef IF<isPassiveopen,
        PassiveOpen<ActiveopenBase>,
        ActiveopenBase
    >::RET ProtocolBase;
    // create configuration
    struct Config {
        typedef BaseClass Base;
        typedef typename Base::Session Session;
        typedef typename Base::Participant Participant;
        typedef ActiveIdClass ActiveId;
        typedef ActiveMapClass ActiveMap;
        typedef EnableClass Enable;
        typedef PassiveIdClass PassiveId;
        typedef PassiveMapClass PassiveMap;
        typedef HeaderClass Header;
        typedef EventManagerClass EventManager;
        typedef typename Base::Message Message;
        typedef ProtocolUIClass ProtocolUI; };
public:
    typedef ProtocolBase RET; };
```

The generator reflects the rules of the GenVoca grammar of Section 3. The `BaseClass` parameter is a base class and the assembly is defined as an extension of it. Logically, it is protocol UI. The `activeopenFlag/passiveopenFlag` selects the inclusion of the `ActiveOpen/PassiveOpen` feature. The result, returned as the `RET` type member, is a base class which can be used through an extension relationship to define a specific protocol.

Configuration of the WLAN Protocol

A user level Wireless LAN protocol is used as an example. The WLAN protocol is a user-level abstraction of an 802.11 interface [7]. A WLAN address is defined by the `WLANAddr` structure. An address is six bytes long, stored in the `data` member.

```
#define WLAN_ADDR_LEN 6
struct WLANAddr {
    // address
    unsigned char data[WLAN_ADDR_LEN]; };
```

The configuration code is as follows:

```
// (1) format of a WLAN header
struct WLANHeader {
    WLANAddr destAddr;
    WLANAddr srcAddr;
    unsigned short type;
#define IP_TYPE x0800
};
#define WLAN_HEADER_LEN 14 // bytes

// (2) definition of a key in the active map
struct WLANActiveId {
    WLANAddr localAddr;
    WLANAddr remoteAddr; };

// (3) configuration of the Active Map
struct WLANActiveMapConfig {
    static const int KEYSIZE = sizeof(WLANActiveId);
    typedef SessionUI ValueType;
    static const int HASHSIZE = 50;
    typedef Map<WLANActiveMapConfig> WLANActiveMap; };

// (4) definition of a key in the passive map
struct WLANPassiveId {
    WLANAddr localAddr; };

// (5) define enable structure
struct WLEnable {
    unsigned int refCnt;
    ProtocolUI *hlp; // ptr to the high-level protocol };

// (6) configuration of passive map
struct WLANPassiveMapConfig {
    static const int KEYSIZE = sizeof(WLANPassiveId);
    typedef WLEnable ValueType;
    static const int HASHSIZE = 50;
    typedef Map<WLANPassiveMapConfig> WLANPassiveMap; };

// (7) generation of the base class used to derive the WLAN protocol
typedef PROTOCOL_GENERATOR <
    ProtocolUI,
    WLANActiveId,
    WLANActiveMapConfig::WLANActiveMap,
    WLEnable,
    WLANPassiveId,
    WLANPassiveMapConfig::WLANPassiveMap,
    WLANHeader,
    SimpleEvent,
    ProtocolUI,
    with_activeopen,
    with_passiveopen >::RET WLANBase;
```

The `WLANHeader` structure defines the format of the header of frames of the WLAN protocol. It consists of a destination address, `destAddr`, a source address, `srcAddr` and a `type` field. The value of the `type` field determines the protocol to which the payload of the frame is associated.

The WLAN protocol has an active map and a passive map. The active map keeps track of sessions managed by the protocol. The passive map keeps track of local addresses with which the protocol is willing to create new sessions. The `WLANActiveId` structure defines the format of a key installed in the active map. It consists of a local address, `localAddr`, and a remote address, `remoteAddr`. Each key installed in the active map is paired with a pointer to an object of a class extending the `SessionUI` class, more specifically a class derived from `SessionUI` that defines a WLAN session. The active map is defined as an actualization of the `Map` template. The configuration passed to the template defines the size of a key, type of values pointed by a pointer paired with a key and hash size. The actualization is named `WLANActiveMap`.

The `WLANPassiveId` structure defines the format of a key installed in the passive map. It consists of a local address, `localAddr`. Each key installed in the passive map is paired with an instance of the `WLANEnable` structure. It consists of an integer counting the number of invocations to the open enable operation that have been performed using address `localAddr` and a pointer to the high-level protocol that was the first to invoke the open enable operation on this address, `hlp`. The passive map is defined as an actualization of the `Map` template. The configuration passed to the template defines the size of a key, type of values pointed by a pointer paired with a key and hash size. The actualization is named `WLANActiveMap`.

Finally, the `WLANBase` base class, used to derive the WLAN protocol, is produced using the `PROTOCOL_GENERATOR` generator. The actual arguments of the generator are the name of the parent class of `WLANBase` (`ProtocolUI`), the type of a key in the active map (`WLANActiveId`), the type of the active map (`WLANActiveMap`), the type of values pointed by entry stored in the passive map (`WLANEnable`), the type of a key in the passive map (`WLANPassiveId`), the type of the passive map (`WLANPassiveMap`), the type of the frame header (`WLANHeader`), the type of the chosen event manager (`SimpleEvent`), the class defining the uniform protocol interface (`ProtocolUI`), a value forcing the selection of the `ActiveOpen` feature (`with_activeopen`) and a value forcing the selection of the `PassiveOpen` feature (`with_passiveopen`). The `ActiveOpen` feature provides all the mechanisms required to maintain the active map while the `PassiveOpen` feature provides all the mechanisms required to maintain the passive map.

Class Diagram

The class diagram for the WLAN protocol is pictured in Figure 4. Each rectangle represents a class while each arrow represents an inheritance relation from a derived class to a parent class. The `ProtocolUI` class is an abstract class resulting from the actualization of the class template implementing the `Protocol` concept. It represents a uniform interface common to all protocols. The `ActiveOpen` and

PassiveOpen classes represent the actualizations of the class templates implementing the ActiveOpen and PassiveOpen features. The WLANBase class is a synonym for the actualization of the PassiveOpen class template (equivalent to typedef of the C language). The WLANProtocol class represents the implementation of the WLAN protocol and augments WLANBase with aspects that are specific to the WLAN protocol. WLANBase has pointers to instances of the WLANSession class (represented by the relationship ended by a diamond on the side of the container).

The SessionUI concept is an actualization of the class template implementing the Session concept. It represents a uniform interface common to all sessions. The WLANSession class represents the implementation of a session (i.e. a channel end-point) managed by the WLAN protocol. It implements the operations declared in the SessionUI class.

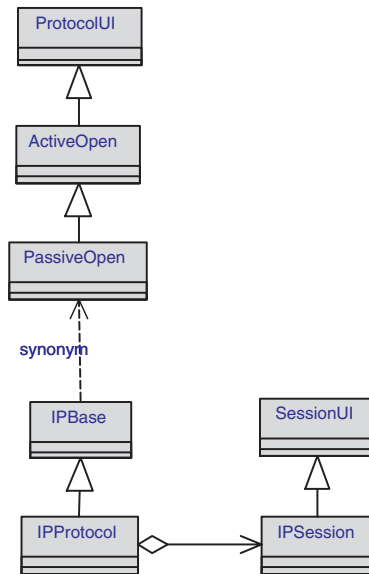


Fig. 4. Class diagram for the WLAN protocol.

5 Towards Dynamic Re-configurability

The main objective of generative programming is to develop families of systems. Until now, the concepts of generative programming have been almost exclusively used to generate systems at compile-time [4] by means of a generator. The

role of the generator consisting in producing a system from a feature specification, described using a GenVoca grammar, input by a user. The generator is also responsible for validating feature specifications and detecting invalid feature combinations. Once the feature specification is validated, the generator produces the requested system by assembling predefined components.

While the current usage of generative techniques offers great potential in the context of software engineering by allowing for the development of families of systems, it remains limited with respect to other crucial issues like system run-time reconfiguration.

It is our opinion that the use of generative techniques should not be limited to static compile-time system configuration, but should also be used to develop systems that could be dynamically configured and reconfigured at run-time, which is considered to be a very important issue in today's software industry. A software system might have to be reconfigured for different reasons. For example, the user wants to change the set of features of the system, or because the current system's operation has degraded to the point where it no longer satisfies the required quality of service (QoS). The system might also have to change because the context in which it evolves has changed. To use the analogy of the car industry discussed in [4], what we often need to build in software engineering is a car that could reconfigure itself to move the driver seat from the left side of the car to the right side of the car when the car moves from France to England.

In our research project, we are currently working on the development of a reconfigurable protocol stack using generative techniques. The goal is to develop a system architecture that would allow the protocol stack to be initially configured and reconfigured at run-time without having to shutdown and restart the system. This would also allow for the online upgrade of the system.

Proposed Architecture

A high-level architecture of our reconfigurable protocol stack is given in Figure 7. The key element of this architecture is the Configuration Agent. The role of the Configuration Agent is in part similar to the role of a generator in generative programming. Initially, the Configuration Agent receives the specification of a desired set of features, described using a GenVoca grammar, it validates the set of features and then if the feature specification is valid, it configures the system by loading the requested components and linking them together in a communication topology that allows satisfying the user feature specification.

However, unlike a generator, the Configuration Agent also has the capability of dynamically reconfiguring the system at run-time. The triggering of the reconfiguration could be external to the system, e.g. an explicit command input by a user, or internal, e.g. as a result of decreasing performance internally observed by the system. The use of internally triggered reconfiguration requires the existence of some internal mechanisms that can determine when reconfiguration must occur. For example, a system might have a performance-monitoring component that would inform the Configuration Agent to reconfigure when the system performance reaches certain thresholds.

Upon reception of a reconfiguration request, the Configuration Agent must determine if the system is in a proper state for reconfiguration. Two parameters must be considered at this point: the type of reconfiguration and state of the system. For this reason, the Configuration Agent needs to maintain, or have ways of obtaining, information concerning the state of the system. To continue with the car analogy, it is important that the Configuration Agent of the car verifies the state of the car before initiating reconfiguration. If the reconfiguration request is about moving the driver seat from left to right, the Configuration Agent must first ensure that the car is stopped and that nobody sits in the front seats before initiating the reconfiguration. However, other types of reconfiguration, like switching from two-wheel-drive to four-wheel-drive, might not require stopping the car.

If the system is not in a proper state for reconfiguration, the Configuration Agent must either reject the reconfiguration request or bring the system to an acceptable safe state before initiating the reconfiguration.

During reconfiguration, the Configuration Agent is responsible for controlling and synchronizing the reconfiguration of the different components and for bringing the system back to its operational state once the reconfiguration is completed.

Protocol Layer Modifications

In order to develop a reconfigurable protocol stack, the different layers that compose the system would also have to be modified to allow for layer reconfiguration. The required layer modifications are twofold: layer interface and layer implementation. The layers of the protocol stack described in the previous sections have two main standard interfaces: the upper-layer interface and the lower-layer interface. The definition of a reconfigurable protocol stack requires the addition of a third standard interface to the different layers for the reconfiguration aspect. These interfaces are the ones used for the communication between the Configuration Agent and the different layers of the communication protocol stack. While the configuration messages defined in the configuration interfaces are standard for all layers, the behaviour executed upon reception of a configuration message is layer dependent.

The implementation of the different layers would also have to be modified to allow for reconfiguration. This can be done by recursively using the concept of a Configuration Agent inside each layer.

Tradeoffs

The main tradeoff between the conventional protocol stack presented in Section 3 and the reconfigurable one described in this section is the one of size (footprint) versus flexibility. The addition of the reconfigurability aspect to the system results in a more complex and larger system, but has the huge advantage of being more flexible.

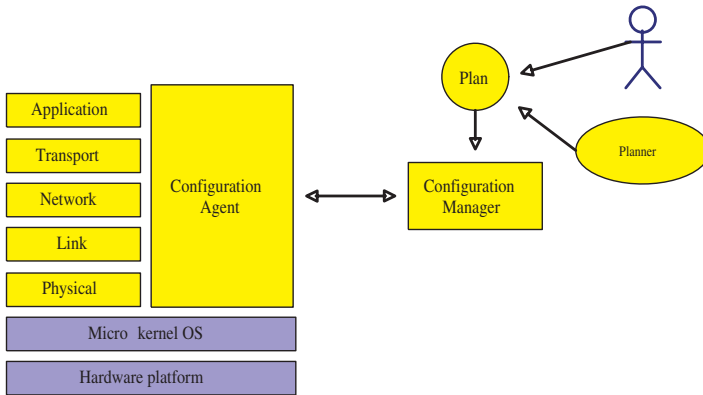


Fig. 5. Architecture of the reconfigurable protocol stack.

In the next generation of our protocol stack, reconfigurability will be considered as a feature. Therefore, our generator will have the capability of i) generating a conventional protocol stack, as described in Section 3, for cases where the footprint is a main concern and ii) generating a reconfigurable protocol stack for cases where reconfigurability is a main concern.

6 Conclusion

Our work consists of a unique combination of well established techniques for protocol implementation and generative programming. Our hypothesis is that by combining these techniques we provide a higher degree of configurability, but not at the expense of performance, correctness or resource usage. Our current implementation of the feature diagrams of Figures 2 and 3 offers 18 different alternatives (there are two alternatives for Message, three for EventManager and three for Protocol). We claim configurability superiority for crosscutting concerns, with respect to several protocol development environments. Regarding performance, correctness and resource usage, apart from the usage of our implementation to develop several protocols, systematic evaluation of the implementation of some of the crosscutting concerns has been conducted, namely, the Message and EventManager features [16]. The performance so far seems to be comparable with the performance obtained in other contexts. It is although premature for us to claim superiority of performance, correctness and resource usage. Further work needs to be conducted to clarify exactly our position on that side.

References

1. Barbeau, M.: Software Engineering for Telecommunications Protocols. Web page, www.scs.carleton.ca/~barbeau/Courses/SETP/index.html

2. Beck, M., Bohme, H., Dziadzka, M., Kunitz, U., Magnus, R., Verworner, D.: *Linux Kernel Internals - Second Edition*. Addison Wesley Longman (1998)
3. Birman, K., Constable, R., Hayden, M., Kreitz, C., Rodeh, O., van Renesse, R., Vogels, W.: *The Horus and Ensemble Projects: Accomplishments and Limitations*. Proc. of the DARPA Information Survivability Conference and Exposition (DIS-CEX '00). Hilton Head, South Carolina (2000)
4. Czarnecki, K., Eisenecker, U.W.: *Generative Programming Methods, Tools, and Applications*. Addison Wesley (2000)
5. Czarnecki, K., Eisenecker, U.W.: *Components and Generative Programming*. Proceedings of the 7th European Engineering Conference held jointly with the 7th ACM SIGSOFT symposium on Foundations of software engineering (1999) 2–19
6. Hutchinson, N.C., Peterson, L.L.: *The x-Kernel: An Architecture for Implementing Network Protocols*. IEEE Transactions on Software Engineering **17** (1) (1991) 64–76
7. LAN/MAN Standards Committee of the IEEE Computer Society: *Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications: Higher-Speed Physical Layer Extension in the 2.4 Ghz Band*. IEEE Std 802.11b (1999)
8. Mellquist, P.E.: *SNMP++: An Object-Oriented Approach to Developing Network Management Applications*. Prentice Hall (1998)
9. van Renesse, R., Birman, K., Hayden, M., Vaysburd, A., Karr, D.: *Building Adaptive Systems Using Ensemble*. Software-Practice and Experience **28** 9 (1998) 963–979
10. Schmidt, D.C., Huston, S.D.: *C++ Network Programming: Systematic Reuse with Frameworks and ACE*. Addison-Wesley Longman (2002)
11. Schmidt, D.C., Huston, S.D.: *C++ Network Programming: Mastering Complexity Using ACE and Patterns*. Addison-Wesley Longman (2001)
12. Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F.: *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Wiley and Sons (2000)
13. Wade, I.: *NOSintro - TCP/IP Over Packet Radio - An Introduction to the KA9Q Network Operating System*. Dowermain Ltd. (1992)
14. Wright, G.R., Stevens, W.R.: *TCP/IP Illustrated - Volume 2*. Addison Wesley (1995)
15. Xu, Y.: *Implementation of Location Detection, Home Agent Discovery and Registration of Mobile IPv6*. Master Thesis, School of Computer Science, Carleton University (2001) (Available at: www.scs.carleton.ca/~barbeau/Thesis/xu.pdf)
16. Zhang, S.: *Channel Management, Message Representation and Event Handling of a Protocol Implementation Framework for Linux Using Generative Programming*. Master Thesis, School of Computer Science, Carleton University (2002) (Available at: www.scs.carleton.ca/~barbeau/Thesis/zhang.pdf)

Building Composable Aspect-Specific Languages with Logic Metaprogramming

Johan Brichau^{1*}, Kim Mens², and Kris De Volder³

¹ Vrije Universiteit Brussel,
Vakgroep Informatica,
Pleinlaan 2,
B-1050 Brussel, Belgium
`johan.brichau@vub.ac.be`

² Université catholique de Louvain,
Place Sainte-Barbe 2,
B-1348 Louvain-la-Neuve, Belgique
`kim.mens@info.ucl.ac.be`

³ University of British Columbia,
309-2366 Main Mall,
V6T 1Z4 Vancouver, BC, CANADA
`kdvolder@cs.ubc.ca`

Abstract. The goal of aspect-oriented programming is to modularize crosscutting concerns (or aspects) at the code level. These aspects can be defined in either a general-purpose language or in a language that is fine-tuned to a specific aspect in consideration. Aspect-specific languages provide more concise and more readable aspect declarations but are limited to a specific domain. Moreover, multiple aspects may be needed in a single application and composing aspects written in different aspect languages is not an easy task.

To solve this composition problem, we represent both aspects and aspect languages as modularized logic metaprograms. These logic modules can be composed in flexible ways to achieve combinations of aspects written in different aspect-specific languages. As such, the advantages of both general-purpose and aspect-specific languages are combined.

1 Introduction

The idea of *separation of concerns* [16] is that the implementation of all concerns in a software application should be cleanly modularized. Today's existing programming techniques have succeeded to support the separation of concerns principle at the code level to a reasonable degree. However, some concerns cannot be modularized using the existing modularizations and tend to crosscut with other concerns. Common examples of such concerns are synchronisation, persistence and error-handling. Aspect-oriented programming (AOP) [8] modularizes such crosscutting concerns as aspects. These aspects are expressed in one or

* Research assistant of the Fund for Scientific Research – Flanders (Belgium) (F.W.O.)

more aspect languages and they are composed with the rest of the program by an aspect weaver.

An aspect language designed to express a specific kind of aspect is highly desirable because it results in more concise and more intentional aspect declarations, making it easier to read and write aspect code. A testament to this is the fact that many of the first aspect languages were aspect-specific [6,10,11] and *not* general purpose. On the other hand, aspect-specific languages are very powerful within their specific scope but they can only be applied to the specific aspect they were designed for. Because of this, many AOP-related techniques [1,7,15] offer a general-purpose aspect language which allows to express many kinds of aspects as well as combinations and interactions between them. The latter becomes more complex when the aspects are implemented in different aspect-specific languages. Our approach to AOP is based on logic metaprogramming (LMP) [12,13,14,20,21]. In a previous paper [4], we explored the use of LMP as an open and extensible aspect-weaver mechanism that facilitates the specialization of a general-purpose aspect language and, as such, facilitates the building of aspect-specific languages (ASLs). In this paper we focus on the combinations and interactions between aspects written in different ASLs. We extend the work in [4] and introduce logic modules to encapsulate aspects and implementations of ASLs. These logic modules provide a composition mechanism that allows us to combine aspects or implement interactions between aspects written in different ASLs. This is made possible because all our ASLs share the same Prolog-like base language. In other words, we obtain a modular aspect-weaver mechanism that offers the generality of a general-purpose aspect language without losing the ability and advantages of defining aspects in aspect-specific languages. In addition, we offer a means of composing and regulating the interactions among different aspect-specific languages. In section 2, we introduce a software application that is used as a running example throughout the remainder of the paper. In section 3, we describe what an aspect language embedded in a logic language looks like and how it supports modularization of crosscutting concerns. Section 4 explains how ASLs are implemented and section 5 shows how aspect composition and interaction issues are handled. We briefly introduce a prototype research tool for our approach in section 6. Sections 7 and 8 discuss future and related work.

2 Case: The Conduit Simulator

2.1 Basic Functionality

Our running example is a conduit simulator, implemented in Smalltalk and which allows to simulate the flow of fluid in a network of conduits. A conduit system can be built using 4 types of conduits: pipes, sources, sinks and joins. An example of a conduit-system is shown in figure 1 and a simplified class diagram of the implementation is shown in figure 2.

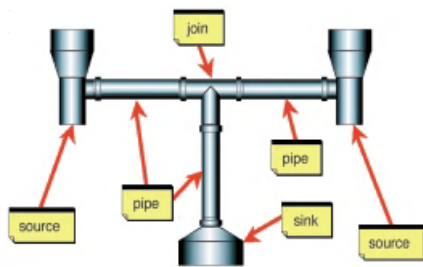


Fig. 1. A Conduit-system.

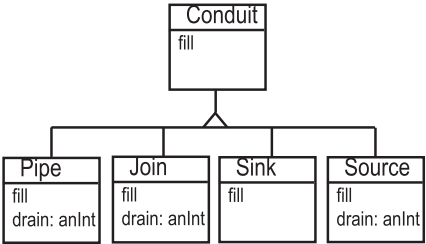


Fig. 2. Class diagram of the Conduit Simulator.

Each type of conduit is implemented by a single class. A conduit-system is built by linking each conduit to an incoming conduit from which it should receive fluid¹. The basic behaviour is implemented in two methods:

- #drain:** Each drainable conduit (source, pipe and join) understands the message **#drain**: which can be used to drain an amount of fluid from it.
- #fill** The **#fill** method of each conduit tries to fill the conduit by draining the incoming conduit(s). A source conduit fills itself based on a profile.

All conduits simultaneously run a looping process that executes the **#fill** method, i.e. conduits are active objects that continuously drain their incoming conduit(s). As a result, fluid will flow from sources to sinks.

2.2 Crosscutting Functionality

Making the conduit simulator work correctly requires us to deal with some cross-cutting concerns.

Synchronizing and Order of Execution. A conduit can only be drained after each time it has been able to fill itself. Therefore, the **#fill** and **#drain**: methods can only be executed in alternating order.

This can be done by inserting synchronisation code at the beginning and at the end of both these methods. Obviously, this leads to tangled functional and non-functional code. In the rest of the paper, we will refer to this aspect as the ‘order of execution’ aspect.

User Interface (UI). We also need to visualize the real-time simulation. Therefore, we create views for each type of conduit and use an Observer design pattern to couple them. The code for this pattern also crosscuts the implementation of all conduit types.

¹ Join conduits are linked to two incoming conduits.

Logging. For debugging purposes, we want to log the execution of the `#drain:` and `#fill` methods, which also amounts to the introduction of code in the beginning and at the end of those methods. Once again, this would tangle logging code with functional code. Also, this addition requires the insertion of very similar code in many places. It is also important to note that writing to the log should also be synchronized.

In the following section, we explain how to write the logging aspect in a general-purpose aspect language (implemented using our LMP approach) and we elaborate on building aspect-specific languages for all aspects of the conduit simulator in section 4.

3 Aspects in a Logic Language

In LMP, we use a logic programming language to reason about object-oriented base programs. The metalevel description of the base-language program consists of logic facts and rules [20]. In the context of AOP, the logic language also serves as the surrounding medium in which we embed our aspect languages. This provides a general framework for declaring and implementing aspects.

3.1 Aspects as Logic Modules

An aspect language is specified as a set of logic predicates, as is shown in table 1 for a general-purpose ‘advice’ aspect language (similar to advices in AspectJ [7]). An aspect in this aspect language is implemented as a set of logic declarations of these predicates, contained in a logic module. An example aspect is shown in figure 3 (this is only a first and simple version of the aspect that will be improved in later sections). The logic inference engine becomes the weaver, which gathers all the logic declarations that are present in such a module. The weaver (for a particular aspect language) understands the declarations and knows how to weave the aspect into the base program. In the code fragments, all predicates that are part of an aspect language are shown in **bold**.

We first provide some details about the syntax of our logic language:

- We have a special logic term (delimited by curly braces: ‘{’ and ‘}’) to embed base program text in the logic declarations. This term can even contain logic variables (their use is explained later, also see [4]).
- Logic variables start with a question mark (e.g. `?variable`).
- The modules basically encapsulate logic declarations. Each logic declaration belongs to a module and is only visible in the module where it is defined.
- Modules can be composed to use declarations of another module or to make declarations in one module visible in another module. How this is done is explained later.

Table 1. A simple advice aspect-language (similar to advices in AspectJ).

Predicate	Description
<code>adviceBefore(?m,?c)</code>	Execute code fragment ?c before executing method ?m
<code>adviceAfter(?m,?c)</code>	Execute code fragment ?c after executing method ?m

Simple Logging Aspect

```

adviceBefore(method(Pipe,drain:),{ Logger log:'Enter Pipe>>drain:' for:thisObject}).
adviceAfter(method(Pipe,drain:),{ Logger log:'Exit Pipe>>drain:' for:thisObject}).
adviceBefore(method(Pipe,fill),{ Logger log:'Enter Pipe>>fill' for:thisObject}).
adviceAfter(method(Pipe,fill),{ Logger log:'Exit Pipe>>fill' for:thisObject}).
adviceBefore(method(Join,drain:),{ Logger log:'Enter Join>>drain:' for:thisObject}).
adviceAfter(method(Join,drain:),{ Logger log:'Exit Join>>drain:' for:thisObject}).
adviceBefore(method(Join,fill),{ Logger log:'Enter Join>>fill' for:thisObject}).
adviceAfter(method(Join,fill),{ Logger log:'Exit Join>>fill' for:thisObject}).

```

Fig. 3. Logging aspect in the advice aspect language

The aspect shown in figure 3 implements the logging for the conduit-system in the aspect language defined in table 1. The **Logger** class keeps a log for each conduit. The logic declarations inform the weaver that some code fragments must be executed before and after the methods **#drain:** and **#fill** in the classes **Pipe** and **Join**. Technically, the weaver gathers all **adviceBefore** and **adviceAfter** declarations and weaves the code fragments in the indicated methods. The **thisObject** keyword is understood by the weaver and is a reference to the object in which the code fragment is executed.

3.2 Logic Pointcuts

An aspect describes where or when its functionality should be invoked in terms of joinpoints. Pointcuts are sets of joinpoints. In our approach, joinpoints are specific points in the base program's code².

The primitive example above expressed an aspect by directly applying advices to individual joinpoints. Adequately expressing aspects also requires a mechanism to abstract over sets of joinpoints (pointcuts) and factor out commonalities between aspect code applied over all of them [3]. This involves (1) a pointcut mechanism to characterize sets of joinpoints, (2) a mechanism of parameterization that allows the aspect's code to have joinpoint-specific behavior. In the LMP approach, both these mechanisms are supported through the use of logic rules. We now discuss each mechanism in more detail.

Defining Pointcuts. Part of the implementation of the observer pattern (for the UI-aspect) is the insertion of code at well defined joinpoints, that triggers the observable's update mechanism (hence, updating the UI). Defining a separate

² Some experiments with dynamic joinpoints in LMP have been conducted in [5].

advice fact for each of those joinpoints would result in a lot of code duplication, because the advice is identical for each joinpoint. A better way to define the UI-aspect is through the use of logic rules, which are a way to define a set of similar of facts.

User Interface Aspect

```

adviceAfter(?method,{ depends do:[each | each update] }) if
  changesState(?method).

changesState(method(?class,fill)) if
  subclass(Conduit,?class),
  classImplementsMethod(?class,fill).

```

Fig. 4. Logic module implementing the UI-aspect.

The implementation of the ‘update’ part of the UI aspect is shown in figure 4. The first logic rule declares **adviceAfter** facts for each joinpoint that is matched by the pointcut defined by **changesState** declarations. The second logic rule defines this pointcut as the **#fill** method of each subclass of **Conduit**. The **subclass** and **classImplementsMethod** predicates are part of a predefined logic library of predicates to reason about Smalltalk code (see [20]).

This example was particularly easy, because the code is identical for each joinpoint of the pointcut. However, an aspect becomes significantly more complex if the code requires variations dependent on the specific joinpoint.

Joinpoint-Dependent Variations. The logging aspect in figure 3 is an example of an aspect that inserts a pattern of code containing joinpoint-dependent variations, i.e. the name of class and selector. We can capture these variation points using logic variables, embedded in the code pattern. Using this technique, we can implement a more generic logging aspect, as is shown in figure 5.

Improved Logging Aspect

```

adviceBefore(method(?class,?selector),{ Logger log:'Enter ?class>>?selector' for: thisObject }) if
  logMethod(?class,?selector).

adviceAfter(method(?class,?selector),{ Logger log:'Exit ?class>>?selector' for: thisObject }) if
  logMethod(?class,?selector).

logMethod(Pipe,drain:).
logMethod(Pipe,fill).
logMethod(Join,drain:).
logMethod(Join,fill).

```

Fig. 5. Logic module implementing the logging aspect.

The code pattern in the `adviceBefore` and `adviceAfter` declarations is now parameterized by logic variables `?class` and `?selector`. The weaver uses the inference engine to substitute them with their specific bindings, dependent on the joinpoint that the advice is woven into.

We have now explained what an aspect language embedded in a logic language looks like and how it is suited to describe crosscutting concerns that should be woven in the base program code. We now elaborate on the use and composition of logic modules to implement and use aspect-specific languages.

4 Aspect-Specific Languages

Aspect languages are implemented through logic rules in logic modules. These rules define the meaning of an aspect language in terms of a more primitive aspect language. This eases the implementation of ASLs because weavers for them do not have to be implemented from scratch. Typically, we have a primitive aspect weaver that implements a low-level, general-purpose aspect language on which other aspect-specific languages can be built. In the following subsections, we illustrate this with the construction of two aspect languages for logging and ‘order of execution’ in our conduit simulator. Both of these aspect languages are defined in terms of the more general-purpose advice aspect language.

Table 2. A simple logging aspect language.

Predicate	Description
<code>logMethod(?c,?m)</code>	Log the execution of the method <code>?m</code> in class <code>?c</code> .

Logging Aspect Language. The logging aspect of figure 5 already defined a simple aspect language for logging. The aspect language consists of a single predicate and is shown in table 2. The first two rules in figure 5 define the meaning of the logging aspect language in terms of the advice aspect language. The remaining facts constitute the implementation of the aspect. But the logic module in figure 5 contains both the logging aspect itself and the implementation of the logging aspect language. To facilitate reuse of the implementation of the aspect language, we prefer to separate the aspect implementation from the aspect language’s implementation. To achieve this, we split this module in an *aspect language module* and an *aspect definition module* and provide a composition mechanism to compose both modules.

The logic rules that implement the logging aspect language are placed in a separate aspect-language module. But now, these logic rules should gather the required `logMethod` declarations in a separate aspect-definition module, which depends on the particular application in which the aspect language is used. Therefore, we parameterize the aspect-language module using a module-variable, which can be bound to a specific aspect-definition module (containing

`logMethod` facts) in the context of a particular application. In the code fragments, all module-variables are shown in *italic*.

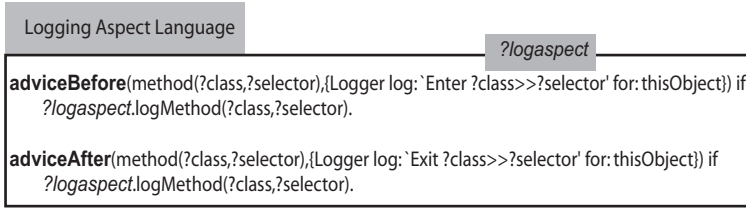


Fig. 6. Aspect-language module implementing the logging aspect language.

Figure 6 shows the aspect-definition module for the logging aspect language implemented in terms of the advice aspect language. The `logMethod` declarations will be gathered in the logic module that is bound to the `?logaspect` module-variable. Hence, the `?logaspect` module-variable parameterizes this logic module with another logic module. In the implementation of a particular application, we can bind the module-variable with an aspect definition module implementing a logging aspect, such as the one shown in figure 7.

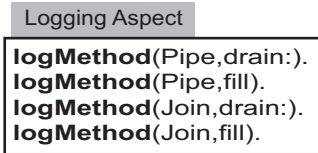


Fig. 7. Logging aspect that is understood by the logging aspect language implemented in figure 6.

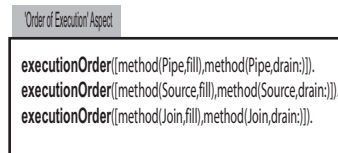


Fig. 8. ‘Order of execution’ aspect that is understood by the ‘order of execution’ aspect language of table 3.

While this example is rather simple, the use of ASLs is particularly interesting for aspects that can be reused in many different contexts (such as the more technical aspects that implement non-functional requirements like synchronization, distribution, ...) and where the code of the aspect is more complicated. The ASL shields the developer from the burden of the implementation while still enabling him to tailor the functionality of the aspect (to the extent that the ASL allows it).

‘Order of Execution’ Aspect Language. The above logging aspect language only allows to specify joinpoints for the logging aspect. The ‘order of execution’ aspect is much more interesting because the aspect language we constructed for it

provides ‘hooks’ that allow us to add behaviour to the aspect. The logic module shown in figure 8 is an aspect-definition module, implementing an aspect for our conduit simulator in the ‘order of execution’ aspect language. This language consists of three logic predicates, described in table 3. The last two predicates define hooks that allow the user of the aspect language to specify additional code that will be inserted in the implementation of the aspect.

Table 3. ‘Order of Execution’ aspect language.

Predicate	Description
<code>executionOrder(?list)</code>	The <code>?list</code> argument of this predicate is a list of methods that should be executed in mutual exclusion and in order of occurrence in the list. After the last method in the list is executed, the first method can again be executed.
<code>onBlock(?m,?c)</code>	Execute code <code>?c</code> when the method <code>?m</code> is blocked by the synchronisation guards.
<code>onStart(?m,?c)</code>	Execute code <code>?c</code> when the method <code>?m</code> is given permission to execute by the synchronisation guards.

The ‘order of execution’ aspect for our conduit simulator is shown in figure 8. In figure 9, we show part of the aspect-language module implementing the ‘order of execution’ ASL in terms of the advice language. Basically, the first two rules respectively describe before advices and after advices that insert a simple synchronization algorithm (using semaphores) on each of the methods given in the `executionOrder` declarations. The auxiliary `orderDependence` rule is part of the internal implementation of the ASL to compute which semaphores should be used by that particular advice and to gather the additional code fragments in the optional `onBlock` and `onStart` declarations.

‘Order of Execution’ ASL

```

adviceBefore(?jp,{ globalSema wait. (semaphores at: ?position)
                                waitAndExecute:[?onBlock. globalSema signal].
                                ?startCode }) if
  orderDependence(?jp,?position,?nextPosition,?blockCode,?startCode).

adviceAfter(?jp,{ (semaphores at: ?nextPosition) signal }) if
  orderDependence(?jp,?position,?nextPosition,?blockCode,?startCode).

orderDependence(?jp,?currentPos,?nextPos,?blockCode,?startCode) if
  ?orderAspect.executionOrder(?list),
  computePositions(?list,<?jp,?currentPos,?nextPos,?blockCode,?startCode>).
...

```

Fig. 9. Part of the aspect-language module implementing the ‘Order of Execution’ ASL.

We have shown how to build aspect-specific languages on top of a more general-purpose aspect language. These ASLs can be reused as black-box entities in many different development contexts through the use of logic rules in logic modules. We do not claim that the actual implementation of an ASL is a simple process. One still has to design an appropriate language and implement its semantics in terms of another (more low-level) aspect language. Also, the fact that all ASLs in LMP remain embedded in the same logic language, obviously bears some advantages as well as disadvantages. On the one hand, it ensures a common medium to express the composition of all aspects in these aspect-languages. On the other hand, no aspect-specific syntax is provided. However, we feel that the advantage is far more greater than the disadvantage because the combination of multiple aspects can raise many subtle and difficult issues that should be tackled by the programmer [9,19]. Also, nothing prohibits us to extend the approach to allow for aspect-specific syntax on top of the underlying logic description. We are currently looking into that issue.

5 Composition and Interaction

Combining multiple aspects in a single application can raise problems that do not exist when the aspects are considered in isolation. For example, in our conduit simulator, combining the logging aspect with the ‘order of execution’ aspect poses some complications:

A: Logging of methods that are ‘ordered’. How do we log methods that may block because of the ‘order of execution’ aspect?

A1 Do we log entry to a method before or after checking the guards?

A2 How do we log the fact that a method blocks?

B: Reducing synchronisation overhead for logging. To synchronize the `Logger` class, we use a synchronisation aspect. But the ‘order of execution’ aspect also synchronizes methods of each conduit and the log itself is different for each conduit. This means that if logging is only executed in the critical sections that are created by the ‘order of execution’ aspect, that it is safe to omit a supplementary synchronisation aspect. On the other hand, in some cases, we also might want to log other methods of a conduit than `#drain:` and `#fill`. In those cases, we do need proper synchronisation for the log aspect.

B1 How do we automatically apply a synchronisation aspect to the logging aspect?

B2 How do we reduce the amount of synchronisation code to be executed, based on interaction with the ‘order of execution’ aspect?

In the following subsections, we show how logic modules can be used to implement the aspect-combination complications mentioned above.

5.1 Combining Aspects

Aspects are combined using *aspect-combination modules*. A combination module is a logic module that is parameterized with several other modules and contains rules that describe how the functionality of these other modules is to be combined. This composition mechanism can be used to compose *aspect-definition modules*, as well as *aspect-language modules*. The aspect combination module then acts as their composition and therefore it is the only module to be handed to the weaver.

Dominates Combination Module. The aspect-composition problem A1 (order of the aspects) could be solved by prioritizing the aspects. This is a general solution which requires no domain-specific knowledge of the aspects themselves. Therefore it can be handled by a general type of prioritization rule. Figure 10 shows part of the ‘dominates’ combination module that prioritizes the advices generated by the aspect-language modules for logging and ‘order of execution’. The rule that handles **adviceAfter** is identical. In figure 12, we show how the ‘dominates’ combination module is used to ensure that the ‘order of execution’ aspect (dominating aspect) is executed before the logging aspect (dominated aspect). The actual composition of the modules, as depicted in figure 12, can either be defined by a logic program or in a visual composition tool (see [2] for a prototype of such a tool).

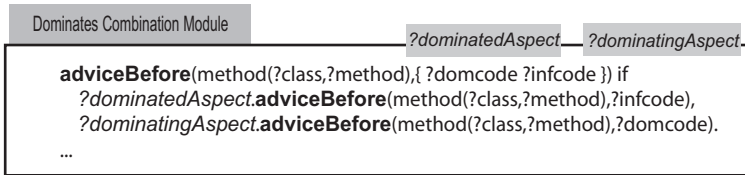


Fig. 10. The Dominates combination module to prioritize an aspect.

Wrapper Combination Module. Another kind of aspect-combination module is required for problem B1, where we merely want to wrap synchronisation code around the logging code. Using the previous ‘dominates’ combination module, this would result in wrapping synchronisation code around the entire method, instead of only around the logging code. Figure 11 shows part of a ‘wrapper’ combination module that produces the desired result. The rule fetches the before and after advice of the ‘wrapper’ aspect for every before advice of the ‘internal’ aspect and concludes a combined before advice. This combined before advice contains the ‘internal’ aspect’s before advice, surrounded with the ‘wrapper’ aspect’s advices. In our particular case, the ‘wrapper’ aspect is the synchronisation aspect and the ‘internal’ aspect is the logging aspect. Once again,

rules that handle **adviceAfter** predicates are similar and are omitted. Also, for simplicity, we do not include the implementation of this synchronisation aspect here.

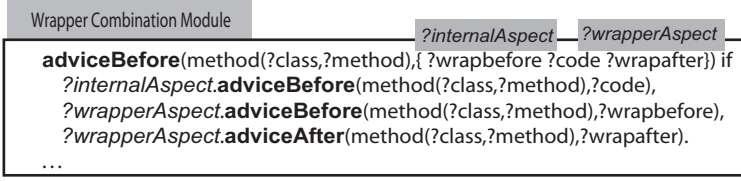


Fig. 11. The Wrapper combination module to wrap an aspect’s advices around another aspect’s advices.

Completely solving problem B and A2 requires some more interacting aspects. These are explained in the following section.

5.2 Interacting Aspects

Logging when a method blocks (problem A2), is conceptually more difficult, it cannot be solved simply using the ‘dominates’ or ‘wrapper’ combination module. It requires an explicit specialization of one of the aspects to adapt to the other one. This requires knowledge about both aspects and is most easily expressed in aspect-specific terms. In our approach, we make use of such high-level declarations and define intuitive logic rules that implement an interaction.

An *aspect interaction module* is implemented as a logic module, parameterized with module-variables. The difference with combination modules is that they do not combine several aspects in one aspect but implement a dependency or interaction between aspects. In other words, they modularize a crosscutting aspect. Interaction modules contain logic rules that are triggered by one aspect and add logic declarations to the other aspect. Furthermore, interaction modules do not compose aspects. To successfully compose aspects that require an interaction, the interaction module should be used together with a combination module, as shown in figure 13.

Log methods that block. The logic module in figure 14 is an aspect interaction module that implements the desired interaction between the logging aspect and ‘order of execution’ aspect to solve problem A2. The logic rule adapts the ‘order of execution’ aspect by adding an additional **onBlock** declaration to it for each method that needs to be logged and ‘ordered’. This is specified by starting the conclusion of the logic rule with a module variable, which will be bound to the ‘order of execution’ aspect module. As such, the rule makes it’s conclusion visible in this module. The rule in figure 14 adds an **onBlock** declaration to the ‘order of execution’ aspect if the method needs to be logged and ‘ordered’.

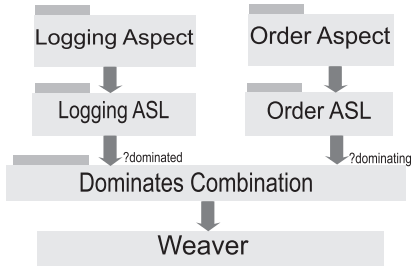


Fig. 12. Combining Logging and Order of Execution.

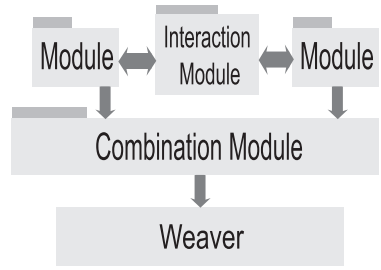


Fig. 13. Composing logic modules to implement interactions between aspects.

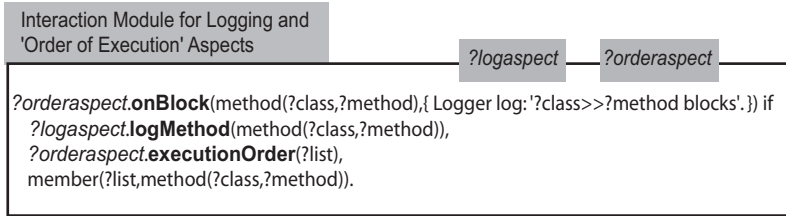


Fig. 14. Interaction to log methods when they block.

Synchronising the log. Another interaction module is required for problem B. In this example, the synchronisation aspect does not include a pointcut definition. Instead, it should fetch its pointcut definition from the logging aspect. The following logic rule could be used to implement such an interaction:

```
?syncaspect.synchronize(?method) if
  ?logaspect.logMethod(?method).
```

However, the logic rule above is too simple to tackle problem B2, which requires a more complex interaction module that also needs to interface with the ‘order of execution’ aspect. Since the log is different for each conduit and the ‘order of execution’ aspect synchronizes the **#drain:** and **#fill** of each conduit, it is safe to omit the synchronisation code of the log if logging only occurs in the critical sections of these methods. As we have seen in section 5.1, in the combination aspect for the ‘order of execution’ and logging aspects, logging code is ‘dominated’ by the ‘order of execution’ aspect’s code.

The interaction module implementing this functionality uses the rule shown in figure 15. It specifies the pointcut of the synchronisation for the logging code. This pointcut contains all methods that need to be logged under the condition that all these methods are not a subset of the methods that are wrapped with the ‘order of execution’ aspect. Indeed, if it would be a subset, the pointcut is empty because in that case synchronisation of the log is already done by the ‘order of execution’ aspect.

To completely solve problem B, the interaction module should also be used with the dominates and the wrapper combination modules (from section 5.1), using the composition structure as shown in figure 16.

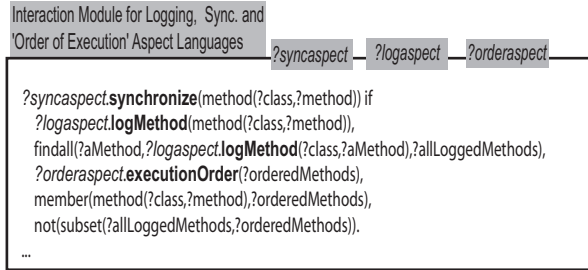


Fig. 15. Interaction to reduce synchronisation overhead.

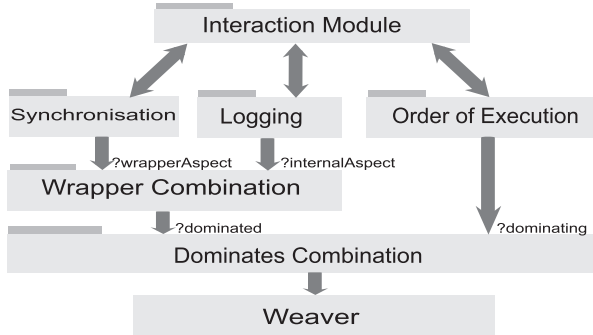


Fig. 16. Composition of logic modules to solve problem B.

6 Tool Support

The Soul/Aop system [2] is a prototype aspect-weaver that implements our logic metaprogramming approach to AOP in Smalltalk. It provides a hard-coded basic aspect language on which we can build our own ASLs using the techniques explained in this paper. The experiment in this paper was conducted using Soul/Aop.

The basic aspect language (table 4) supports wrapping of methods with aspect code as well as the definition of aspect-instance variables. Furthermore, the aspect code (defined in `wrap` declarations) can contain two special keywords

(`original`, `thisObject`) that respectively allow access to the wrapped method and the executing object with which the aspect is woven.

The weaver itself consists of two parts: the logic inference engine and the low-level (Smalltalk) weaver. This low-level weaver launches queries to gather the logic declarations written in the basic aspect language and generates the appropriate Smalltalk constructs to produce the required behaviour described by these declarations. We can also say that the low-level weaver actually is the kernel of the Soul/Aop aspect-weaver and that the logic inference engine weaves the extensions described by the logic declarations in the various logic modules together and transforms them into the basic aspect language.

Table 4. Basic SOUL/Aop aspect language.

Predicate	Description
<code>wrap(?m,?code)</code>	Wrap/shadow the method <code>?m</code> with <code>?code</code>
<code>instvars(?list,?scope)</code>	Declares a list <code>?list</code> of aspect-instance variables of which the scope is defined as <code>?scope</code> . How this scope is specified is out of the scope of this paper (see [2]).

7 Future Work

Although the experiment with the conduit simulator is rather small, it was chosen specifically to illustrate how the LMP approach can be used to implement composable ASLs. This approach will now be used to investigate the many complex and interesting problems that can arise when combining aspects as well as aspect languages.

For the reason above, the logic modules have a flexible composition mechanism, which could even be more flexible when we extend it with the ability to override predicates in a logic module. For now, the ability to express interaction issues between ASLs is limited in terms of the expressiveness of the ASLs themselves. For example, the interaction to solve problem A2 relies on the *onBlock* predicate of the ‘order of execution’ ASL. Overriding of predicates would allow an interaction aspect to change the implementation of the ASL itself. As such, an interaction module itself could also have added the *onBlock* predicate to the ‘order of execution’ ASL.

Furthermore, the LMP approach presented in this paper uses static joinpoints, which are locations in the source code. LMP has also been used to express crosscutting on a dynamic joinpoint model [5]. The issue remains open whether this joinpoint model can be easily merged with the LMP approach we discussed. We also envision more fine-grained weaving than method-level wrapping as one of the future improvements of the Soul/Aop aspect-weaver.

8 Related Work

In [4], we explained how to use logic metaprogramming as a technology to implement extensible aspect weavers. However, no means for modularization of aspects and aspect languages was discussed, nor did we address the combination and interaction of aspects and aspect languages.

AspectJ [7] is an aspect-oriented extension to Java. Aspects are written like normal java classes, extended with pointcuts and advices. The *dominates* keyword accomplishes the same as our dominates combination. However, a combination such as the wrapper combination is harder to achieve because pointcuts cannot refer to advices. The modularization of interactions between aspects (or crosscutting aspects) is not yet supported. AspectJ also features the definition of abstract aspects through the use of abstract methods and abstract pointcuts. This allows to write aspects that can be reused and adapted and hide much of the implementation from the reuser. This is somewhat similar to what ASLs accomplish. But all combinations and interactions in AspectJ need to be expressed in general-purpose terms and not in more intuitive, aspect-specific terms.

An approach to validate combinations of aspects is presented in [9]. Aspects are augmented with specifications that describe the mutual exclusiveness or dependencies with other aspects. This allows to detect or prevent some faulty combinations of aspects. The approach provides a conflict-detection mechanism, but does not discuss how conflicting aspects could be combined.

In JAC (Java Aspect Components) [17], aspects can be wrapped around objects at run time. The precedence of wrapping is addressed by an explicit composition aspect written in a general-purpose language. Other adaptations to aspects, such as the interactions we discussed, are not addressed in this technique. An advantage is that the composition aspect can use dynamic information to decide on the composition.

In [18], a number of approaches to modularize crosscutting concerns are combined in a hybrid system. This system allows a developer to use the most applicable approach for the implementation of a given concern. Interactions between the different concerns are possible because the different approaches have been integrated in a (general-purpose) object-oriented approach.

9 Conclusion

In this paper we explained how aspect-specific languages can be implemented and combined using a logic metaprogramming approach. Logic metaprogramming provides a uniform and intuitive mechanism that reconciles the ability to build aspect-specific languages with the ability to compose aspects. The common logic medium facilitates the combination and interaction of aspects written in different aspect-specific languages. Furthermore, the logic modules that govern the interactions and combinations can use aspect-specific terms, which allows an intuitive description of the desired combination and interaction of the aspects involved.

References

1. L. Bergmans, M. Aksit, and B. Tekinerdogan. Aspect composition using composition filters. In *Software Architectures and Component Technology: The State of the Art in Research and Practice*, pages 357–382. Kluwer Academic Publishers, 2001.
2. J. Brichau. Soul/Aop weaver. <http://prog.vub.ac.be/research/aop/soulaop.html>.
3. K. De Volder. Code reuse, an essential concern in the design of aspect languages? Position Paper on Workshop on Advanced Separation of Concerns at ECOOP 2001, 2001.
4. K. De Volder and T. D’Hondt. Aspect-oriented logic meta-programming. In *Meta-Level Architectures and Reflection, Second International Conference, Reflection’99*, volume 1616 of *LNCS*, pages 250–272. Springer-Verlag, 1999.
5. K. Gybels. Expressing crosscutting on a dynamic joinpoint structure using logic meta programming. Graduation thesis, Vrije Universiteit Brussel, 2001.
6. J. Irwin, J.-M. Loingtier, J. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, and T. Shpeisman. Aspect-oriented programming of sparse matrix code. In *ISCOPE*, volume 1343 of *LNCS*. Springer-Verlag, 1997.
7. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP 2001*, *LNCS*. Springer-Verlag, 2001.
8. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *ECOOP ’97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241, pages 220–242. Springer-Verlag, New York, NY, 1997.
9. H. Klaeren, E. Pulvermueller, A. Rashid, and A. Speck. Aspect composition applying the design by contract principle. In *Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering*, volume 2177 of *LNCS*, pages 57–69. Springer-Verlag, 2000.
10. C. V. Lopes and G. Kiczales. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, 1997.
11. A. Mendhekar, G. Kiczales, and J. Lamping. RG: A case-study for aspect-oriented programming. Technical Report SPL97-009P9710044, Xerox PARC, February 1997.
12. K. Mens. *Automating Architectural Conformance Checking by means of Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, 2000.
13. K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. In *Proceedings of the 13th SEKE Conference*, pages 236–243. Knowledge Systems Institute, 2001.
14. T. Mens and T. Tourwe. A declarative evolution framework for object-oriented design patterns. In *Proceedings of Int. Conf. on Software Maintenance*. IEEE Computer Society Press, 2001.
15. H. Ossher and P. L. Tarr. Hyper/J: multi-dimensional separation of concerns for Java. In *Proceedings of ICSE 2000*, pages 734–737, 2000.
16. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.
17. R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible and efficient solution for aspect-oriented programming in Java. Submitted to the Reflection Conference, 2001.

18. A. Rashid. A hybrid approach to separation of concerns: The story of SADES. In *Proceedings of the 3rd International Conference on Meta-Level Architectures and Separation of Concerns Reflection 2001*, volume 2192 of *LNCS*, pages 231–249. Springer, 2001.
19. P. L. Tarr, M. D'Hondt, L. Bergmans, and C. V. Lopes. Workshop on aspects and dimensions of concerns: Requirements on, and challenge problems for, advanced separation of concerns. In *ECOOP Workshop reader*, volume 1964 of *LNCS*, pages 203–240. Springer-Verlag, 2000.
20. R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of TOOLS-USA '98*, 1998.
21. R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.

Architectural Refactoring in Framework Evolution: A Case Study

Greg Butler

Department of Computer Science, Concordia University
Montréal H3G 1M8 Canada
`gregb@cs.concordia.ca`

Abstract. The Know-It-All Project is investigating methodologies for the development, application, and evolution of frameworks. A concrete framework for database management systems is being developed as a case study for the methodology research. The methodology revolves around a set of models for the domain, the functionality, the architecture, the design, and the code. These models reflect the common and variable features of the domain.

Refactoring of source code has been studied as a preliminary step in the evolution of object-oriented software. In cascaded refactoring, we view framework evolution as a two-step process: refactoring and extension. The refactoring step is a set of refactorings, one for each model. The refactorings chosen for a model determine the rationale or constraints for the choice of refactorings of the next model.

There are several issues with respect to architecture that we have encountered and are exploring. These include (1) the choice of models for the architecture; (2) the design of the architecture and its evaluation; (3) the evolution of the architecture by extending the concept of refactoring from source code to architecture; and (4) the modeling of variation in architectures across the product line. Here we focus on the refactoring of the architecture.

1 Introduction

Systematic reuse using domain engineering, product lines, or frameworks has been successful in delivering the economic benefits of reuse. An object-oriented framework is a concrete realization in source code of a domain-specific software architecture. Methodologies for the development of a framework have been suggested that use domain analysis, software evolution, and design patterns. However, identifying the required flexibility for the family of applications and designing mechanisms that provide this flexibility is the problem. Furthermore, the evolution of the framework must be considered, especially as all frameworks seem to mature from initial versions through to a stable platform.

The Know-It-All project [4], which has been underway at Concordia University since 1997, has three sets of aims:

1. to research methodologies and models for framework development, application, and evolution;

2. to develop a framework for database managements systems, supporting a variety of data models of data and knowledge, the integration of different paradigms, and heterogeneous databases; and
3. to apply the Know-It-All framework to advanced database applications for bioinformatics.

Refactoring of source code is one approach suggested for the development and evolution of frameworks. Refactoring as a concept or method is gaining in recognition through the publicity of extreme programming.

In earlier work [5], we introduced the concept of *cascaded refactoring* to show how refactoring as a concept can be broadened to apply to each of these models used for framework development, and not just to source code. Our view of refactoring includes the underlying motivation for the restructuring performed during the refactoring. This motivation could be expressed as an issue, rationale, or force (or sets of these). Evolution is viewed as refactoring followed by extension, but we do not address the extension step here.

The layout of the paper is to present the background on frameworks, framework evolution, and refactoring. The main part of the paper discusses architectural refactoring with a small case study. We finish by discussing related work and our conclusions.

2 Background

2.1 Frameworks

A framework exists to support the development of a family of applications. Reuse involves an application developer, or team of application developers, customizing the framework to construct one concrete application. Typically a framework is developed by expert designers who have a deep knowledge of the application domain and long experience of software design.

A framework [9] is a collection of abstract classes that provides an infrastructure common to a family of applications. The design of the framework fixes certain roles and responsibilities amongst the classes, as well as standard protocols for their collaboration. The variability within the family of applications is factored into so-called “hotspots”, and the framework provides simple mechanisms to customize each hotspot. Customizing is typically done by subclassing an existing class of the framework and overriding a small number of methods. Sometimes, however, the framework insists that the customization preserves a protocol of collaboration between several subclasses, so customization requires the parallel development of these subclasses and certain of their methods.

Methodologies for the development of a framework have been suggested that use domain analysis, software evolution, and design patterns. However, identifying the required flexibility for the family of applications and designing mechanisms that provide this flexibility is the problem. Furthermore, the evolution of the framework must be considered, especially as all frameworks seem to mature from initial versions through to a stable platform.

2.2 Framework Life Cycle

A framework evolves through several stages of maturity [16] as the developers increase their understanding of the domain and the required customizations:

- White-box framework: the application developer creates subclasses in order to customize the framework and needs to look at the code of the abstract classes being subclassed.
- Component library: many concrete subclasses are available for selection by the application developer, so much fewer new subclasses are created during customization.
- Pluggable object: extensive use is made of delegation and there are concrete subclasses available to serve as the targets of delegation, so the application developer can customize by parameterizing subclasses.
- Fine-grained object: the role or functionality of classes is decomposed into smaller classes, allowing more mix-and-matching of pluggable objects because of their finer granularity.
- Black-box framework: the application developer does not need to look at the code internals in order to customize, instead existing fine-grained components are selected and composed.
- Visual builder: the choice of components and their composition is done using drag-and-drop in a graphical interface.

2.3 Refactoring

Evolution is naturally viewed as refactoring followed by extension in those bottom-up methodologies for framework development that are centered on refactoring. The new structure of the system and the refactorings to effect the restructuring are chosen with the required extension in mind. However, there is little literature on this connection between refactoring and extension. We also say very little in this paper. A useful reference on extension of object-oriented systems is Mätzel and Bischofberger [11].

Refactoring [19] is a behavior-preserving program transformation that automatically updates an application's design and underlying source code. Primitive refactorings perform simple edits such as adding new classes, creating instance variables, and moving instance variables up the class hierarchy. Compositions of refactorings can create abstract classes, capture aggregation and components [14], and even install design patterns [19].

Opdyke [14] introduced the term refactoring and defined a set of behavior preserving transformations for object oriented applications based on the work by Banerjee and Kim [1], the design principles of Johnson and Foote [9], and the design history of the UIUC Choices software system [12]. Roberts [15] developed the Smalltalk Refactory Browser which implements many of these refactorings.

Tokuda and Batory [17] proposed additional refactoring to support design patterns as target states for software restructuring efforts. Extreme programming, and Fowler's book [7] have increased the visibility of refactoring of source

code within the community. Tokuda [18] divides the kinds of architectural changes offered by source code refactoring of object oriented systems into:

- schema transformations, which change class structure;
- introduction of design patterns as micro-architectures; and
- hotspot generalization.

John McGregor and his colleagues [13] have developed *use case assortment* for the requirements analysis phase of framework development. The use case model is factored by introducing abstract actors and abstract use cases, and re-arranging responsibilities. This allows the use case model to reflect the commonality/variability analysis. This work struck us as the germ of refactoring of use case models, similar to the refactoring of a class hierarchy, and inspired our work on cascaded refactoring.

2.4 Cascaded Refactoring

We introduced the concept of *cascaded refactoring* [5] as a methodology for framework development and evolution. It extends the notion of refactoring from source code to all the models that we use to describe a framework: feature model, architecture, design, and source code. The methodology is use case driven and issue-driven, so cascaded refactoring views refactoring as an issue-driven activity where the rationale for the decision to apply a particular refactoring is documented as a triple: intent of restructuring, choice of refactoring(s), and impact of the restructuring.

The original paper provides illustrative examples of refactorings of each of the models, and provides an adequate treatment of feature model refactorings. Since then we have made substantial progress on refactoring of use case models, which is the PhD topic of Kexing Rui. Our work on refactoring architectures is still preliminary, but we advance on our earlier work with a concrete case study in this paper.

3 The Know-It-All Framework

Our case study, Know-It-All [4] is an object-oriented framework for database management systems. It is written in C++, with some Java for user interfaces, and XML for communication of data between the C++ framework and the Java tools. The user interfaces will provide a full range of query mechanisms, from icons for canned queries, to forms, to textual queries in set comprehension languages, and diagrammatic queries. Know-It-All is designed with scientific databases in mind, and does not provide for transactions. Instead, it provides a data feed mechanism for bulk or incremental data loads. The prime concern is querying of existing data. The framework provides a generic infrastructure for database management systems and allows them to support a range of data models (relational, object, object-relational, etc) where the data model itself, and its constituents for query language, query optimizing, indexing, and storage have clearly defined roles.

4 Modeling the Architecture

Software architectures provide an abstract description of the organisational and structural decisions that are evident in a software system. The development of an architecture requires the decomposition of system into subsystems, the distribution of control and responsibility, and the development of the components and their connections or means of communication. General principles of information hiding, such as the use of modules, layers, and abstract machines (API's) help manage the complexity of the task.

We have adopted the Siemens approach [8] which consists of four views modeled in terms of UML. The conceptual view presents the configuration of components and connectors. The module view shows the structure in terms of layers, subsystems, modules and their interfaces. The execution view identifies the hardware resources, communication mechanisms and paths, and the runtime entities such as processes. The code view presents the organisation of source code, libraries, binaries, and executables. Their description of the meta-models together with the stereotypes and notations that extend UML make the Siemens approach much more precise than that of the Rational Unified Process (RUP).

The traditional architecture of a database management system as depicted in most textbooks (see [10, page 17]) is a module diagram that follows a functional decomposition of the system. The main data structure is a monolithic “system catalog” holding all information about users, databases, and their schemas (but not the data per se).

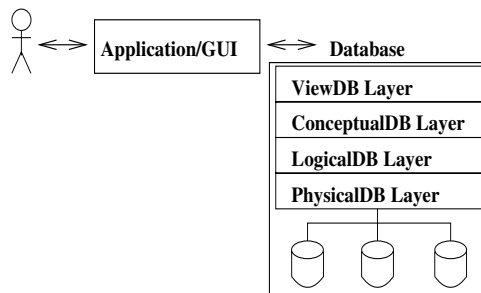


Fig. 1. Conceptual Architecture of a Database

In the Know-It-All case study we want to take a more object-oriented approach and explore reuse across different kinds of data models. A database management system in Know-It-All is a container of databases. A database in Know-It-All is seen as a series of layers, each of which provides the same interface. The usual breakdown of responsibilities into physical, logical, conceptual, and view layers is followed by Know-It-All. However, a database, as seen by the end-user, allows views of views, and mappings of object conceptual models to relational conceptual models. Eventually, Know-It-All will incorporate com-

posite databases (such as integrated or heterogeneous databases) and make no distinction between simple and composite databases.

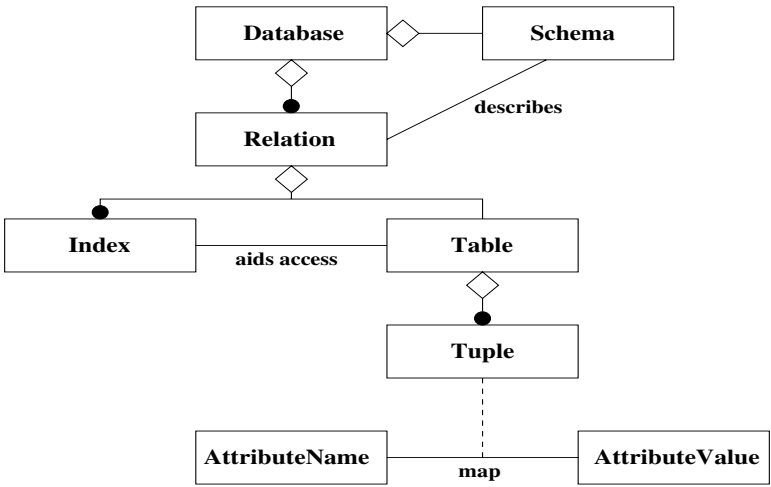


Fig. 2. Conceptual Class Diagram for Relational Database

The conceptual view of a database as seen by an application or user is shown in Figure 1. The class diagram for the conceptual model of the relational data model is shown in Figure 2, while the context for the DBMS in relation to the system is seen in Figure 3.

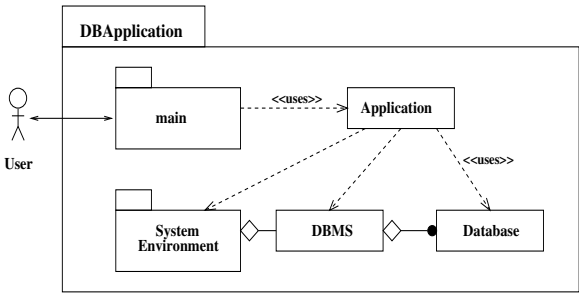


Fig. 3. Architectural Context

The Know-It-All framework contains subframeworks for query optimization and for indexing, and eventually will have a subframework for physical storage. Figure 4 shows how the different data models plug into the framework and how the subframeworks can be customized to be specific to the data model. The

impact of these is internal to a layer, so to date we have found it sufficient to use a module view of the architecture presenting layers, subsystems, modules, and classes.

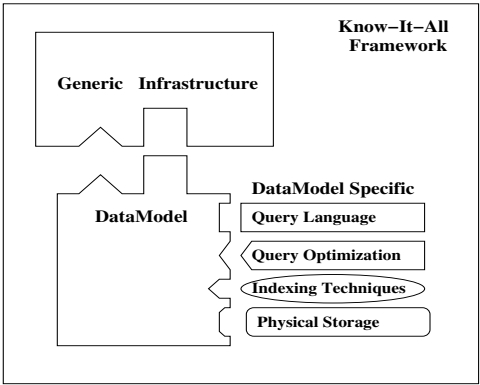


Fig. 4. Framework Plug-and-Play View

Each layer in Know-It-All is basically a translator between its client layer and its supplier layer(s), as shown in Figure 5. A layer provides a mechanism to decompose or translate queries, and a mechanism to reconstruct answers (for example, an execution plan for relational algebra expressions). The translation is done with the aid of the schema, and produces both the translated query, and the mechanism to reconstruct answers. The layer architecture is adapted from one for heterogeneous databases, while the reconstruction mechanism is designed as an iterator (which is a tree of iterators) that returns the next result.

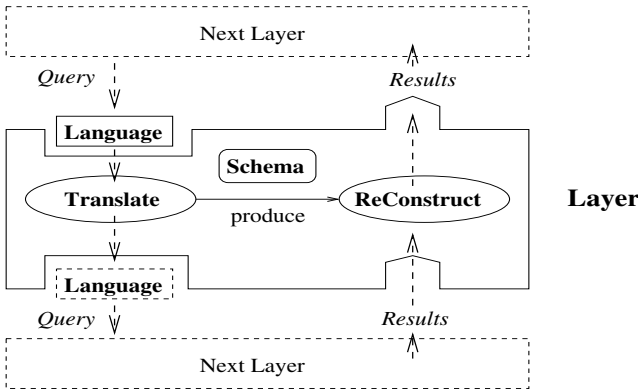


Fig. 5. Basic Building Block

We need to model variation in architecture for our case study. The major variability such as different data models, different query optimization strategies, different choices of indexes or physical storage are covered within the subframeworks or within the selection of layers. What has not been dealt with is the runtime variation. To date we have not done this in a systematic way other than listing the main variants individually:

Client-Server with network connection between the client front-end and the server back-end;

Three-Tier Client-Server where the back-end server splits off the database server from the network server;

Embedded in Application where the DBMS is linked with the application and access data on the file system;

Embedded In-Memory in Application where the data is stored in-memory;

Distributed Database where the physical data storage is distributed across several platforms;

Integrated Database where an application is accessing a number of databases (and DBMS's) which may be on different platforms and views them as one; and

Parallel Database where one or more components of the DBMS (such as the query execution or indexing) may be executing on a multiple cpu node or cluster.

These variants require us to specify connectors between certain components or layers since the connector might be a remote procedure call over TCP/IP for a client-server system, or a normal procedure call for an embedded database, or broadcast in a parallel database. These variants also require us to indicate the deployment of components across processes and processors.

5 Architectural Refactoring in Action

Our work on refactoring of architectures focusses on the traditional subsystems and interfaces view of an architecture. It is possible to take a use case view of a subsystem, where the subsystem is regarded as the target system with the host system consisting of all other subsystems in the architecture. Then a subsystem's services are described in the use case model for the subsystem, and accessed through its interfaces. It is also possible to take a class view of a subsystem, where one identifies a subsystem with a facade class. One identifies the subsystem interfaces with the class methods.

5.1 The Architecture Refactorings

The first set of refactorings [5] looks at the interfaces of a subsystem and redistribute their methods.

Split_interface takes an interface I of a subsystem S and redistributes the methods of I across two new interfaces I_1 and I_2 of the subsystem S .

Merge_interfaces is the inverse transformation.

The second set of refactorings [5] re-distribute the services provided by a subsystem. Usually these are accompanied by a re-distribution of interfaces.

Move_service_to_sibling takes a service s of subsystem S_1 with sibling subsystem S_2 in a hierarchical client-supplier architecture and assigns it to subsystem S_2 .

Delegate_service_to_supplier takes a service s of subsystem S_1 with a supplier subsystem S_2 and assigns the service to S_2 .

Promote_service_from_internal takes as service s provided by a nested subsystem S_2 of a subsystem S_1 and makes it a service of S_1 .

To this list of refactorings one can add those that introduce design patterns [19].

5.2 A Case Study in Evolution

Let us look at a situation that arises in the evolution of our architecture from the traditional monolithic API such as JDBC or ODBC and supported by the breadth of functionality in the SQL language used by those APIs. In Figure 6 we see the subsystems and interface view of the DBMS and Database as seen by an application. There is a single interface I_DBMS that offers all the services of a DBMS as realized by SQL. This includes

1. connection and disconnection from the database;
2. the definition of database tables and views;
3. the querying of data in the database; and
4. manipulations and transactions on the data itself.

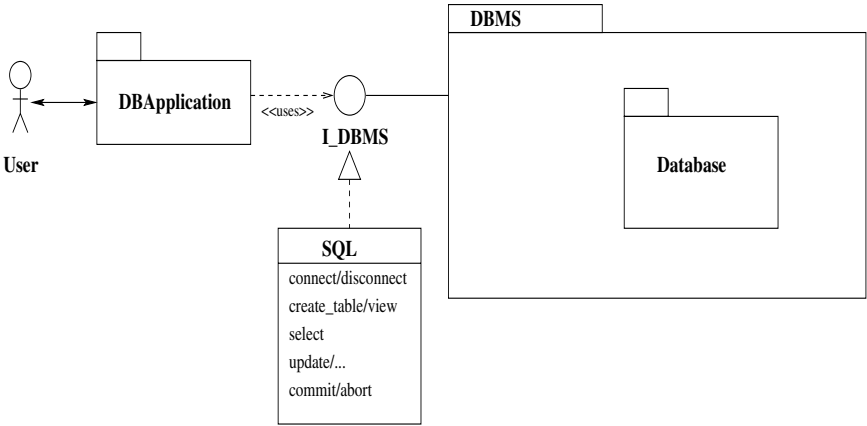


Fig. 6. Subsystems and Interfaces: I

Figure 7 shows the architecture after the decision to split the interface so that connection and disconnection are clearly indicated to be the responsibility of the DBMS itself, whereas other functions are the responsibility of the Database.

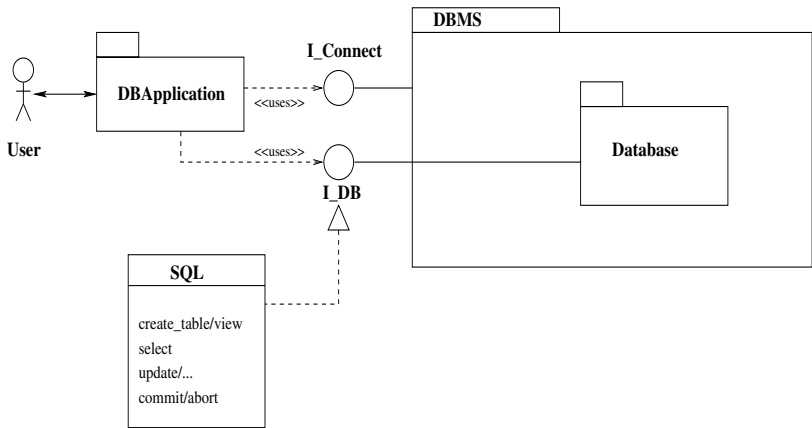


Fig. 7. Subsystems and Interfaces: II

Figure 8 shows the architecture after further refactoring that shows that it is the responsibility of the **Transaction Manager** to deal with transactions, and that the functions of database creation such as the definition of tables and views is (a) the job of the Database Administrator (DBA), and (b) the responsibility of the DBMS and not the Database.

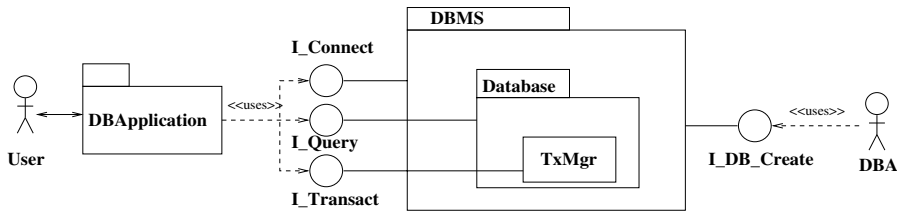


Fig. 8. Subsystems and Interfaces: III

6 Related Work

There are several categories of methodologies for the development of object-oriented frameworks [6]. Bottom-up methodologies follow an incremental spiral generalization from a small number of applications into a framework [16]. Refactoring of source code is a key step. Top-down methodologies borrow heavily

from domain analysis, and generally say very little about evolution. They do not mention refactoring. There are also several use case driven methodologies for framework development. John McGregor and his colleagues [13] have developed *use case assortment* for the requirements analysis phase of framework development. The use case model is factored by introducing abstract actors and abstract use cases, and re-arranging responsibilities. This allows the use case model to reflect the commonality/variability analysis. This work struck us as the germ of refactoring of use case models, similar to the refactoring of a class hierarchy, and inspired our work on cascaded refactoring.

The Design Maintenance System (DMS) [2,3] is a transformation system that is rule-based. It works with a hierarchy of domains, each specified by a syntax, semantics, and mappings to other domains (or to the same domain). DMS can implement source code transformations: DMS has been used for transformation of COBOL programs for the removal of duplicate code and dead code. In general, the DMS transformations do not have to be behaviour-preserving. It is feasible within DMS to define a domain corresponding to each architectural models, and to define a set of transformations for the models. Thus, refactoring could be realized within DMS. However, to the best of our knowledge, this has not been done.

7 Conclusion

Refactoring of source code has long been used for bottom-up development and evolution of object-oriented frameworks. In this paper, we present an example of architectural refactoring drawn from the evolution of our database framework. This is part of a larger investigation into the methodology of cascaded refactoring for the development and evolution of frameworks.

Our work is ongoing, particularly in terms of enumerating all the refactorings of the particular models, and in investigating whether there is a need for architectural refactorings that preserve quality attributes other than functionality.

Acknowledgements. This work has been supported by the Natural Sciences and Engineering Research Council of Canada, and *Fonds pour la Formation de Chercheurs et l'Aide à la Recherche* of Québec.

References

1. J. Banerjee and W. Kim. *Semantics and implementation of schema evolution in object-oriented databases*. In Proceedings of the ACM SIGMOD Conference, 1987.
2. I. D. Baxter, *Design maintenance systems*, Communications of the ACM, 35(4):73–89, 1992.
3. I. D. Baxter, *DMS (transformational software maintenance by reuse): A production research system?*, Proceedings of the Fifth Symposium on Software Reusability, ACM, 1999, p. 163.

4. G. Butler, L. Chen, X. Chen, A. Gaffar, J. Li, L. Xu. The Know-It-All Project: A Case Study in Framework Development and Evolution. In *Domain Oriented Systems Development: Perspectives and Practices*, K. Itoh and S. Kumagai (eds). Gordon Breach Science Publishers, UK, 2002, to appear.
5. G. Butler and L. Xu. Cascaded refactoring for framework evolution. In *Proceedings of 2001 Symposium on Software Reusability*, ACM Press, 2001, pp. 51–57.
6. M. E. Fayad, D. C. Schmidt, and R. E. Johnson, editors. *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. Wiley, 1999.
7. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
8. C. Hofmeister, R. Nord, D. Soni. *Applied Software Architecture*. Addison-Wesley, 1999.
9. Ralph E. Johnson and B. Foote. *Designing reusable classes*. Journal of Object-Oriented Programming, pp. 22–35, July 1988.
10. H.F. Korth and A. Silberschatz. *Database System Concepts*. McGraw-Hill, 1988.
11. Kai-Uwe Mätzel, Walter Bischofberger. *Designing object systems for evolution*. TAPoS, Vol. 3, No. 4, 1997.
12. P. W. Madany, R. H. Campbell, V. Russo, and D. E. Leyens. *A class hierarchy for building stream-oriented file systems*. Proceedings of ECOOP'89 (Nottingham, UK, July 1989), pp. 311–328.
13. G. G. Miller, J. D. McGregor and M. L. Major. *Capturing framework requirements*. in [6].
14. W.F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois, 1992.
15. D. Roberts, J. Brant and R. Johnson. *A refactoring tool for Smalltalk*. In Theory and Practice of Object Systems, Vol. 3, No. 4, 1997.
16. D. Roberts and R. Johnson. *Patterns for evolving frameworks*. In R. C. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design 3*, pages 471–486. Addison-Wesley, 1997.
17. Lance Tokuda and Don Batory. *Automating software evolution via design pattern transformations*. In 3rd International Symposium on Applied Corporate Computing, Monterrey, Mexico, Oct. 1995.
18. Lance Tokuda and Don Batory. *Automating three modes of evolution for object-oriented software architectures*, 5th USENIX Conference on Object-Oriented Technologies, (COOTS '99), May 1999.
19. Lance Tokuda. *Evolving object-oriented architectures with refactorings*. Proceedings of ASE-99: The 14th Conference on Automated Software Engineering, IEEE CS Press, October 1999.
20. D. M. Weiss and C. T. R. Lai. *Software Product-Line Engineering*. Addison-Wesley, 1999.

Towards a Modular Program Derivation via Fusion and Tupling

Wei-Ngan Chin^{1,2} and Zhenjiang Hu^{3,4}

¹ National University of Singapore

² Singapore-MIT Alliance

`chinwn@comp.nus.edu.sg`

³ University of Tokyo

⁴ Presto 21, Japan Science and Technology Corporation

`hu@ipl.t.u-tokyo.ac.jp`

Abstract. We show how programming pearls can be systematically derived via fusion, followed by tupling transformations. By focusing on the elimination of intermediate data structures (fusion) followed by the elimination of redundant calls (tupling), we systematically realise both space and time efficient algorithms from naive specifications. We illustrate our approach using a well-known maximum segment sum (MSS) problem, and a less-known maximum segment product (MSP) problem. While the two problems share similar specifications, their optimised codes are significantly different. This divergence in the transformed codes do not pose any difficulty. By relying on modular techniques, we are able to systematically reuse both code and transformation in our derivation.

1 Introduction

A major impetus for highlighting programming pearls is to better understand how elegant and efficient algorithms could be invented. While creative algorithms are interesting to exhibit, they often lose their links to the programming techniques that were employed in their discoveries.

A more motivated approach to programming pearls is to formally derive creative algorithms from naive specifications. While elegant, most program derivations require deep insights to obtain major efficiency jumps for the transformed code. This can make it particularly difficult for human to comprehend, and machine to implement. In this paper, we show that it is possible to minimise some of these insights, and provide a systematic and modular approach towards discovering programming pearls.

Consider the maximum segment product problem. Given a list $[x_1, \dots, x_n]$, we are interested to find the maximum product of all non-empty (contiguous) segments (of the form $[x_i, x_{i+1}, \dots, x_j]$ where $1 \leq i \leq j \leq n$) taken from the input list. An initial specification for this problem can be written, as follows:

$$\text{msp}(xs) = \max(\text{map}(\text{prod}, \text{segs}(xs)))$$

As defined below, the innermost *segs* call returns a complete list of all segments, while the *map* call applies *prod* to each segment to yield its product, before the outermost *max* call chooses the largest value.

$segs([x])$	$= [[x]]$
$segs(x:xs)$	$= inits(x:xs) ++ segs(xs)$
$inits([x])$	$= [[x]]$
$inits(x:xs)$	$= [x]:map((x:), inits(xs))$
$map(f, Nil)$	$= Nil$
$map(f, x:xs)$	$= f(x):map(f, xs)$
$prod([x])$	$= x$
$prod(x:xs)$	$= x * prod(xs)$
$max([x])$	$= x$
$max(x:xs)$	$= max2(x, max(xs))$
$max2(x, v)$	$= \text{if } v > x \text{ then } v \text{ else } x$

The above specification uses modular and reusable coding. Through the use of functions, such as *segs*, *inits*, *map*, *max* and *prod*, we can specify the *mss* function via straightforward composition of simpler functions. These high level specification are easier to comprehend and more reusable. For example, the better known maximum segment sum problem [Ben86] can be specified by replacing the *prod* call with *sum*, as follows:

$mss(xs)$	$= max(map(sum, segs(xs)))$
$sum([x])$	$= x$
$sum(x:xs)$	$= x + sum(xs)$

Unfortunately, high-level specifications have one major drawback, namely that they may be terribly inefficient. Fortunately, it is possible to use transformation to calculate efficient algorithms, that are usually unintuitive.

Our thesis is that high-level transformation techniques can provide a systematic approach to discovering programming pearls with good time and space behaviours. To substantiate this claim, we propose to apply two key transformation techniques, namely (i) *fusion* enhanced with laws, and (ii) *tupling*. The insights needed by our derivation are mainly confined to the fusion technique, in the form of laws needed to facilitate its transformation.

To appreciate the virtues of the transformational approach, the reader may want to try invent an efficient algorithm for maximum segment product, before studying the rest of this paper. We had some difficulties, until we embark on the transformational approach.

The main contributions in this paper are:

- We propose a *modular* derivation that supports the reuse of codes and transformation techniques. Particularly, we highlight two important transformation techniques, fusion and tupling, which in combination can be surprisingly good for deriving efficient algorithms.
- Our derivation is more *systematic*, minimizing the use of complex laws with deeper insights, such as Horner’s rule in [Bir89]. Instead, we use a set of smaller laws which are motivated by the need to make fusion succeed. Most of these laws are *distributive* in nature.
- Our derivation is *powerful*. To the best of our knowledge, we demonstrate the first full and systematic derivation for the maximum segment product problem, without “suitable cunning” used in a previous derivation [Bir89].

- We show how *accumulation transformation*, known to invalidate *tupling* method, can be avoided.

In the rest of this paper, we first outline an enhanced fusion technique, which depends on laws, for its transformation (Sec 2). Later, we apply our modular approach, based on fusion and tupling, to a well-known maximum segment sum problem (Sec 3). We also highlight how a related but little known problem, called maximum segment product, can be similarly derived by our approach (Sec 4). We then compare with a classical derivation via Horner's rule (Sec 5), before an advice on the use of *accumulation* technique (Sec 6).

2 Enhanced Fusion with Laws

Fusion method [Chi92,TM95,CK01] is potentially a useful and prevalent transformation technique. Given a composition $f(g(x))$ where $g(x)$ yields an intermediate data structure for use by f , fusion would attempt to merge the composition into a specialised function $p(x)$ with the same semantics as $f(g(x))$ but without the need for an intermediate data structure.

In recent years, many attempts have been put forward to automate such fusion calculations [SF93, GLPJ93, SS97, HIT96]. Most current attempts are restricted to using equational definitions during transformation. For example, the deforestation algorithm [Wad88] relies on only define, unfold and fold rules [BD77] base on equational definitions. Unfortunately, this approach is inadequate since we often need laws (useful properties between functions, such as associativity and distributivity) to apply fusion successfully.

Consider a program which flattens a tree into a list, before finding its size.

$$\begin{aligned}
 \text{sizetree}(t) &= \text{length}(\text{flattree}(t)) \\
 \text{flattree}(\text{Leaf}(a)) &= [a] \\
 \text{flattree}(\text{Node}(l,r)) &= \text{flattree}(l) ++ \text{flattree}(r) \\
 \text{length}(\text{Nil}) &= 0 \\
 \text{length}(x:xs) &= 1 + \text{length}(xs)
 \end{aligned}$$

To optimise this program, we could try to fuse $\text{length}(\text{flattree}(t))$. However, this cannot be done without the distributive law of length over $++$.

$$\text{length}(xr ++ xs) = \text{length}(xr) + \text{length}(xs)$$

With this law, the fusion derivation of *sizetree* can be carried out, as follows:

$$\begin{aligned}
 \text{sizetree}(\text{Leaf}(a)) &= \{ \text{instantiate } t = \text{Leaf}(a) \} \\
 &\quad \text{length}(\text{flattree}(\text{Leaf}(a))) \\
 &= \{ \text{unfold } \text{flattree}, \text{ then } \text{length} \} \\
 &\quad 1 \\
 \text{sizetree}(\text{Node}(l,r)) &= \{ \text{instantiate } t = \text{Node}(l,r), \text{ unfold } \text{flattree} \} \\
 &\quad \text{length}(\text{flattree}(l) ++ \text{flattree}(r)) \\
 &= \{ \text{apply law (2) : } \text{length}(xr ++ xs) = \text{length}(xr) + \text{length}(xs) \} \\
 &\quad \text{length}(\text{flattree}(l)) + \text{length}(\text{flattree}(r)) \\
 &= \{ \text{fold with } \text{sizetree} \text{ twice} \} \\
 &\quad \text{sizetree}(l) + \text{sizetree}(r)
 \end{aligned}$$

What was the rationale for using a distributive law during the above fusion? Informally, the inner *flattree* produces $++$ calls during unfolding, which cannot be consumed by the pattern-matching equations of the outer *length* function. Instead, we need the distributive law of *length* over $++$ to successfully consume $++$ calls from the inner *flattree* function. A more detailed description of how laws may help fusion can be found in [Chi94].

These needed laws must either be supplied by programmers, or be derived via advanced synthesis techniques, such as [Smi89,CT97]. There are scope for automated help to synthesize (or check) these laws, but this issue is beyond the scope of the present paper. In the rest of this paper, we shall assume that relevant laws will be provided by users.

3 A Modular Derivation Strategy

We propose a modular derivation strategy based on two key transformation, namely fusion and tupling. To illustrate this strategy, consider:

$$mss(xs) = \max(\text{map}(\text{sum}, \text{segs}(xs)))$$

The above specification has bad time and space complexities. If n is the size of the input list, then *mss* has a time complexity of $O(n^3)$. Note that *segs* returns $O(n^2)$ sub-lists which each requires $O(n)$ time to process by *sum*.

In general, space usage can be broken down into three parts:

- stack space for the function calls (such as *segs*, *map*, *sum*).
- heap space for input and output of main function (i.e. *mss*).
- heap space for intermediate data structures (by *segs*, *map* and *sum*).

We shall ignore the somewhat *fixed* space cost associated with the stack and input/output, but focus on the *variable* space cost due to intermediate data structures. In the case of *mss*, this space cost is due to *segs* generating $O(n^2)$ sub-lists of $O(n)$ length each, while *map* yields another intermediate list of size $O(n^2)$; giving a space complexity of $O(n^3)$.

Our strategy for deriving efficient algorithms via *fusion*, followed by *tupling* is outlined in Figure 1 for the MSS problem.

Fusion transformation is capable of eliminating all intermediate data structures for this example. During this fusion, we encountered another composition which was defined as the following new definition:

$$mis(xs) = \max(\text{map}(\text{sum}, \text{inits}(xs)))$$

With the help of appropriate laws, both *mss* and *mis* functions can be transformed to a pair of new recursive functions, shown in Figure 1(b). The fused *mss* has a much improved $O(1)$ variable space complexity. However, it still suffers from a time-complexity of $O(n^2)$ due primarily to redundant *mis* calls. The redundant calls can be eliminated by tupling transformation. Firstly, define:

$$mss\text{stup}(xs) = (\text{mss}(xs), \text{mis}(xs))$$

Subsequent transformation yields a new recursive tupled definition shown in Figure 1(c). Without any redundant calls, the new *mssstup* definition has a time-complexity of $O(n)$. We present the detailed derivations next.

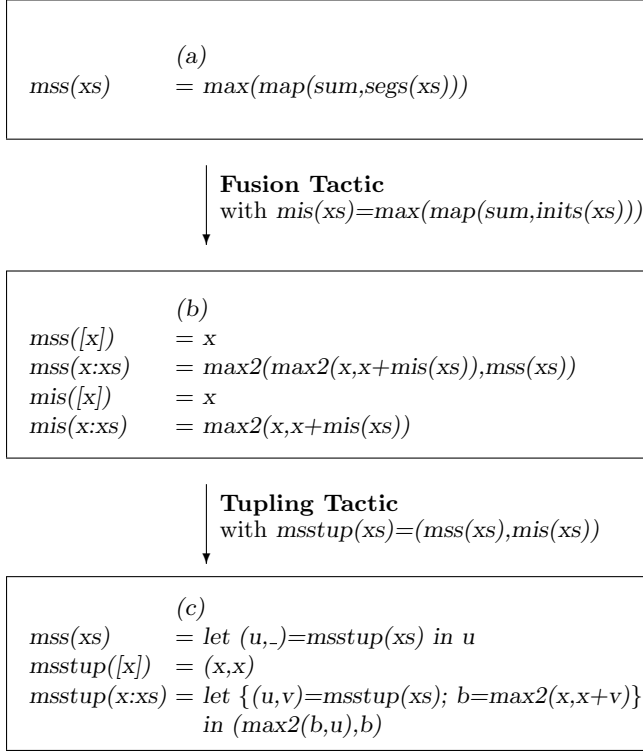


Fig. 1. Modular Derivation Strategy via Fusion and Tupling

3.1 Fusion to Remove Intermediate Data Structures

The enhanced fusion method relies on laws, in addition to the supplied equation, for its transformation. We would like to stress again that these laws do not come from thin air, but are instead motivated by the need to perform fusion. In the case of *mss*, we need the following *distributive* laws.

$$\text{map}(f, xr ++ xs) = \text{map}(f, xr) ++ \text{map}(f, xs) \quad (1)$$

$$\text{max}(xr ++ xs) = \text{max2}(\text{max}(xr), \text{max}(xs)) \quad (2)$$

$$\text{map}(f, \text{map}(g, xs)) = \text{map}(f \circ g, xs) \text{ where } (f \circ g)(x) = f(g(x)) \quad (3)$$

$$\text{max}(\text{map}(x+, xs)) = x + \text{max}(xs) \quad (4)$$

The first two laws are distributive laws of *map* and *max* over the *++* operator, while law (3) distributes over an inner *map* call (or over function composition if used backwards). The last law is concerned with the distributivity of *max* over

an $(x+)$ call that is being applied to each element of its input list. A more general version of this last law can be constructed in conjunction with law (3), as follows:

$$\max(\text{map}((x+) \circ g, xs)) = x + \max(\text{map}(g, xs)) \quad (5)$$

Fusion/deforestation method makes use of normal-order transformation strategy [SGN94], to merge functional compositions. In the case of *mss*, the outermost *max* call demands an output from an inner *map* call, which in turn demands an output from *segs*. Thus, the innermost *segs(xs)* call is selected for unfolding. This can be done via two possible instantiations to its argument, *xs*. The base case instantiation results in:

$$\begin{aligned} \text{mss}([x]) &= \{ \text{instantiate } xs=[x] \} \\ &\quad \max(\text{map}(\text{sum}, \text{segs}([x]))) \\ &= \{ \text{unfold } \text{segs}, \text{map}, \text{max}, \text{sum} \} \\ &\quad x \end{aligned}$$

For the recursive case instantiation, the *segs* function actually produces $++$ calls which must be consumed by *map*, as follows:

$$\begin{aligned} \text{mss}(x:xs) &= \{ \text{instantiate } xs=x:xs, \text{unfold } \text{segs} \} \\ &\quad \max(\text{map}(\text{sum}, \text{inits}(x:xs) ++ \text{segs}(xs))) \\ &= \{ \text{apply law (1) : } \text{map}(f, xr ++ xs) = \text{map}(f, xr) ++ \text{map}(f, xs) \} \\ &\quad \max(\text{map}(\text{sum}, \text{inits}(x:xs)) ++ \text{map}(\text{sum}, \text{segs}(xs))) \end{aligned}$$

Another $++$ operator is produced by the distributive law of *map* itself. This must in turn be consumed via the distributive law of *max*, as follows:

$$\begin{aligned} \text{mss}(x:xs) &= \{ \text{apply law (2) : } \max(xr ++ xs) = \max2(\max(xr), \max(xs)) \} \\ &\quad \max2(\max(\text{map}(\text{sum}, \text{inits}(x:xs))), \max(\text{map}(\text{sum}, \text{segs}(xs)))) \end{aligned}$$

At this point, $\max(\text{map}(\text{sum}, \text{segs}(xs)))$ is a re-occurrence of the definition for *mss* which can be handled using a fold operation. Also, $\max(\text{map}(\text{sum}, \text{init}(xs)))$ represents a new composed expression just encountered. We could introduce a new function, say *mis*, to denote it and then obtain:

$$\begin{aligned} \text{mss}(x:xs) &= \{ \text{fold with } \text{mss} \} \\ &\quad \max2(\max(\text{map}(\text{sum}, \text{inits}(x:xs))), \text{mss}(xs)) \\ &= \{ \text{fold with a new } \text{mis}, \text{then unfold} \} \\ &\quad \max2(\max2(x, s + \text{mis}(xs)), \text{mss}(xs)) \end{aligned}$$

The new composition encountered is captured by:

$$\text{mis}(xs) = \max(\text{map}(\text{sum}, \text{inits}(xs)))$$

We re-apply fusion transformation, by beginning with an unfold of *inits(xs)* using the two possible instantiation to *xs*. A similar sequence of transformations via unfolding, application of laws, and folding yield the following equations.

$$\begin{aligned} \text{mis}([x]) &= x \\ \text{mis}(x:xs) &= \max2(x, x + \text{mis}(xs)) \end{aligned}$$

The primary gain from fusion method is the complete elimination of intermediate data structures. This results in an improved time complexity of $O(n^2)$, and a much improved variable space complexity of $O(1)$.

3.2 Tupling to Eliminate Redundant Calls

After fusion, our program may have redundant function calls. This inefficiency can be overcome by the tupling method [Chi93,HITT97] which essentially gathers calls with overlapping arguments together. In the case of *mss*, we find two calls with identical arguments in its recursive equation. Tupling would gather these two calls, as follows.

$$msstup(xs) = (mss(xs), mis(xs))$$

This can then be instantiated and further transformed, as follows:

$$\begin{aligned} msstup([x]) &= \{ \text{instantiate } xs=[x] \} \\ &\quad (mss([x]), mis([x])) \\ &= \{ \text{unfold } mss \text{ \& } mis \} \\ &\quad (x, x) \end{aligned}$$

The recursive case instantiation and transformation is outlined below.

$$\begin{aligned} msstup(x:xs) &= \{ \text{instantiate } xs=x:xs \} \\ &\quad (mss(x:xs), mis(x:xs)) \\ &= \{ \text{unfold } mss \text{ \& } mis \} \\ &\quad (max2(max2(x, x+mis(xs)), mss(xs)), max2(x, x+mis(xs))) \\ &= \{ \text{gather } mss \text{ and } mis \text{ calls using let} \} \\ &\quad \text{let } (u, v) = (mss(xs), mis(xs)) \text{ in } (max2(max2(x, x+v), u), max2(x, x+v)) \\ &= \{ \text{fold with } msstup, \text{ and abstract } b \} \\ &\quad \text{let } \{(u, v) = msstup(xs); b = max2(x, x+v)\} \text{ in } (max2(b, u), b) \end{aligned}$$

Note the use of a gathering step to collect calls with overlapping arguments, resulting in a tuple of two calls. This can later be folded against *msstup*. The redundant occurrences of *mis* call was eventually shared by such a tuple gathering step. The end result is an efficient linear time $O(n)$ algorithm for maximum segment sum.

4 Maximum Segment Product

Let us now turn our attention to a related but less-known problem for finding maximum segment product (MSP). This MSP problem was proposed by Richard Bird in the 1989 STOP Summer School [Bir89]. It is of interests because its specification is closely related to the MSS problem, but yet its efficient implementation is considerably more complex.

For its specification and transformation, we can reuse all functions and laws used by *mss*, with the exception of those related to *sum* and $+$. Specifically, the distributive law of *max* over *map* needs to be replaced by corresponding laws over $(x*)$. This property can be specified by a pair of laws, namely:

$$max(map((x*), xs)) = \text{if } x \geq 0 \text{ then } x * max(xs) \text{ else } x * min(xs) \quad (6)$$

$$min(map((x*), xs)) = \text{if } x \geq 0 \text{ then } x * min(xs) \text{ else } x * max(xs) \quad (7)$$

Note the need for *min*, as a dual of *max*, with definition:

$$\begin{aligned} min([x]) &= x \\ min(x:xs) &= min2(x, min(xs)) \\ min2(x, v) &= \text{if } v < x \text{ then } v \text{ else } x \end{aligned}$$

Why is *min* needed? Consider the expression $x*b$ where b is taken from a list. If x is negative, then the value of $x*b$ would be maximal if the selected element b is of smallest value. More practical versions of the above pair of laws are obtained by combining them with law (3), as shown below.

$$\max(\text{map}((x*) \circ f, xs)) = \text{if } x \geq 0 \text{ then } x * \max(\text{map}(f, xs)) \text{ else } x * \min(\text{map}(f, xs)) \quad (8)$$

$$\min(\text{map}((x*) \circ f, xs)) = \text{if } x \geq 0 \text{ then } x * \min(\text{map}(f, xs)) \text{ else } x * \max(\text{map}(f, xs)) \quad (9)$$

With the help of these two extra laws, we can perform a similar fusion transformation on the naive specification for *msp*. Recall:

$$\text{msp}(xs) = \max(\text{map}(\text{prod}, \text{segs}(xs)))$$

The base case equation is easily derived, as follows.

$$\begin{aligned} \text{msp}([x]) &= \{ \text{instantiate } xs=[x] \} \\ &\quad \max(\text{map}(\text{prod}, \text{segs}([x]))) \\ &= \{ \text{unfold } \text{segs}, \text{map}, \max, \text{prod} \} \\ &\quad x \end{aligned}$$

The recursive case equation can be derived, as outlined below.

$$\begin{aligned} \text{msp}(x:xs) &= \{ \text{instantiate } xs=x:xs \} \\ &\quad \max(\text{map}(\text{prod}, \text{inits}(x:xs) ++ \text{segs}(xs))) \\ &= \{ \text{apply law (1) \& law (2)} \} \\ &\quad \max2(\max(\text{map}(\text{prod}, \text{inits}(x:xs))), \max(\text{map}(\text{prod}, \text{segs}(xs)))) \\ &= \{ \text{fold with } \text{msp} \} \\ &\quad \max2(\max(\text{map}(\text{prod}, \text{inits}(x:xs))), \text{msp}(xs)) \\ &= \{ \text{fold with a new defn for } \text{mip} \} \\ &\quad \max2(\text{mip}(x:xs), \text{msp}(xs)) \end{aligned}$$

A new composed expression, defined as *mip*, was encountered.

$$\text{mip}(xs) = \max(\text{map}(\text{prod}, \text{inits}(xs)))$$

Similar fusion derivation results in:

$$\begin{aligned} \text{mip}([x]) &= x \\ \text{mip}(x:xs) &= \{ \text{instantiate } xs=x:xs \text{ \& fuse} \} \\ &\quad \max2(x, \text{if } x \geq 0 \text{ then } x * \text{mip}(xs) \text{ else } x * \text{mipm}(xs)) \\ &= \{ \text{apply law (10) to float } \text{if} \text{ outwards} \} \\ &\quad \text{if } x \geq 0 \text{ then } \max2(x, x * \text{mip}(xs)) \text{ else } \max2(x, x * \text{mipm}(xs)) \end{aligned}$$

The last step floats an inner *if* out of the outermost *max2* call. This transformation can be effected by the following generic law where $E[]$ denotes an arbitrary expression context with a hole. (Its floatation can facilitate the elimination of common *if* test during tupling transformation, as shown later.)

$$E[\text{if } e_1 \text{ then } e_2 \text{ else } e_3] = \text{if } e_1 \text{ then } E[e_2] \text{ else } E[e_3] \quad (10)$$

Another composition encountered, $x * \min(\text{map}(\text{prod}, \text{inits}(xs)))$, was defined as:

$$\text{mipm}(xs) = \min(\text{map}(\text{prod}, \text{inits}(xs)))$$

Its fusion derivation for *mipm* is very similar to *mip*, and results in:

$$\begin{aligned} \text{mipm}([x]) &= x \\ \text{mipm}(x:xs) &= \text{if } x \geq 0 \text{ then } \min2(x, x * \text{mipm}(xs)) \text{ else } \min2(x, x * \text{mip}(xs)) \end{aligned}$$

Tupling analysis of [Chi93,HITT97] would reveal that there are redundant calls to *mip* and *mipm*, which can be eliminated by gathering the following tuple of calls.

$$\text{msptup}(xs) = (\text{msp}(xs), \text{mip}(xs), \text{mipm}(xs))$$

Subsequently, tupling transformation can be applied as follows:

$$\begin{aligned} \text{msptup}([x]) &= \{ \text{instantiate } xs=[x] \} \\ &\quad (\text{msp}([x]), \text{mip}([x]), \text{mipm}([x])) \\ &= \{ \text{unfold } \text{msp}, \text{mip} \text{ \& } \text{mipm} \} \\ &\quad (x, x, x) \\ \text{msptup}(x:xs) &= \{ \text{instantiate } xs=x:xs \} \\ &\quad (\text{msp}(x:xs), \text{mip}(x:xs), \text{mipm}(x:xs)) \\ &= \{ \text{unfold } \text{msp}, \text{mip}, \text{mipm} \text{ and floats if over tuple structure } \} \\ &\quad \text{if } x \geq 0 \text{ then } (\max2(\max2(x, x * \text{mip}(xs)), \text{msp}(xs)) \\ &\quad \quad \quad , \max2(x, x * \text{mipm}(xs)), \min2(x, x * \text{mipm}(xs))) \\ &\quad \text{else } (\max2(\max2(x, x * \text{mipm}(xs)), \text{msp}(xs)), \max2(x, x * \text{mipm}(xs)) \\ &\quad \quad \quad , \min2(x, x * \text{mip}(xs))) \\ &= \{ \text{gather } \text{msp}, \text{mip} \text{ and } \text{mipm} \text{ calls using let } \} \\ &\quad \text{let } (u, v, w) = (\text{msp}(xs), \text{mip}(xs), \text{mipm}(xs)) \text{ in} \\ &\quad \text{if } x \geq 0 \text{ then } (\max2(\max2(x, x * v), u), \max2(x, x * v), \min2(x, x * w)) \\ &\quad \text{else } (\max2(\max2(x, x * w), u), \max2(x, x * w), \min2(x, x * v)) \\ &= \{ \text{fold with } \text{msptup} \} \\ &\quad \text{let } (u, v, w) = \text{msptup}(xs) \text{ in} \\ &\quad \text{if } x \geq 0 \text{ then } (\max2(\max2(x, x * v), u), \max2(x, x * v), \min2(x, x * w)) \\ &\quad \text{else } (\max2(\max2(x, x * w), u), \max2(x, x * w), \min2(x, x * v)) \\ &= \{ \text{abstract \& share common sub-expressions } \} \\ &\quad \text{let } \{ (u, v, w) = \text{msptup}(xs); r = x * v; s = x * w; b = \max2(x, r); \\ &\quad \quad d = \max2(x, s) \} \text{ in if } x \geq 0 \text{ then } (\max2(b, u), b, \min2(x, s)) \\ &\quad \quad \text{else } (\max2(d, u), d, \min2(x, r)) \end{aligned}$$

The final optimised program is:

$$\begin{aligned} \text{msp}(xs) &= \text{let } (u, -, -) = \text{msptup}(xs) \text{ in } u \\ \text{msptup}([x]) &= (x, x, x) \\ \text{msptup}(x:xs) &= \text{let } \{ (u, v, w) = \text{msptup}(xs); r = x * v; s = x * w; b = \max2(x, r); \\ &\quad d = \max2(x, s) \} \text{ in if } x \geq 0 \text{ then } (\max2(b, u), b, \min2(x, s)) \\ &\quad \text{else } (\max2(d, u), d, \min2(x, r)) \end{aligned}$$

The derived algorithm for *msptup* is more complex than that for *msstup*, even though their initial specifications are similar. However, we used essentially the same transformation techniques, namely fusion followed by tupling. We reiterate that fusion helps to eliminate intermediate data structures (improving on space), while tupling helps to eliminate redundant calls (improving on time).

Compared to *mss* derivation, only two extra laws, that allow distribution of *max* (and *min*) over products, are needed to systematically derive a more intricate, but yet efficient algorithm for the MSP problem. An alternative derivation proposed by Bird, requires a somewhat deeper insight based on Horner's rule. This approach is considerably more complex since the corresponding Horner's rule have to be invented over tupled functions (for the MSP problem). Let us review the Horner's rule approach.

5 Classical Derivation via Horner's Rule

The MSS problem originated from Bentley [Ben86]. Formal derivation to obtain efficient linear-time algorithm was developed by Bird [Bir88], amongst others.

The traditional derivation for the MSS problem has been based on function-level reasoning via the Bird-Meerstens Formalism (BMF). A major theme of the BMF approach is to capture common patterns of computations via higher-order functions, and to make heavy use of laws/theorems concerning these operations. Often, algebraic properties on the components of higher-order operations are required as side-conditions.

An important example is the Horner's rule to reduce the number of operations used for polynomial-like evaluation. A special case of this rule/law (instantiated to three terms) can be stated as:

$$(a_1 \otimes (a_2 \otimes a_3)) \oplus ((a_2 \otimes a_3) \oplus a_3) = ((a_1 \oplus 1_\otimes) \otimes a_2 \oplus 1_\otimes) \otimes a_3$$

The algebraic side-conditions required are that both \oplus and \otimes be associative, 1_\otimes be the left identity of \otimes , and \otimes distributes through \oplus . To generalise to n terms, we could express this rule as:

$$(\oplus /) \text{map}(\otimes /, \text{tails}([a_1, \dots, a_n])) = \otimes \not\vdash_{1_\otimes} [a_1, \dots, a_n] \quad (11)$$

where the operators \otimes , $/$, $\not\vdash$ and tails are defined by:

$$\begin{aligned} a \otimes b &= (a \otimes b) \oplus 1_\otimes \\ \odot / [x] &= x \\ \odot / (xs ++ ys) &= (\odot / xs) \odot (\odot / ys) \\ \odot \not\vdash_e \text{Nil} &= e \\ \odot \not\vdash_e (xs ++ [y]) &= (\odot \not\vdash_e xs) \odot y \\ \text{tails}(\text{Nil}) &= \text{Nil} \\ \text{tails}(x:xs) &= (x:xs):\text{tails}(xs) \end{aligned}$$

Horner's rule is a key insight used in the calculational derivation of *mss* in [Bir88]. We re-produce this classical derivation below.

$$\begin{aligned} \text{mss}(xs) &= (\text{max2 } /)(\text{map}((+ /), \text{segs}'(xs))) \\ &= \{ \text{unfold } \text{segs}'(xs) = \text{flatten}(\text{map}(\text{tails}, \text{inits}(xs))) \} \\ &\quad (\text{max2 } /)(\text{map}((+ /), \text{flatten}(\text{map}(\text{tails}, \text{inits}(xs)))))) \\ &= \{ \text{apply law : } \text{map}(f, \text{flatten}(xss)) = \text{flatten}(\text{map}(\backslash xs.\text{map}(f, xs), xss)) \} \\ &\quad (\text{max2 } /)(\text{flatten}(\text{map}(\backslash z.\text{map}((+ /), z), \text{map}(\text{tails}, \text{inits}(xs))))) \\ &= \{ \text{apply law : } \text{max}(\text{flatten}(xss)) = \text{max}(\text{map}(\text{max}, xss)) \} \\ &\quad (\text{max2 } /)(\text{map}((\text{max2 } /), \text{map}(\backslash z.\text{map}((+ /), z), \text{map}(\text{tails}, \text{inits}(xs))))) \\ &= \{ \text{apply law : } \text{map}(f, \text{map}(g, xs)) = \text{map}(f \circ g, xs) \text{ twice} \} \\ &\quad (\text{max2 } /)(\text{map}(\backslash y. (\text{max2 } /)(\text{map}(\backslash z.\text{map}((+ /), z), \text{tails}(y))), \text{inits}(xs))) \\ &= \{ \text{apply Horner's rule : } (\oplus /) \text{map}(\otimes /, \text{tails } xs) = \otimes \not\vdash_{1_\otimes} xs \} \\ &\quad (\text{max2 } /)(\text{map}(\otimes \not\vdash_0, \text{inits}(xs))) \text{ where } a \otimes b = \text{max2}(a+b, 0) \\ &= \{ \text{apply scan law : } \text{map}(\odot \not\vdash_0, \text{inits}(xs)) = \odot \not\vdash_0 xs \} \\ &\quad (\text{max2 } /)(\otimes \not\vdash_0 xs) \end{aligned}$$

Note that we used a non-recursive definition of *segs'* which returns segments in a different order (from *segs* given in Sec. 1). We used:

$$\begin{aligned}
\odot \#_e \text{ Nil} &= [e] \\
\odot \#_e (xs ++ [y]) &= (\odot \#_e xs) ++ [\text{last}(\odot \#_e xs) \odot y] \\
\text{last}(xs ++ [y]) &= y \\
\text{flatten}(\text{Nil}) &= \text{Nil} \\
\text{flatten}(xs:xss) &= xs ++ \text{flatten}(xss)
\end{aligned}$$

The final algorithm obtained for *mss* has a linear time complexity, and also a linear (variable) space complexity due to an intermediate list from $(\odot \#_0 xs)$. This slightly worse space behaviour may be improved by fusion transformation. Although this classical derivation looks more concise than our proposed derivation, it requires more insightful steps with non-trivial side conditions.

For example, the Horner's rule for *MSS* problem requires that $+$ distributes through *max2*, and that the identity of $+$, namely 0 , be present. (The use of 0 as the identity of $+$ actually results in a less defined *mss* algorithm since it becomes ill-defined for lists with only negative numbers. But this shortcoming is often tolerated.) Worse still is the possibility that distributive property required may not be immediately detected, but support for such property may come from generalised/tupled functions instead. Consider the *MSP* problem. The $*$ operator does not distribute over *max2* for negative numbers, but we do have:

$$\begin{aligned}
\text{max2}(a,b)*c &= \text{if } x \geq 0 \text{ then } \text{max2}(a*c,b*c) \text{ else } \text{min2}(a*c,b*c) \\
\text{min2}(a,b)*c &= \text{if } x \geq 0 \text{ then } \text{min2}(a*c,b*c) \text{ else } \text{max2}(a*c,b*c)
\end{aligned}$$

As Bird reported : “*These facts are enough to ensure that, with suitable cunning, Horner's rule can be made to work*”[Bir89]. Instead of *max2* and $*$ as instantiations of \oplus and \otimes operators for its Horner's rule, a more insightful tupled functions was used instead.

$$\begin{aligned}
(a_1, b_1) \oplus (a_2, b_2) &= (\text{min2}(a_1, a_2), \text{max2}(b_1, b_2)) \\
(a, b) \otimes c &= \text{if } c \geq 0 \text{ then } (a*c, b*c) \text{ else } (b*c, a*c)
\end{aligned}$$

With the above, we can now prove that \otimes distributes through \oplus :

$$((a_1, b_1) \oplus (a_2, b_2)) \otimes c = ((a_1, b_1) \otimes c) \oplus ((a_2, b_2) \otimes c)$$

Inventive insights are needed to come up with such tupled functions for *MSP*-like problems. In addition, the original definition of *mss* has to be rewritten to use such tupled functions before its calculational derivation can be applied. The main difficulty stems from the highly abstract nature of Horner's rule and its algebraic side-conditions. Fortunately, our proposal avoids this problem by decomposing the derivation into fusion (which requires the distributive conditions), followed by tupling (to eliminate redundant calls). Such separation allows difficult theorems/insights to be dispensed by simpler transformation techniques.

6 Avoiding Accumulation to Save Tupling

The perceptive reader may noticed that our specification of *mss* differs slightly from [Bir89]. Specifically, the classical definition of *mss* generates segments via:

$$\text{segs}'(xs) = \text{flatten}(\text{map}(\text{tails}, \text{inits}(xs)))$$

In contrast, we started with the following definition before it was fused to the recursive definition given in Sec 1.

$$\text{segs}(xs) = \text{flatten}(\text{map}(\text{inits}, \text{tails}(xs)))$$

Both *segs'* and *segs* yield the same set of segments, except for their order. Unfortunately, this innocuous change seems to have an effect on the kind of derivations which can be performed.

For example, if *segs* were used by the classical derivation, we will need a different type of Horner' rules, that are oriented for right-to-left reductions. Correspondingly, if *segs'* were used by our modular approach to derivation, we require equations based on right-to-left evaluation. These equations are typically referred to as *snoc*-based equations (which deconstruct a given list backwards), instead of the usual *cons*-based equations.

At this point, two questions may puzzle the reader : How do we obtain such *snoc*-based equations? When should we use them?

The *snoc*-based equations can be obtained as a by-product of parallelization. Given a *cons*-based equation, the inductive parallelization method presented in [HTC98] is capable of (automatically) deriving a *++*-based parallel equation. This can subsequently be instantiated to the *snoc*-based equation. As an example, consider the *cons*-based version of *inits* function given in Sec 1. Using the method of [HTC98], it is possible to derive the following *++*-based parallel equation.

$$\text{inits}(xs ++ ys) = \text{inits}(xs) ++ \text{map}((xs ++), \text{inits}(ys))$$

By instantiating *ys* to *[y]*, we can obtain the following *snoc*-based equation.

$$\text{inits}(xs ++ [y]) = \text{inits}(xs) ++ [xs ++ [y]]$$

The second question is *when should we use such snoc-based equations?* We should consider them when our fusion technique is about to fail through the application of an *accumulation* tactic – which is known be unhelpful to tupling! For example, consider the fusion of *segs'* below.

$$\begin{aligned} \text{segs}'(x:xs) &= \{ \text{instantiate } xs = x:xs \} \\ &\quad \text{flatten}(\text{map}(\text{tails}, \text{inits}(x:xs))) \\ &= \{ \text{unfold } \text{inits}, \text{map}, \text{flatten} \} \\ &\quad \text{tails}[x] ++ \text{flatten}(\text{map}(\text{tails}, \text{map}((x:), \text{inits}(xs)))) \\ &= \{ \text{apply law (3) : } \text{map}(f, \text{map}(g, xs)) = \text{map}(f \circ g, xs) \} \\ &\quad \text{tails}[x] ++ \text{flatten}(\text{map}(\text{tails} \circ (x:), \text{inits}(xs))) \end{aligned}$$

After several steps, we are still unable to fold as we encountered a slightly enlarged expression of the form $\text{flatten}(\text{map}(\text{tails} \circ (x:), \text{inits}(xs)))$. As reported elsewhere [Bir84] and [HIT99], this calls for the use of an *accumulation tactic* which generalizes $(x:)$ to $(w ++)$:

$$\text{asegs}'(w, xs) = \text{flatten}(\text{map}(\text{tails} \circ (w ++), \text{inits}(xs)))$$

A subsequent fusion transformation obtains:

$$\begin{aligned} \text{asegs}'(w, [x]) &= \text{tails}(w ++ [x]) \\ \text{asegs}'(w, x:xs) &= \text{tails}(w ++ [x]) ++ \text{asegs}'(w ++ [x], xs) \end{aligned}$$

In general, this accumulation tactic is bad for two reasons. Firstly, the presence of an accumulating (list) parameter indicates that fusion is not totally successful (having failed for the accumulating parameter). Secondly, the resulting function (with an accumulating parameter) is **bad** for tupling since its redundant calls may have infinitely many variants of the accumulative arguments during transformation. This reduces the chances of successful folding. As a result, we are unable to apply tupling to *asegs'* (or its *mss* counterpart) to eliminate the redundant *tails* calls (or its *mis*-like counterparts). A key lesson is – *avoid (or delay) the application of accumulating tactic, where possible*. One way to avoid accumulation is to turn to *snoc*-based equations, whenever the use of accumulation is inevitable. In the case of *segs'*, the corresponding fusion transformation using *snoc*-based equations can proceed (without accumulation), as shown:

$$\begin{aligned}
 \text{segs}'(xs++[y]) &= \{ \text{instantiate } xs=xs++[y] \} \\
 &\quad \text{flatten}(\text{map}(\text{tails}, \text{inits}(xs++[y]))) \\
 &= \{ \text{unfold } \text{inits}, \text{apply law (1)} \} \\
 &\quad \text{flatten}(\text{map}(\text{tails}, \text{inits}(xs))++\text{map}(\text{tails}, [xs++[y]])) \\
 &= \{ \text{apply law : } \text{flatten}(xr++xs) = \text{flatten}(xr)++\text{flatten}(xs) \} \\
 &\quad \text{flatten}(\text{map}(\text{tails}, \text{inits}(xs)))++\text{flatten}(\text{map}(\text{tails}, [xs++[y]])) \\
 &= \{ \text{fold with } \text{segs}', \text{unfold map, flatten} \} \\
 &\quad \text{segs}'(xs)++ [\text{tails}(xs++[y])]
 \end{aligned}$$

With this version of *segs'*, the main *mss* function can be transformed to:

$$\begin{aligned}
 \text{mss}([x]) &= x \\
 \text{mss}(xs++[y]) &= \text{max2}(\text{mss}(xs), \text{max2}(\text{mis}(xs)+y, y)) \\
 \text{mis}([x]) &= x \\
 \text{mis}(xs++[y]) &= \text{max2}(\text{mis}(xs)+y, y)
 \end{aligned}$$

The redundant calls in the above fused program can now be eliminated via tupling without being hindered by accumulating parameters.

7 Discussion and Concluding Remarks

Fusion transformation is considered to be one of the most important derivation technique in the constructive algorithmics [Bir89, Fok92], with many useful fusion theorems being developed for deriving various classes of efficient programs (A good summary of these theorems can be found in [Jeu93]). In contrast, the importance of tupling transformation technique [Fok89] for program derivation was hardly addressed, let alone a good combination of fusion and tupling.

In this paper, we have proposed a new strategy for algorithm derivation through two key transformation techniques. Our strategy provides a more modular derivation with two key phases, for the eliminations of intermediate data and redundant calls, respectively. While the steps taken are smaller than the traditional BMF approach, the opportunities for mechanisation are high since we rely on less insightful laws/theorems. In particular, only simple distributive laws are needed in the enhanced fusion process, while tupling depends on only equational definitions for its transformation. This combination of fusion (with

laws) and tupling is particularly powerful. Finding a good collection of modular transformation techniques could provide an improved methodology for developing programming pearls.

References

- [BD77] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of ACM*, 24(1):44–67, January 1977.
- [Ben86] Jon Bentley. *Programming Pearls*. Addison-Wesley, 1986.
- [Bir84] Richard S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Trans. on Programming Languages and Systems*, 6(4):487–504, October 1984.
- [Bir88] Richard S. Bird. *Lectures on Constructive Functional Programming*. Springer-Verlag, 1988.
- [Bir89] Richard S. Bird. Lecture notes on theory of lists. In *STOP Summer School on Constructive Algorithmics, Abeland*, pages 1–25, 9 1989.
- [Chi92] Wei-Ngan Chin. Safe fusion of functional expressions. In *7th ACM LISP and Functional Programming Conference*, pages 11–20, San Francisco, California, June 1992. ACM Press.
- [Chi93] Wei-Ngan Chin. Towards an automated tupling strategy. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 119–132, Copenhagen, Denmark, June 1993. ACM Press.
- [Chi94] Wei-Ngan Chin. Safe fusion of functional expressions II: Further improvements. *Journal of Functional Programming*, 4(4):515–555, October 1994.
- [CK01] Manuel MT Chakravarty and Gabriele Keller. Functional array fusion. In *ACM Intl. Conference on Functional Programming*, pages 205–216, Florence, Italy, September 2001. ACM Press.
- [CT97] W.N. Chin and A. Takano. Deriving laws by program specialization. Technical report, Hitachi Advanced Research Laboratory, July 1997.
- [Fok89] M. Fokkinga. Tupling and mutumorphisms. *Squiggolist*, 1(4), 1989.
- [Fok92] M. Fokkinga. *Law and Order in Algorithmics*. Ph.D thesis, Dept. INF, University of Twente, The Netherlands, 1992.
- [GLPJ93] A. Gill, J. Launchbury, and S. Peyton-Jones. A short-cut to deforestation. In *6th ACM Conference on Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark, June 1993. ACM Press.
- [HIT96] Z. Hu, H. Iwasaki, and M. Takeichi. Deriving structural hylomorphisms from recursive definitions. In *ACM SIGPLAN International Conference on Functional Programming*, pages 73–82, Philadelphia, Pennsylvania, May 1996. ACM Press.
- [HIT99] Z. Hu, H. Iwasaki, and M. Takeichi. Calculating accumulations. *New Generation Computing*, 17(2):153–173, 1999.
- [HITT97] Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple traversals. In *2nd ACM SIGPLAN International Conference on Functional Programming*, pages 164–175, Amsterdam, Netherlands, June 1997. ACM Press.
- [HTC98] Z. Hu, M. Takeichi, and W.N. Chin. Parallelization in calculational forms. In *25th Annual ACM Symposium on Principles of Programming Languages*, pages 316–328, San Diego, California, January 1998. ACM Press.

- [Jeu93] J. Jeuring. *Theories for Algorithm Calculation*. Ph.D thesis, Faculty of Science, Utrecht University, 1993.
- [JH99] S. Peyton Jones and J. Hughes, editors. *Haskell 98: A Non-strict, Purely Functional Language*. Available online: <http://www.haskell.org>, February 1999.
- [SF93] T. Sheard and L. Fegaras. A fold for all seasons. In *6th ACM Conference on Functional programming Languages and Computer Architecture*, Copenhagen, Denmark, June 1993. ACM Press.
- [SGN94] M.H. Sørensen, R. Glück, and Jones N.D. Towards unifying deforestation, supercompilation, partial evaluation and generalised partial computation. In *European Symposium on Programming (LNCS 788)*, Edinburgh, April 1994.
- [Smi89] Douglas R. Smith. KIDS - a semi-automatic program development system. Technical report, Kestrel Institute, October 1989.
- [SS97] H. Seidl and M.H. Sørensen. Constraints to stop higher-order deforestation. In *24th ACM Symposium on Principles of Programming Languages*, Paris, France, January 1997. ACM Press.
- [TH00] M. Takeichi and Z. Hu. Calculation carrying programs: How to code program transformations (invited paper). In *International Symposium on Principles of Software Evolution (ISPSE 2000)*, Kanazawa, Japan, November 2000. IEEE Press.
- [TM95] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *ACM Conference on Functional Programming and Computer Architecture*, pages 306–313, San Diego, California, June 1995. ACM Press.
- [Wad88] Phil Wadler. Deforestation: Transforming programs to eliminate trees. In *European Symposium on Programming*, Nancy, France, (LNCS, vol 300, pp. 344–358), March 1988.
- [YHT02] T. Yokoyama, Z. Hu, and M. Takeichi. Yicho: A system for programming program calculations. Technical Report METR 2002–07, Department of Mathematical Engineering, University of Tokyo, June 2002. submitted for publication.

Appendix: Implementing Modular Derivation Using Yicho

The modular derivation approach via fusion and tupling can be easily implemented using the Yicho system [YHT02], a system supporting the coding of calculation carrying programs [TH00] that can relax the tension between efficiency and clarity in programming. The main idea is to accompany clear programs with appropriate calculations describing intention of how to manipulate programs to be efficient. Calculation specification can be executed automatically by the Yicho system to derive efficient programs.

To illustrate, we give the complete code for solving the maximum segment sum problem. The initial program is first coded in Haskell [JH99], using curried syntax. The laws used for fusion (Section 3.1) are coded as follows. The names of meta variables are prefixed with %, and # $e_1 \rightarrow e_2$ denotes meta lambda abstraction (over object expression).

```

%mss_law = %map_append <+
           %max_append <+
           %map_distribution <+
           %max_add
%map_append = # (map %f (%xs ++ %ys)) -> map %f %xs
                                     ++ map %f %ys;
%max_append = # (max (%xs ++ %ys)) -> max2 (max %xs) (max %ys);
%map_distribution = # (map %f (map %g %xs)) ->
                                     map (%f . %g) %xs;
%max_add = # (max (map ((+) %x) %xs)) -> %x + max %xs;

```

The modular derivation of tupling transformation after fusing function *f* with another auxiliary function *aux* using *law* can be described by

```

%fusing_tupling2 = # %law %f %aux ->
  letm
    -- fusion --
    {e1 = %rec %law (%f []);
      \a x -> %h1 a (%f x,%aux x) =
        \a x -> %rec %law (%f (a:x));
      e2 = %rec %law (%aux []);
      \a x -> %h2 a (%f x,%aux x) =
        \a x -> %rec %law (%aux (a:x))}
  in
    -- tupling --
    let %h = \a (x,y) -> (%h1 a (x,y), %h2 a (x,y))
    in fst (foldr %h (%e1,%e2));

```

We will not explain this calculation program in detail, where higher-order pattern matching plays an important role [YHT02,TH00] in the implementation of fusion transformation. The code fragment:

```

\ a x -> %h1 a (%f x,%g x) = \ a x -> %rec %law (%f (a:x));

```

is intended to apply law *%law* recursively to transform the expression *%f (a:x)*, and then match the result with a higher-order pattern *%h1 a (%f x, %aux x)* to get a definition for *%h1*. With these definitions, we can execute the expression

```

%fusing_tupling2 %mss_law mss mis
  where mis = max . map sum . inits;

```

on Yicho to obtain the efficient program for *mss* (as in Figure 1) in a fully automatic way. One can follow the above to implement the derivation for *m_{sp}*. The calculation code is almost the same, except for the laws and auxiliary functions.

Generative Programming for Embedded Software: An Industrial Experience Report

Krzysztof Czarnecki,¹ Thomas Bednasch,² Peter Unger,³ and Ulrich Eisenecker²

¹DaimlerChrysler AG, Research and Technology, Ulm, Germany

²University of Applied Sciences Kaiserslautern, Zweibrücken, Germany

³Technical University of Ilmenau, Germany

czarnecki@acm.org, Thomas.Bednasch@epost.de, Punger@gmx.de,
Ulrich.Eisenecker@t-online.de

Abstract. Physical products come in many variants, and so does the software embedded in them. The software embedded in a product variant usually has to be optimized to fit its limited memory and computing power. Generative programming is well suited for developing embedded software since it allows us to automatically produce variants of embedded software optimized for specific products. This paper reports on our experience in applying generative programming in the embedded domain. We propose an extended feature modeling notation, discuss tool support for feature modeling, describe a domain-independent system configuration editor, and comment on the applicability of static configuration in the area of embedded systems.

1 Introduction

An embedded system is any computer that is a component in a larger system and that relies on its own microprocessor [Wol02]. The products that contain embedded software usually come in many variants and thus the requirements on the embedded software also vary. Sources of variation include different product features and different supported hardware, e.g., on-board satellite software may support different services and different fuel injection systems impact automotive engine control. At the same time, in order to reduce recurring hardware costs, embedded software usually runs on platforms with reduced amount of memory and processor capability and with minimum support hardware. As a result, embedded software usually needs to be optimized for minimum memory footprint and for maximum speed to perform its required computations and actions within its real-time deadlines.

Generative programming is well suited for developing embedded software since it allows us to automatically produce variants of embedded software optimized for specific products based on a set of reusable components. However, a practical approach has to take into account the special requirements of industrial embedded software development, such as requirements on the programming languages being used or the need to support validation and verification procedures.

This paper reports on our experience in applying generative programming in the embedded domain. It also proposes a technology projection that is geared towards satisfying the current requirements of industrial embedded software development. The case studies used to illustrate our points are described in Section 3. The specific contributions of this paper include (1) extended requirements on feature modeling (Section 4), (2) comparison of different approaches to tool support for feature modeling (Section 5), (3) concept of a domain-independent system configuration

editor based on feature modeling (Section 6), (4) combination of the configuration editor with a template-based generation approach (Sections 7-9), and discussion of the applicability of this approach in the industrial context (Sections 10). Section 11 discusses related work. We conclude with ideas for future work in Section 12.

2 What Is Generative Programming?

Generative programming builds on system-family engineering (also referred to as product-line engineering) [CN01, WL99, Par76] and puts its focus on maximizing the automation of application development [Cle01, CE00, BO92, Cle88, Nei80]: given a system specification, a concrete system is generated based on a set of reusable components.

The means of application specification, the generators, and the reusable components are developed in a domain-engineering cycle and then used in application engineering to create concrete applications. The main domain engineering steps are

- domain analysis, which involves domain scoping, finding common and variable features and their dependencies, and modeling the structural and behavioral aspects of the domain concepts
- domain design, which involves the development of a common architecture for the system family and a production plan
- domain implementation, which involves implementing reusable components, domain-specific languages, and configuration generators.

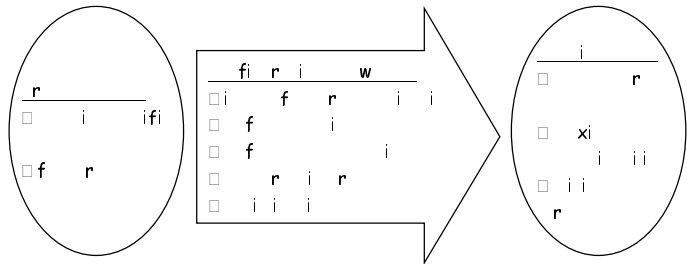


Fig. 1. Elements of a generative domain model [CE00]

The key to automating the assembly of systems is a generative domain model that consists of a problem space, a solution space, and the configuration knowledge mapping between them (see Fig. 1). The solution space comprises the implementation components and the common system family architecture, defining all possible combinations of the implementation components. The implementation components are designed for maximum combinability, minimum redundancy, and maximized reuse. The problem space, on the other hand, consists of the application-oriented concepts and features that application engineers use to express their needs. This space is implemented as a domain-specific language (DSL). The configuration knowledge specifies illegal feature combinations, default settings, default dependencies (some defaults may be computed based on some other features), construction rules (combinations of certain features translate into certain combinations of implementation components), and optimizations. The configuration knowledge is

implemented in the form of generators. The generated products may also contain non-software artifacts, such as test plans, manuals, tutorials, maintenance guidelines, etc.

Each of the elements of a generative domain model can be implemented using different technologies, which gives rise to different *technology projections*:

- Components can be implemented using, e.g., generic components (such as in the STL), component models (e.g., JavaBeans, ActiveX, or CORBA), or aspect-oriented programming approaches.
- Generators can be implemented using preprocessors (e.g., template processors), application generators, built-in metaprogramming capabilities of a language (e.g., template metaprogramming in C++), or extendible programming systems.
- DSLs can be implemented using new textual or graphical languages (or language extensions), programming language-specific features, or interactive wizards.

The choice of a specific technology depends on its technical suitability for a given problem domain, mandated programming languages, existing infrastructure, familiarity of the developers with the technology, political and other considerations. We present one specific technology projection geared towards embedded software.

3 Case Studies

This paper is based on our experience in working with internal DaimlerChrysler business units from the space, aerospace, and automotive domains. The main example for this paper comes from the satellite domain. The European Space Agency has issued the Packet Utilization Services (PUS) standard [PUS], which defines a service-based interface for communicating with a satellite and a number of generic services, such as housekeeping (periodically sending telemetry reports about the status of various satellite functions to the ground) or storage control (storing telemetry data when there is no connection to the ground). The satellite case study described in this paper is based on a project at DaimlerChrysler Research and Technology to develop a prototype of a product-line architecture implementing the PUS standard for Astrium Space. Astrium Space is a leading European satellite manufacturer and was part of the DaimlerChrysler AG until 2001. The purpose of this product-line architecture is to avoid the re-development of the higher-level, application-oriented communication software for each new satellite from scratch.

Another example which is discussed in this paper is the OSEK/VDX standard [OSEK] defining a realtime operating system for embedded automotive applications. OSEK/VDX is defined as a statically configured, application-specific operating system. The number and names of tasks, stack sizes, events, alarms, etc. used by an application are statically defined using the configuration language OIL [OIL]. An OSEK/VDX implementation provides a generator that takes an OIL specification and generates the application-specific tasking and communication support (usually as C code). We have conducted an experiment to express the configuration space defined by OIL using feature models.

4 Extended Feature Modeling

During domain analysis, the scope of a product line is captured as a feature model [KCH+90,CE00]. A feature model describes the common and variable features of the

products, their dependencies, and supplementary information such as feature descriptions, binding times, priorities, etc. Features are organized into feature diagrams, which reveal the kinds of variability contained in the product configuration space. An example of a feature diagram is shown in Fig. 2. It describes a simple model of a car. The root of the diagram represents the concept *car*. The remaining nodes are features:

- *Mandatory features*: Every car has a body, transmission, and engine.
- *Optional feature*: A car may pull a trailer or not.
- *Alternative features*: A car may have either an automatic or a manual transmission.
- *Or-features*: A car may have an electric engine, a gasoline engine, or both.

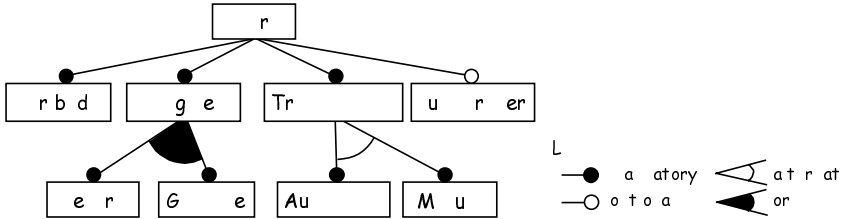


Fig. 2. A sample feature diagram of a car

A different representation of a product configuration space is to use parameter tables (e.g., as in the FAST method [WL99] or [Cle01]). In our experience, feature diagrams allow better modularity in large configuration spaces than flat parameter tables. Thanks to the hierarchical organization of feature diagrams, parameter sets that are pertinent to just one class of variants can be hidden in one branch of a feature diagram and need not be considered when configuring a variant outside this class. In this context, it is important to note that a feature diagram is not just a part-of hierarchy of the software modules, but a hierarchical decomposition of the configuration space. Features in a feature diagram need not correspond to concrete software modules, but may represent abstract properties such as performance requirements and aspectual properties affecting several software modules such as synchronization.

The original feature diagram notation from [KCH+90] is insufficient to effectively model the variabilities in our case studies. The following subsections list the shortcomings and propose extensions to overcome them. The proposed extensions are geared towards representing feature models using a MetaCASE environment as described in Section 5. We are currently working on an alternative solution to the presented one which is based on annotating feature models with partial or full specializations of other feature models at the meta-level and base level.

4.1 Cardinalities

[CE00, p. 117] suggests to model multiplicity using a subfeature (see Fig. 3) rather than extending the feature notation with cardinalities. The reason behind this decision was to keep the notation as simple as possible and to avoid cluttering it with all the mechanisms known from structural modeling such as UML class diagrams.

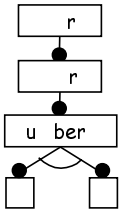


Fig. 3.
Representing
multiplicity in
a feature dia-
gram

Unfortunately, this approach does not necessarily work when the feature that needs some multiplicity is the root of a subtree containing variable features. In that case, our intention could be not just to specify the number of doors as in Fig. 3, but to express that a specific configuration may include several different variants of the same feature. This is demonstrated in Fig. 4, which shows a fragment of the feature diagram from our satellite software case study. According to the PUS standard, the interface to a satellite consists of a number of applications, which implement services consisting of subservices. As the diagram shows, every satellite has a packet router application. It may have one storage control application or not. And it may have zero or more user defined applications, which is expressed by the cardinality “*” next to the feature `UserDefinedApp`. Each of the

user defined applications may implement a different selection of PUS-predefined services (the service and subservice numbers are defined in the PUS standard) and user-defined services. Finally, the PUS standard defines some of the subservices as mandatory and some as optional. In the diagram, the filled or empty circles could have been alternatively rendered as the cardinalities 1 or 0..1, respectively.

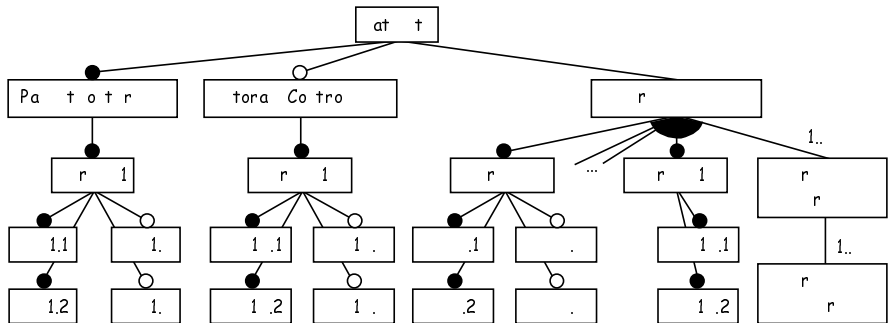


Fig. 4. Fragment of a feature diagram representing the configuration of a satellite interface according to the PUS standard

4.2 Attributes

The next extension was to introduce attributes. A feature may contain a number of attributes, just like a UML classifier. An exploded representation of the feature `StorageControlApp` is shown in Fig. 5.

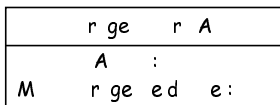


Fig. 5. Exploded representation of a feature showing its attributes

The reason for introducing attributes was to allow for a more concise representation of feature diagrams. Of course, each attribute could be represented as a subfeature, but this quickly leads to very large diagrams. With attributes, the tree structure is primarily used to organize a parameter space in a hierarchical way. If certain parameters of a given feature, then they can be represented as attributes include a type specification. Enumeration and enumeration type in different places and this way use alternative subfeatures multiple times. Finally,

attributes allow for a concise representation in the configuration tool (see Section 6). Currently, we work on a more uniform feature model representation, which avoids attributes as a separate concept and allows to achieve the goals described in this section using an extended form of features.

4.3 Reference Attributes

In addition to integer, float, boolean, string, enumeration and list attributes, we also need reference attributes. Cardinalities allow us to include several different configurations of feature subtrees in a system, and reference attributes allow us to model graph-like connections between the different configurations of feature subtrees. As an example, consider the partial feature diagram of the configurable real-time operating system OSEK/VDX in Fig. 6. An OSEK/VDX configuration for a particular application consists of a number of tasks, resources, events, alarms, and counters. The feature Task contains the reference attribute Resource, which specifies (using the attribute specification notation of UML) that a given task in a OSEK/VDX configuration can have zero or more specific resources. Despite these graph-like relationships, the feature-subfeature decomposition remains hierarchical, which is important for aiding the configuration process described in Section 6.

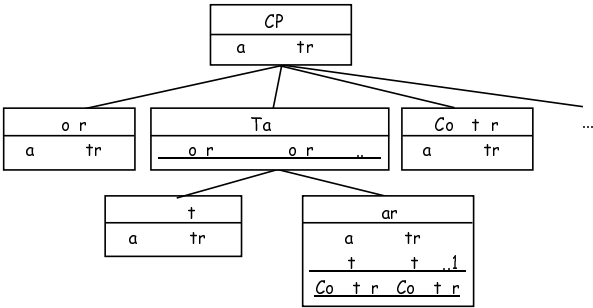


Fig. 6. Fragment of a feature diagram representing the configuration space of the OSEK/VDX real-time operating system (reference attributes are underlined)

4.4 Metamodeling for Supplementary Information

A feature model has a feature diagram and usually contains additional information such as feature descriptions, rationale for each feature, stakeholders and client programs interested in each feature, examples of systems with a given feature, configuration constraints between features, default dependency rules, binding times and modes, open/closed attributes, and feature priorities (see [CE00]). From our experience, not all of these are needed in every project, and different projects often have their own additional kinds of supplementary information. For example, a project may have its own binding site model with product-specific binding times and sites. This problem can be solved by allowing developers to edit the metamodel of the feature modeling notation in order to tailor it to the project-specific needs. This capability needs to be supported by the feature modeling tool.

5 Feature Modeling Tool Support

Feature modeling can be performed using UML tools, MetaCASE tools, and dedicated feature modeling tools. We discuss each of these choices as next.

UML tools. A feature model notation can be created using the standard UML extension mechanisms, e.g., using a stereotyped class to represent features and the containment relationship to represent the feature-subfeature relationship (see e.g., [GFA98, Cla01a, Cla01b]). This approach, although practicable, has the drawback that current major UML modeling tools do not sufficiently support profiles. In particular, it is not possible to add constraints to a profile and have them checked automatically by the tool while creating a model. As a result, such tools cannot check whether the feature model being created is actually a valid feature model or not.

MetaCASE tools. MetaCASE tools genuinely support metamodel editing and allow us to create new notations. Examples of MetaCASE tools are MetaEdit+ from MetaCase Consulting [MC] and the Generic Modeling Environment (GME2000) from the Vanderbilt University [LMB+01]. We have used GME2000 to define a feature modeling notation and to perform the feature modeling for the satellite case study. (GME2000 is freely available at <http://www.isis.vanderbilt.edu>).

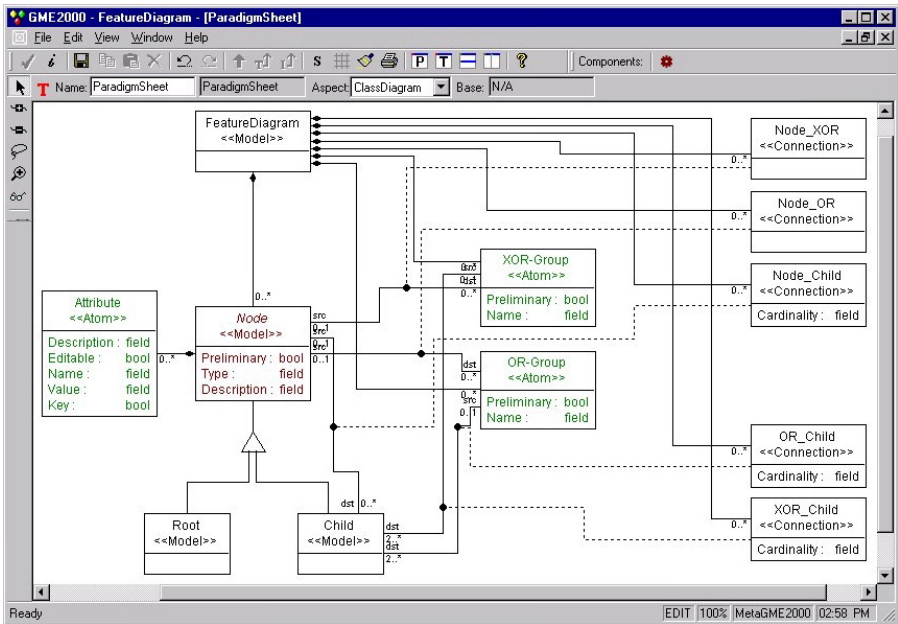


Fig. 7. UML metamodel of a feature modeling notation in GME2000

Fig. 7 shows the UML metamodel of our feature modeling notation in GME. (Some aspects of the notations, such as visualization, constraints, and attributes are not shown in this view.) According to our metamodel, a *FeatureDiagram* contains a number of *Nodes* (with the specializations *Root* and *Child*). A *Node* may have a number of *Attributes*, and *Nodes* and *Attributes* have the field *Description*. Furthermore, we have *XOR-Group* and *OR-Group* to represent

groups of alternative and or-features, respectively. Finally, connections with Children at the one end (i.e., Node_Child, OR_Child, and XOR_Child) have the field Cardinality with the default value 1.

The metamodel also contains a number of constraints in the Object Constraint Language (OCL) [UML, chapter 6], such as that a feature diagram may only contain one root, a child or a group has to have one incoming connection, a group has to have at least two children, and no cycles are allowed. These constraints are enforced automatically by GME when creating a concrete feature diagram.

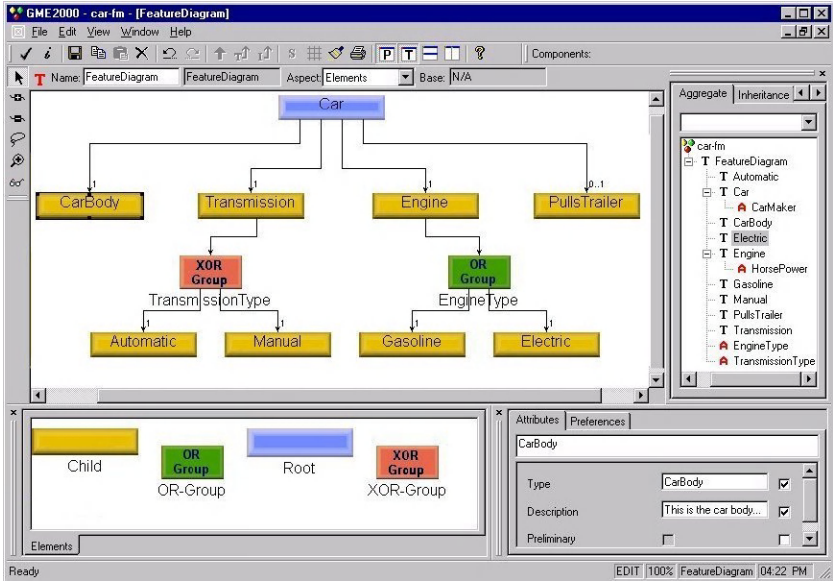


Fig. 8. Example of a feature model in the MetaCASE tool GME2000

Fig. 8 shows the sample feature diagram of a car from Fig. 2 using the notation defined in Fig. 7. The *Aggregate* tab on the right shows that *Car* has the attribute *CarMaker*, and *Engine* has the attribute *HorsePower* (there is a separate view in which attributes can be added to the features¹). Please note that based on the currently loaded metamodel, the GME environment offers only the desired modeling elements. The validity of the feature diagram can be checked by activating the OCL constraint checker through the *File* menu.

We found using a MetaCASE tool such as GME2000 particularly useful for prototyping new notations (such as experimenting with the extensions described in Section 4) because it allows us to quickly evolve a notation without reprogramming a dedicated tool.

¹ The *Attribute* tab in Fig. 8 shows the meta-attributes of the currently selected feature-model element. They are defined as fields of the metamodel elements, e.g., *Description* in Fig. 7, and are used to model the additional information described in Section 4.4. They should not be confused with the feature attributes described in Section 4.2, such as *CarMaker* or *HorsePower*. The latter attributes are instances of the *Attribute* atom in Fig. 7.

Dedicated feature modeling tools. A dedicated feature modeling tool is designed to provide optimal support for feature modeling. Compared to general-purpose modeling tools, it supports drawing and layouting feature trees in the concise notation from Fig. 2. An example of such a tool is Ami Eddi ([Sel00, Bli01, ESBC01]), which is freely available at www.generative-programming.org (see Fig. 9).

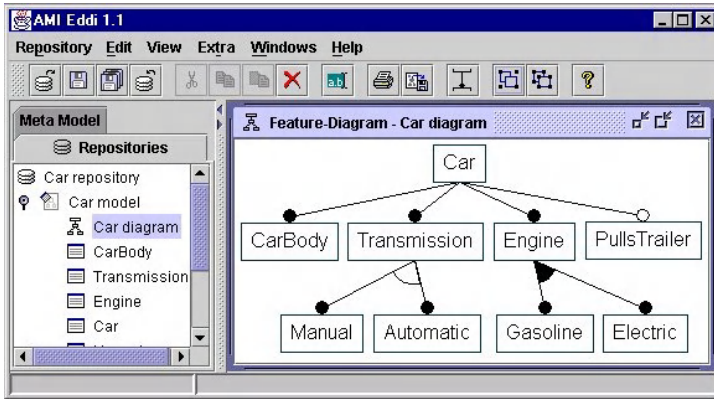


Fig. 9. Dedicated feature modeling tool Ami Eddi

An important feature of Ami Eddi is its metamodeling capability, which allows the user to extend the modeling notation with a user-defined list of additional information to be recorded with each feature (see Section 4.4). In GME2000, this capability corresponds to adding new fields to the metamodel elements (such as *Description* in Fig. 7). Future work on Ami Eddi includes the ability to annotate feature models with partial or full specializations of feature models at the metalevel and base level. This extension will provide the capability to appropriately model cardinalities and attributes without extending the notation with these concepts.

6 Configuration Editor

A feature model defines the configuration space of a product. We have developed ConfigEditor, a graphical configuration editor that helps an application engineer to create a specific product configuration based on a feature model.

In ConfigEditor, a configuration is constructed top-down by successively selecting the features of the currently loaded feature model according to its variabilities. The configuration process starts at the root, which is automatically inserted into a configuration. Whenever a new feature is added to a configuration, its mandatory subfeatures are added automatically. This is demonstrated in Fig. 10, which shows the initial configuration of a satellite (according to the feature model in Fig. 4). The feature tree being constructed is displayed in the left part of the ConfigEditor window. The tabbed panes on the right are related to the currently selected feature in the tree under construction. The *Attribute*-tab pane allows us to edit the attributes of the selected feature. A separate window shows the description of the currently selected feature (it displays the contents of the description field of the selected feature as provided in the feature model).

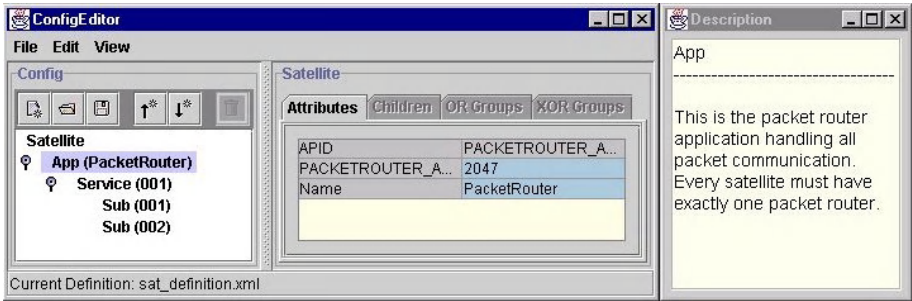


Fig. 10. The minimal satellite configuration in ConfigEditor. The *Attribute*-tab pane shows the attributes of the packet router application

The *Children*-tab pane allows us to add subfeatures not belonging to any group and having a cardinality other than 1. Fig. 11 shows this pane for our satellite example. According to Fig. 4, satellite may have an optional storage control application and zero or more user defined applications. These choices appear in the list on the *Children*-tab pane and the corresponding cardinalities are shown in brackets. The feature tree on the left shows the result of adding one storage control application and two user-defined applications. Please note that both newly added user-defined applications are marked with the red symbol “fill”. This marking gives the application engineer a visual clue that both features require further editing. In our example, the reason is that according to Fig. 4, a user defined application has a group of or-features and at least one of them needs to be added to the current configuration.

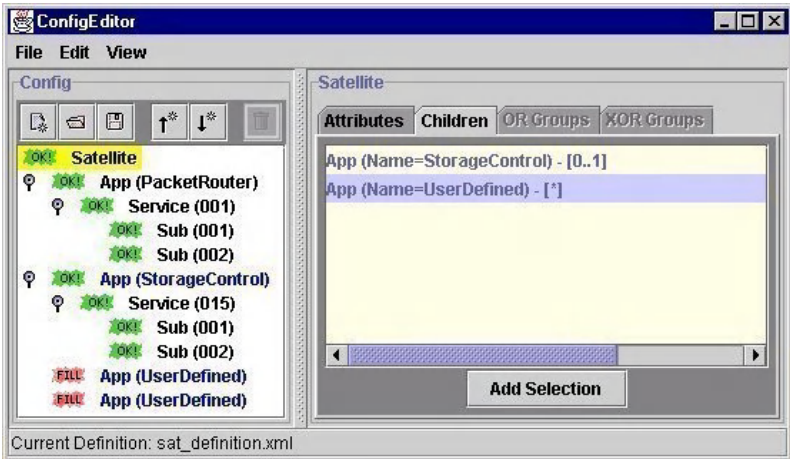


Fig. 11. Satellite configuration after adding the storage control and two user applications

Features belonging to a group of or-features or alternative features can be added using the *OR Group*- or the *XOR Group*-tab pane. Fig. 12 shows the *OR Group*-tab pane for our example. Because a feature may have more than one group of or-subfeatures, the pane provides a drop-down menu to switch between them. The list below the drop-down menu shows the list of or-features in the currently selected group. One or more different or-features from the list can be added to the current

configuration. The same or-feature can be added as many times as specified by its cardinality. The XOR Group-tab pane works similarly, with the difference that only one alternative feature can be selected from the list.

ConfigEditor accepts a feature model definition in a simple exchange format in XML. The idea is to provide a set of import filters that can transform XML exports of feature models from different modeling tools to our XML format. The current implementation of ConfigEditor includes an import filter for the XML export of a feature model from GME2000. A concrete configuration created using ConfigEditor is also saved in an XML format.

A configuration editor should also support checking validity of a configuration using additional constraints and automatically completing it by computing defaults. In our satellite example, the inclusion of some optional subservices (e.g., the telecommand subservice 15.5) requires the inclusion of other subservices (e.g., the corresponding telemetry subservice 15.6). Such dependencies are not represented in the feature diagram and need to be expressed as additional constraints. A constraint checker for ConfigEditor is currently being implemented.

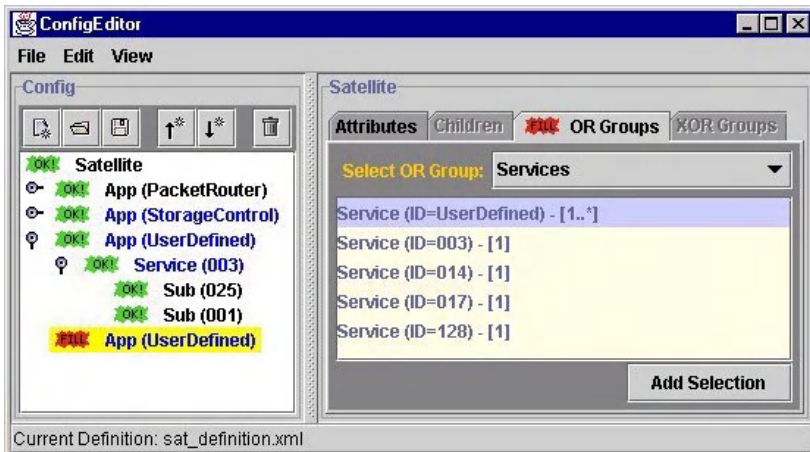


Fig. 12. Satellite configuration after adding service 3 to the first user-defined application and before adding services to the second user-defined application

7 Product-Line Architectures and Template-Based Generators

A product-line architecture (PLA) [Bos00] defines the components needed to build each of the products in the product line. In our satellite example, the architecture consists of a number of Ada83 packages representing domain abstractions such as up/down communication link, on-board storage manager, packet router, service decoder, components providing the different services, etc.

A PLA for embedded software is usually required to include only those components in a given product configuration that are actually used. This helps to minimize resource consumption such as memory and computing power and to

minimize recurring hardware costs. Additionally, this strategy reduces amount of code that needs to go through quality assurance procedures.²

In our satellite case study, the configuration of applications and services determines which service-providing components need to be included in a specific system. Other components requiring configuration are the service decoder and the storage control manager. For example, the service decoder needs to be able to decode communication packages only for the services included in the configuration.

We found template-based code generation [Cle01] particularly suited for the static configuration of embedded PLAs. In this approach, an arbitrary text file such as a source program file in any programming language or a documentation file are instrumented with code selection and iterative code expansion constructs. Such an instrumented file called is a *template*. A *template processor* takes a template plus a set of configuration parameters and generates a concrete instance of the template.

Compared to a programming-language-based generation approach, a simple text-based template approach has a number of properties that significantly ease its adoption on industrial project in the embedded software area. Most importantly, the approach is programming-language-independent. From our experience, organizations developing safety-critical embedded software are very reluctant to introduce major changes to their existing development procedures. In particular, they are significantly slower in adopting new programming languages than it is the case in other domains. Two major programming languages currently used in the embedded domain are C and Ada. For our satellite case study, we were mandated to use Ada 83 rather than Ada 95 in order to stay compatible with some radiation-tolerant processors for which no Ada 95 compiler exists. Another advantage of the template approach in the industrial context is the possibility to easily template existing source code. Finally, the template approach encourages generating readable source code. On our satellite project, this property, which is often considered to be counterproductive in the context of code generation, turned out to be actually critical for the adoption of the generation approach. By generating well-formatted, readable source code, the existing validation and verification procedures of a development organization can be applied to the generated product, just as to a manually created one. In our view, the high cost of validation and verification of the generator itself may be economical for general-purpose generators such as programming-language compilers, but it appears too high for in-house, product-line-specific generators.

8 Code Generation Using TL

For our satellite case study, we have used Template Language (TL) [Cle01] and the freely available TL processor (<http://craigc.com/>). TL provides code selection and iterative code expansion constructs and it can also be extended with user-defined functions. The configuration parameters for a TL template are stored in an XML file and the TL constructs can conveniently access them using XPath [XP] expressions. For example, the following code snippet uses the TL if-construct to decide whether to

² In fact, the quality assurance guidelines of several manufacturers form the space and aerospace domain forbid the inclusion of unused code in the safety-critical software embedded in their products. This is usually not the case in the automotive domain.

include the definition of the constant `MAX_STORAGE_OBJECTS` in the generated code or not:

```
#if "/Satellite/App/Service/Attribute[@Name='ID' and @Value='015']"##
  MAX_STORAGE_OBJECTS: constant integer
  := #"/Attribute[@Name='MAX_STORAGE_OBJECTS']/attribute::Value" ;
#fi#
```

The TL constructs are enclosed in `#...#` and shown in bold. The if-condition is an XPath expression which checks in the XML configuration whether any application of the satellite includes the service with the ID 015. The value of the constant is also retrieved from the XML configuration (the XPath expression looks for any attribute with the appropriate name). Assuming an XML configuration that includes service 15 and a `MAX_STORAGE_OBJECTS` attribute with the value 100, the TL processor would generate the following code:

```
MAX_STORAGE_OBJECTS: constant integer
:= 100 ;
```

A larger example of a template from the satellite case study is shown in Table 1. This is an excerpt from the template implementing the service decoder procedure, which calls the appropriate service package depending on the service and subservice number provided as a parameter. The template code is shown on the left and the corresponding generated code is shown on the right. The template uses nested pairs of for-constructs to iterate over the services defined in the XML configuration as subnodes in of `/Satellite/PUS/` and over the subservices of each service. One such nested pair is used to generate the list of service packages to be included using the `WITH` statement. Another pair is used to generate the nested `CASE` statement that calls the appropriate service procedure. Assuming that the XML configuration includes service 1 with subservices 1 and 2, and service 15 with subservices 1, 2, and 3, we get the code shown in the right column in Table 1 (the code generated by the for-constructs is shown in bold).

9 Putting It Together

The entire generative process supporting an embedded product line is shown in Fig. 13. It is interesting to note that the same configuration that is used to generate the embedded software is used to configure the testing and simulation environment. In the case of our satellite example, the ground station software that needs to communicate with the satellite is configured with the same configuration as the satellite software. This way, the ground station can only send packets that are understood by the satellite. In addition to generating the embedded software, we can use the same XML satellite configuration and the same template technology to generate the documentation for a concrete satellite configuration.

10 Applicability of the Approach

Static configuration is not always the best choice for embedded software product lines. In some cases, dynamic configuration can be more economical. The choice between these two approaches depends on the production volume of the products containing the embedded software, the resource limitations of the target platform, and whether the software embedded in a product needs to be maintained during its

lifetime or not. High production volumes usually imply the opportunity to significantly reduce recurring hardware costs by applying static configuration and cutting down on memory and processor power. In this case, the approach described in this paper may be well suited.

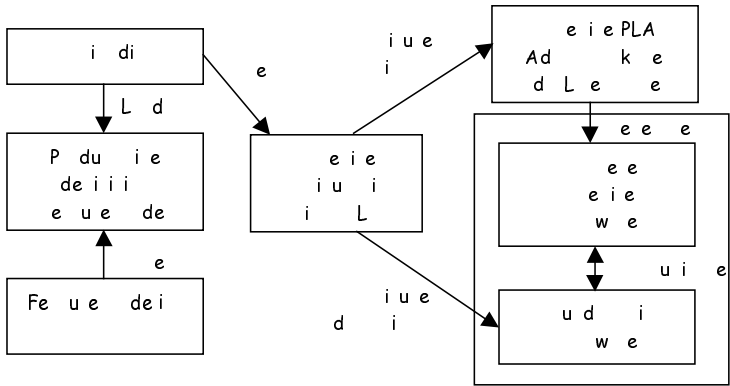


Fig. 13. Overview of the entire modeling and configuration chain

If the individual products need to be maintained over their lifetime, the cost of logistics for keeping track of static configurations may outweigh the savings on hardware. For example, this is the case for control functions in the automotive area, where the usual strategy is to have just one version of a software component for all different car models. Typically, an electronic control unit (ECU) comes with embedded functions (such as ABS, ESP, or engine control) capable of supporting different car models (i.e., the whole product-line code is put on one ECU). The idea is that the repair shops do not have to carry different variants of an ECU for different car models, but only one, which significantly reduces logistic costs.³ Consequently, automotive functions use dynamic configuration parameters for both tuning the software to the mechanics of a car (so called *end-of-line configuration*) and activating the necessary program code for a given car model. The car model identification is broadcast on the network in the car on start-up and the ECUs configure themselves automatically.

The situation is different for basic components that need to be installed on several ECUs in different configurations. This mandates the use of static configuration. An example of such a basic component is the operating system. Consequently, the automotive real-time operating system OSEK/VDX is set up as a generator emitting application-specific tasking and communication support for each ECU as a set of C functions. Thus, our approach is applicable to basic components that need to be included in different configurations on several different ECUs.

Finally, even in the case of relatively low production volumes, there still might be memory and computing power constraints mandating the use of static configuration. This is the case in the satellite domain, where the production volume is usually less

³ Currently, most repair shops do not have the equipment and expertise to flush the ECU memory, and software updates are usually done by replacing the entire ECU. This situation may change in future.

then 10 units per year, but the software and hardware has to be optimized for minimum power consumption. Also, in contrast to cars, satellites do not undergo any maintenance. As a result, our approach is well suited for this domain.

Table 1. An excerpt from the service decoder template and the corresponding generated code

Decoder procedure template	Generated code
<pre> WITH Ada.Exceptions; #for i "/Satellite/PUS/Service"# #for j "\$i/Sub"# WITH Service#"\$j/../@ID" "#_#"\$j/@ID"#; #end# -- next service type #end# SEPARATE (App) PROCEDURE Decode(S :IN Pus.Servicetype; P :IN Pus.Ptr) IS BEGIN ... -- select the service provider CASE S.Type IS #for i "/Satellite/PUS/Service"# WHEN #"\$i/@ID"# => CASE S.Sub IS #for j "\$i/Sub"# WHEN #"\$j/@ID"#=> Service#"\$j/../@ID" #_#"\$j/@ID"#.Exec(P); #end# WHEN OTHERS => NULL; END CASE; -- next case #end# WHEN OTHERS => NULL; END CASE; ... END Decode; </pre>	<pre> WITH Ada.Exceptions; WITH Service001_001; WITH Service001_002; -- next service type WITH Service015_001; WITH Service015_002; WITH Service015_003; -- next service type ... SEPARATE (App) PROCEDURE Decode(S :IN Pus.Servicetype; P :IN Pus.Ptr) IS BEGIN ... -- select the service provider CASE S.Type IS WHEN 001 => CASE S.Sub IS WHEN 001=> Service001_001.Exec(P); WHEN 002=> Service001_002.Exec(P); WHEN OTHERS => NULL; END CASE; -- next case WHEN 015 => CASE S.Sub IS WHEN 001=> Service015_001.Exec(P); WHEN 002=> Service015_002.Exec(P); WHEN 003=> Service015_003.Exec(P); WHEN OTHERS => NULL; END CASE; -- next case ... -- next case WHEN OTHERS => NULL; END CASE; ... END Decode; </pre>

11 Related Work

Various extensions to feature modeling were previously proposed (e.g., [CE00, HSV00]), but none of them addresses the problems solved by the extensions presented in this paper.

The use of a configuration tool to produce input for a generator is not new, but we are not aware of any other GUI-based general-purpose configuration editor based on feature modeling. OSEK/VDX is probably the most prominent example of using a

configuration tool and a generator in the automotive embedded domain. OSEK/VDX implementations include an application generator that takes an OIL file and generates the application-specific code. The OIL files can be written manually, or they can be created using graphical configuration tools, which most commercial OSEK/VDX implementations provide. However, the OSEK/VDX configuration space is hardwired in these tools, while our configuration editor is general-purpose and based on feature modeling. Thus, our technology can be used to rapidly create configuration support for new system families.

The extended design space approach of Baum [Bau01] comes closest to our work. The concept of an extended design space defines the system family configuration space and covers all the elements of the configuration knowledge from Fig. 1. The design space is stored in a database system and can be manipulated using an editor called Reboost. A graphical representation such as in our feature modeling approach is missing, however. The concept of an extended design space has been validated by applying it as a configuration front-end for QNX, a commercial operating system for embedded applications. The case study provides a tool for creating specific configurations, called D-Space-1. However, the tool can only produce QNX configurations and is not a general-purpose configuration editor.

12 Conclusions and Future Work

Our experience shows that generative programming is well suited for the industrial use in the embedded domain. However, the specific technologies being used need to fit industrial constraints such as mandated programming languages, existing validation and verification procedures, and economics of supporting system variants.

Our future work will include formalizing the representation techniques of the generative domain model and completing the modeling and configuration tools to support all the elements of the generative domain model including consistency constraints, default dependency rules, and rules for computing implementation features.

Acknowledgements. The authors would like to thank Craig Cleaveland and the anonymous reviewers for providing valuable comments on the paper.

References

- [Bli01] Frank Blinn. Entwurf und Implementierung eines Generators für Merkmalmetamodelle. Diploma thesis, IMST, University of Applied Sciences Kaiserslautern, Zweibrücken 2001 (in German)
- [BO92] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. In *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 4, October 1992, pp. 355–398
- [Bos00] J. Bosch. *Design and Use of Software Architecture: Adopting and evolving a product-line approach*. Addison-Wesley, 2000
- [Bau01] L. Baum. A Generative Approach to Customized Run-time Platforms. Ph.D. thesis, University of Kaiserslautern, 2001 (Shaker, Aachen, 2001, ISBN 3-8265-8966-1)
- [CE00] K. Czarnecki and U. Eisenecker. *Generative Programming – Methods, Tools, and Applications*. Addison-Wesley, Boston, MA, 2000

- [Cla01a] M. Clauß. Modeling Variability with UML. In online proceedings of the GCSE 2001 Young Researchers Workshop, 2001, <http://i44w3.info.uni-karlsruhe.de/~heuzer/GCSE-YRW2001/program.html>
- [Cla01b] M. Clauß. Untersuchung der Modellierung von Variabilität in UML. Diploma Thesis, Technical University of Dresden, 2001, (in German)
- [Cle01] C. Cleaveland. *Program Generators with XML and Java*. Prentice-Hall, 2001
- [Cle88] J. C. Cleaveland. Building Application Generators. In *IEEE Software*, no. 4, vol. 9, July 1988, pp. 25-33
- [CN01] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001
- [ESBC01] U. Eisenecker, M. Selbig, F. Blinn, K. Czarnecki. Feature Modeling for Software System Families. In *OBJEKTSpektrum*, No. 5, 2001, pp. 23-30 (in German)
- [GFA98] M. L. Griss, J. Favaro, and M. d'Alessandro. Integrating Feature Modeling with the RSEB. In *Proceedings of the Fifth International Conference on Software Reuse (ICSR5)*. P. Devanbu and J. Poulin, (Eds.), IEEE Computer Society Press, 1998, pp. 76-85, see www.intecs.it
- [HSV00] A. Hein, M. Schlick, R. Vinga-Martins. Applying Feature Models in Industrial Settings. In *Software Product Lines: Experience and Research Directions. (Proceedings of The First Software Product Line Conference - SPLC1)*. P. Donohoe, Kluwer Academic Publishers, 2000, pp.47-70
- [KCH+90] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, November 1990
- [LMB+01] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, P. Volgyesi. The Generic Modeling Environment. Workshop on Intelligent Signal Processing, Budapest, Hungary, May 17, 2001, <http://www.isis.vanderbilt.edu/>
- [MC] Domain-Specific Modeling: 10 Times Faster Than UML. Whitepaper by MetaCase Consulting available at <http://www.metacase.com/papers/index.html>
- [Nei80] J. Neighbors. Software construction using components. Ph. D. Thesis, (Technical Report TR-160), University of California, Irvine, 1980
- [OIL] OSEK/VDX System Generation – OIL: OSEK Implementation Language Version 2.3, September 10th, 2001, <http://www.osek-vdx.org/>
- [OSEK] OSEK/VDX Operating System Specification Version 2.2. OSEK/VDX Technical Committee, September 10th, 2001, <http://www.osek-vdx.org/>
- [Par76] D. Parnas. On the design and development of program families. In *IEEE Transactions on Software Engineering*, vol. SE-2, no. 1, 1976, pp. 1-9
- [PUS] Packet Utilisation Standard. European Space Agency, ESA PSS-07-101 Issue 1, 1994
- [Sel00] Mario Selbig. A Feature Diagram-Editor – Analysis, Design, and Implementation of its Core Functionality. Diploma Thesis, I/MST, University of Applied Sciences Kaiserslautern, Zweibrücken 2000
- [UML] OMG Unified Modeling Language Specification, Version 1.4. OMG, September 2001, www.omg.org
- [WL99] D. M. Weiss and C. T. R. Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, Reading, MA, 1999
- [Wol02] W. Wolf. What Is Embedded Computing? In *IEEE Computer*, January 2002, 136-137
- [XP] XML Path Language (XPath), Version 1.0. W3C Recommendation, November 16, 1999, www.w3.org

A Framework for the Detection and Resolution of Aspect Interactions

Rémi Douence^{1,*}, Pascal Fradet², and Mario Südholt^{1,*}

¹ École des Mines de Nantes/INRIA, Nantes, France

www.emn.fr/{douence,sudholt}

² IRISA/INRIA, Rennes, France

www.irisa.fr/lande/fradet

Abstract. Aspect-Oriented Programming (AOP) promises separation of concerns at the implementation level. However, aspects are not always orthogonal and aspect interaction is an important problem. Currently there is almost no support for the detection and resolution of such interactions. The programmer is responsible for identifying interactions between conflicting aspects and implementing conflict resolution code.

In this paper, we propose a solution to this problem based on a generic framework for AOP. The contributions are threefold: we present a formal and expressive crosscut language, two static conflict analyses and some linguistic support for conflict resolution.

1 Introduction

Separation of concerns is a valuable structuring principle for the development of software systems. Aspect-Oriented Programming (AOP) [8] promises a systematic treatment of concern separation at the implementation level. Once concerns are expressed separately in terms of different aspect definitions, one of the most fundamental problems of AOP is that of interaction between aspects, i.e., conflicts between aspects which are not orthogonal [4]. There is almost no support for the treatment of aspect interactions: the programmer is responsible for identifying interactions between conflicting aspects and for implementing conflict resolution code.

We believe that the treatment of aspect interactions should be separated from the definition of the aspects themselves. We therefore propose a three-phase model for multi-aspect programming:

1. Programming. The aspects which are part of an application are written independently, possibly by different programmers.
2. Conflict analysis. An automatic tool detects interactions among aspects and returns informative results to the programmer.

* Partially funded by the EU project “EasyComp” (www.easycomp.org), no. IST-1999-014191.

3. Conflict resolution. The programmer resolves the interactions using a dedicated composition language. The result of this phase can be checked once again as in phase 2.

The main objective of this paper is to provide support for this three-phase process. Our solution is based on a generic framework for AOP, which is characterized by a very expressive crosscut language, static conflict analyses and linguistic support for conflict resolution.

In Section 2, we formally define a general model for AOP which (conceptually) relies on a monitor observing the execution trace. Aspects consist of crosscuts matching regular expressions (i.e., sequences) of execution points, which define where the execution of the base program is modified, and inserts, which define the aspect code to be executed. Aspect weaving is modeled as an execution monitor recognizing crosscuts and applying inserts. The interactions treated in this paper occur when two crosscuts match the same point in the execution trace. In Section 3, we propose two different analyses detecting aspect interactions, independently of the base program or *w.r.t.* a specific base program. Section 4 proposes some linguistic support for the resolution of conflicts caused by aspect interactions. In particular, we introduce commands making explicit the composition of several inserts at the same execution point. We also present commands to control the visibility of aspects *w.r.t.* other aspects. These commands are taken into account by aspect transformation so that interaction analyses can still be applied to check that conflicts have been effectively resolved. We conclude by a brief review of related work and future research directions.

This paper provides a generic model for AOP that does not rely on any specific programming language. In order to provide more intuition, we illustrate our different concepts by instantiating the framework to ASPECTJ [9]. We assume a basic familiarity with AOP [8] in general and ASPECTJ in particular.

2 Framework

We model weaving as a dynamic monitor, observing the execution of the program and inserting instructions according to execution states. An aspect specifies which instructions to insert at which execution state. This study is made within a generic formal framework. In particular, we do not rely upon a particular programming language and consider a very expressive crosscut language.

We first present our model of program execution. We then introduce our aspect language that is based on crosscuts, inserts and composition operators. The operators support expressive aspect definitions and enable modeling of aspect interactions. Finally, we describe the weaver, that is to say, how executions are monitored, i.e., observed and woven.

2.1 Observable Execution and Join Points

The relevant part of an execution for weaving is called the observable execution trace. We define it using a transition relation (\rightarrow) between observable execution states. A \rightarrow -step may represent a sequence of actual execution steps. The

relation \rightarrow can be defined on the basis of a small-step semantics [11] of the base programming language. Observable states are *configurations* of the form (j, P, σ) , where j , the current join point, is an abstraction of the (static) program P and the (dynamic) execution state σ . Join points are terms which can be matched against crosscuts, i.e., term patterns. Their nature can be syntactic (e.g., instructions) but also semantic (e.g., dynamic values).

The entry and exit of a program are denoted by two special join points: \downarrow and \uparrow , respectively. The observable execution trace of a program with an initial state σ_0 is then of the form:

$$(\downarrow, P, \sigma_0) \rightarrow \dots \rightarrow (j_i, P, \sigma_i) \rightarrow \dots$$

If the reduction terminates, there exists a σ_n such that $(\downarrow, P, \sigma_0) \xrightarrow{*} (\uparrow, P, \sigma_n)$, where $\xrightarrow{*}$ denotes the transitive, reflexive closure of \rightarrow .

AspectJ: In ASPECTJ, join points denote, among others, method calls, field accesses and exception handler executions. They are represented at run time by a variable `thisJoinPoint` that contains static information (e.g., method signatures, source locations) as well as dynamic information (e.g., values of the receiver and arguments of a call). For example, when a point object (whose address is 4711) is moved to the origin through a call occurring in a line object (at address 1213), the corresponding join point can be modeled in our framework by a term:

```
call(void Point.move(int,int), within(Line),
      this(Line,1213), target(Point,4711), args(0,0))
```

■

2.2 The Aspect Language

The basic constituents of aspects are rules of the form

$$C \triangleright I$$

where C is a *crosscut* and I an *insert*. The insert I is a program that is executed whenever the crosscut C matches the current join point. Rules are combined into aspects using three operators (sequence, repetition and choice).

Crosscuts. To achieve highest expressiveness, crosscuts would be defined as arbitrary functions matching join points. However, this is too general for our purposes: we consider a more specific yet expressive crosscut language in which checking interactions is feasible.

Let us define terms as finite trees of the form

$$T ::= \mathbf{f} \ T_1 \dots T_n \mid x$$

where \mathbf{f} is an n -ary ($n \geq 0$) symbol and x is a variable. A term can be seen as a pattern to be matched on join points. The symbol \mathbf{f} can represent a syntactic element of the programming language or, more generally, an information contained within join points. Note that our aspect language is generic and can be used to define more specialized term languages (e.g., one for ASPECTJ).

A crosscut is made of conjunctions, disjunctions and negations of terms:

$$C ::= T \mid C_1 \wedge C_2 \mid C_1 \vee C_2 \mid \neg C$$

For example, using a more concrete syntax than the abstract trees denoting terms, the crosscut matching calls to a function g where one of the two arguments is a constant a can be written $g(x, a) \vee g(a, x)$.

The formulas used to express crosscuts belong to the so-called quantifier free equational formulas [3]. Whether such a formula has a solution is decidable. This is one of the key properties making the analyses described in Section 3 feasible.

The application of a crosscut to a join point j is written $C \ j$. It amounts to solving the formula obtained by replacing each term T in C by the equation $j = T$. If a crosscut does not match the program point (i.e., the formula has no solution) then we write $C \ j = \text{fail}$. If the crosscut matches the program point then we write $C \ j = \phi$ where ϕ is a substitution mapping the variables of the crosscut to their unique solution (variables with several solutions do not appear in ϕ).

We use *false* for the crosscut which does not match any join point and *true* for the crosscut that matches all join points. Let z be a fresh variable then *false* can be defined by the crosscut $z \wedge \neg z$ and *true* by z .

AspectJ: ASPECTJ's crosscuts ("pointcuts") are very close to the crosscut language we introduced. They are defined as terms containing variables ranging over values of programs and wildcards. They may be combined using the same logical operators ($\&\&$, $\mid\mid$, $!$). For example, moving points could be tracked in ASPECTJ with the crosscut definition:

```
pointcut moving(Point p, int x):target(p) && call(void move(x,*))
```

This crosscut can be translated in our framework into the pattern:

```
call(void Point.move(int,w0),within(w1),this(w2),
    target(Point p),args(x,w3))
```

where fresh variables w_i express wildcards or irrelevant information. ■

Inserts. An insert I is a term as defined above. The intuition behind a rule $C \triangleright I$ is that when the crosscut matches the current join point, i.e., $C \ j = \phi$, then ϕI is executed. Hence, $C \ j$ must yield a substitution binding all the variables of I . Any specific aspect language must ensure that ϕI is always a valid piece of code (in particular, that it does not contain undefined term variables). In the remainder of this paper we assume that all ϕI are valid.

At some places, we use the special insert **skip** that represents an instruction doing nothing. We write *always* for the rule $\text{true} \triangleright \text{skip}$ that matches any join point and does nothing and *never* for the rule $\text{false} \triangleright \text{skip}$ that does not match any join point.

AspectJ: ASPECTJ's inserts ("advice") are defined as Java code to be executed when a crosscut matches. As in our language, they may refer to the values bound to the variables occurring in the corresponding crosscut. ■

Aspects. In order to define aspects, we use a syntax similar to process calculi such as CSP. An aspect is defined by the following grammar:

$$\begin{array}{ll}
 A ::= \mu a.A & ; \text{recursive definition} \\
 | C \triangleright I ; A & ; \text{sequence} \\
 | C \triangleright I ; a & ; \text{end of sequence} \\
 | A_1 \sqcap A_2 & ; \text{choice}
 \end{array}$$

An aspect is either

- The recursive definition of an aspect $\mu a.A$ which is equivalent to the aspect A where all the occurrences of the variable a are replaced by $\mu a.A$.
- A sequence $C \triangleright I ; X$, where X is an aspect or a variable. These linear sequences always end with a variable. This is needed to ensure that aspects are regular (finite state). If the crosscut C matches the current join point, then X becomes the aspect to be woven. We consider that as soon as a rule has matched a join point, it terminates. An aspect trying to apply $C \triangleright I$ throughout the execution can be expressed as

$$\mu a.C \triangleright I ; a$$

This aspect does not evolve during the execution: such an aspect is called *stateless*. An aspect applying $C \triangleright I$ only once can be expressed as

$$C \triangleright I ; (\mu a.\text{never} ; a)$$

Indeed, as soon as $C \triangleright I$ is applied, the weaver will try to apply *never*. This is an instance of an aspect evolving according to the join points encountered. Their implementation must use some kind of state to represent this evolution. We use the term *stateful* to refer to this general form of aspects.

- A choice construction $A_1 \sqcap A_2$ which chooses the first aspect that matches a join point (the other is thrown away). If both match the same join point, A_1 is chosen. For example, the aspect trying to apply $C \triangleright I$ only on the current join point and doing nothing afterward can be expressed as

$$(C \triangleright I ; (\mu a.\text{always} ; a)) \sqcap (\mu a.\text{always} ; a)$$

If C matches the current join point, the weaver chooses the first aspect, executes the insert I and the aspect becomes $\mu a.\text{always} ; a$ that keeps doing nothing. Otherwise, the weaver chooses the second recursive aspect which is $\mu a.\text{always} ; a$ as well.

Recursive definitions, sequencing, and choices allow the specification of finite state aspects which evolve according to the join points encountered. For example, a security aspect that logs file accesses (calls to **read**) during a session (from a call to **login()** until a call to **logout()**) can be expressed as

$$\mu a_1.\text{login}() \triangleright \text{skip} ; \mu a_2.(\text{logout}() \triangleright \text{skip} ; a_1) \sqcap (\text{read}(x) \triangleright \text{addLog}(x) ; a_2)$$

where x denotes the name of the accessed file.

Aspect composition. Aspects addressing different issues (such as debugging and profiling) are composed using a parallel operator \parallel . Typically, the weaver takes a parallel composition of n aspects $A_1 \parallel \dots \parallel A_n$ and tries to apply each of them at each join point. The parallel operator is non-deterministic. For example, the composition

$$(\mu a. C_1 \triangleright I_1 ; a) \parallel (\mu a. C_2 \triangleright I_2 ; a)$$

inserts I_1 (resp. I_2) if C_1 (resp. C_2) matches the current join point. When C_1 and C_2 match the same join point, it is not specified whether I_1 is executed before I_2 or vice versa. In this case, we say that $(\mu a. C_1 \triangleright I_1 ; a)$ and $(\mu a. C_2 \triangleright I_2 ; a)$ interact.

AspectJ: ASPECTJ's aspects are rules (in the sense above) and they are repeatedly applied throughout the program execution. They can be expressed in our framework as $\mu a. C \triangleright I ; a$. Several aspects are composed in parallel (\parallel). Therefore, they may match the same join point and interact. In ASPECTJ, conflicts are resolved based on user annotations ("aspect domination") and the hierarchy of aspects. However, when two aspect are unrelated *w.r.t.* the domination or hierarchy relations, the ordering of inserts is undefined.

Composition of aspects by means of sequence and choice operators have no equivalent in ASPECTJ. The user must manually instrument advices with a state and appropriate conditions in order to simulate them. So, our crosscut language is more expressive than the crosscut language of ASPECTJ. ■

2.3 Weaving

In order to describe aspect weaving we need to introduce several auxiliary functions.

The **sel** function takes an aspect and extracts the rule to apply at the current join point j .

$$\begin{aligned} \text{sel } j (\mu a. A) &= \text{sel } j A \\ \text{sel } j (C \triangleright I ; A) &= \emptyset && \text{if } C j = \text{fail} \\ &= \{C \triangleright I\} && \text{otherwise} \\ \text{sel } j (A_1 \sqcap A_2) &= \text{sel } j A_1 && \text{if } \text{sel } j A_1 \neq \emptyset \\ &= \text{sel } j A_2 && \text{otherwise} \end{aligned}$$

The following rule extends **sel** to the parallel composition of several aspects

$$\text{sel } j (A_1 \parallel \dots \parallel A_n) = (\text{sel } j A_1) \cup \dots \cup (\text{sel } j A_n)$$

The **next** function represents the evolution of an aspect after the current join point j . It takes a composite aspect and yields the aspect to be applied to the next join point.

$$\begin{aligned} \text{next } j (\mu a. A) &= \text{next } j A[\mu a. A/a] \\ \text{next } j (C \triangleright I ; A) &= C \triangleright I ; A && \text{if } C j = \text{fail} \\ &= A && \text{otherwise} \\ \text{next } j (A_1 \sqcap A_2) &= \text{next } j A_1 && \text{if } \text{sel } j A_1 \neq \emptyset \\ &= \text{next } j A_2 && \text{if } \text{sel } j A_2 \neq \emptyset \\ &= (A_1 \sqcap A_2) && \text{otherwise} \end{aligned}$$

It is extended to the parallel composition of several aspects using the rule

$$\mathbf{next} \ j \ (A_1 \parallel \dots \parallel A_n) = (\mathbf{next} \ j \ A_1) \parallel \dots \parallel (\mathbf{next} \ j \ A_n)$$

The woven execution is performed relative to a composite aspect A (see Figure 1). The transition relation \Longrightarrow represents the woven execution. It is defined by the application of the monitor followed by a standard execution step and yields the aspect $(\mathbf{next} \ j \ A)$ to be applied to the following join point. At each join point, the applicable rules are selected ($\mathbf{sel} \ j \ A$). The monitor (relation \models) applies the selected rules in no specific order: if the crosscut of the current rule matches the current join point, the corresponding substitution is applied to the insert and ϕI is executed.

<i>Woven execution</i>	
$\frac{[j, P, \sigma] \mathbf{sel} \ j \ A \xRightarrow{*} \sigma_a \quad (j, P, \sigma_a) \rightarrow (j', P, \sigma')}{(A, j, P, \sigma) \Longrightarrow (\mathbf{next} \ j \ A, j', P, \sigma')}$	
<i>Monitor</i>	
$[j, P, \sigma]^\emptyset \models \sigma \text{ [end]}$	
$\frac{\mathcal{S} = \{C \triangleright I\} \cup \mathcal{S}' \quad C \ j = \phi \quad (\downarrow, \phi I, \sigma) \xrightarrow{*} (\uparrow, \phi I, \sigma')}{[j, P, \sigma]^\mathcal{S} \models [j, P, \sigma']^{\mathcal{S}'}} \text{ [match]}$	

Fig. 1. Dynamic weaving of silent inserts

Note that we use $\xrightarrow{*}$ to reduce inserts. This implies that inserts are not subject to weaving. In this case, we say that inserts are *silent*. Figure 2 formalizes another option which uses $\xRightarrow{*}$ to execute inserts. This makes inserts *visible* to the weaver. Since the composition of aspects may evolve during the execution of visible inserts, it must be passed to and returned by the monitor.

The programmer may want to choose whether the (inserts of an) aspect A_1 is visible for the weaving of aspect A_2 . We come back to this issue in Section 4.

Note that since no specific order of application of aspects has been specified, weaving may be non-deterministic. This situation arises when aspects interact, that is to say when $\mathbf{sel} \ j \ A$ returns a set of at least two rules. Detecting and resolving such cases is the objective of the next two sections.

AspectJ: In ASPECTJ, aspects are silent *w.r.t.* one another and the base program is visible to all aspects. So, the first version of the weaver should be used. However, when an advice calls a method of the base program, the woven version of the method is executed in ASPECTJ. Indeed, static weaving (based on program

<i>Woven execution</i>	
$\frac{[j, P, \sigma]_{\text{next } j \ A}^{\text{sel } j \ A} \xRightarrow{*} (\sigma_a, A') \quad (j, P, \sigma_a) \rightarrow (j', P, \sigma')}{(A, j, P, \sigma) \Rightarrow (A', j', P, \sigma')}$	
<i>Monitor</i>	
$[j, P, \sigma]_A^\emptyset \xRightarrow{*} (\sigma, A) \text{ [end]}$	
$\frac{\mathcal{S} = \{C \triangleright I\} \cup \mathcal{S}' \quad C \ j = \phi \quad (A, \downarrow, \phi I, \sigma) \xRightarrow{*} (A', \uparrow, \phi I, \sigma_a)}{[j, P, \sigma]_A^{\mathcal{S}} \xRightarrow{*} [j, P, \sigma_a]_{A'}^{\mathcal{S}'}} \text{ [match]}$	

Fig. 2. Dynamic weaving of visible inserts

transformation rather than execution monitoring) makes this natural. This behavior does not correspond exactly to either of the two weaver definitions given above.

In the beginning of this paper, we introduced method-call join points. On occurrence of such a join point, our weaver definitions first execute the insert followed by the base execution after the join point. This behavior corresponds to ASPECTJ's before-advice. In order to take into account ASPECTJ's advice qualifier *after*, a new kind of join point must be introduced which represents when a method returns (the insert is executed *after* the method returns). In our framework, the weaver cannot skip portions of the base program execution. So, we can only model (by means of before and after) around advices that call proceed as part of the advice. ■

3 Aspect Interactions

One of our goals is to detect when the naive parallel composition of aspects does not guarantee a deterministic weaving. We say that two aspects are independent if they do not interact (i.e., none of their crosscuts may match the same join point). Independence of two aspects is a sufficient condition to ensure that weaving is well-defined: in this case, they can be woven in any order. On the opposite, dependent aspects require the programmer to resolve the interactions.

We distinguish between two notions of independence:

- *Strong independence* does not depend on the program to be woven. The aspects are independent for all programs. The advantage of this property is that it does not have to be checked after each program modification.
- *Independence w.r.t. a program* takes into account the possible sequences of join points generated by the program to be woven. The advantage of this property compared to strong independence is that it is a weaker condition to enforce.

Note that independence (strong or *w.r.t.* a program) is a sufficient but not a necessary condition. If two crosscuts C and C' match the same join point but their corresponding inserts I and I' commute (i.e., executing I then I' is equivalent to executing I' then I) then the woven execution remains deterministic.

3.1 Strong Independence

We start by defining strong independence for crosscuts.

Definition 1. *Two crosscuts C and C' are said to be strongly independent if $C \wedge C'$ has no solution.*

This ensures that the two crosscuts can never match the same join point. When crosscuts are simple patterns (i.e., terms) strong independence amounts to checking that they are not unifiable. When the crosscuts involve negations, conjunctions and disjunctions, $C \wedge C'$ is an equational formula and remains solvable [3].

The algorithm to check strong independence of aspects is based on the laws shown in Figure 3. The algorithm, which is similar to the algorithm for finite-state product automata, terminates due to the finite-state nature of our aspects (the *(un)fold* law is used to fold already encountered aspects). We only describe here its essential properties.

$[(un)fold]$	$\mu a.A = A[\mu a.A/a]$
$[assoc]$	$(A_1 \sqcap A_2) \sqcap A_3 = A_1 \sqcap (A_2 \sqcap A_3)$
$[commut]$	$(C_1 \triangleright I_1 ; A_1) \sqcap (C_2 \triangleright I_2 ; A_2) = (C_2 \triangleright I_2 ; A_2) \sqcap (C_1 \triangleright I_1 ; A_1)$ if $C_1 \wedge C_2$ has no solution
$[elim_1]$	$C \triangleright I = false \triangleright I$ if C has no solution
$[elim_2]$	$(false \triangleright I ; A_1) \sqcap A_2 = A_2$
$[elim_3]$	$false \triangleright I ; C_1 \triangleright I_1 ; A = false \triangleright I ; A$
$[priority]$	$(C_1 \triangleright I_1 ; A_1) \sqcap (C_2 \triangleright I_2 ; A_2) = (C_1 \triangleright I_1 ; A_1) \sqcap (C_2 \wedge \neg C_1 \triangleright I_2 ; A_2)$
$[propag]$	let $A = (C_1 \triangleright I_1 ; A_1) \sqcap \dots \sqcap (C_n \triangleright I_n ; A_n)$ and $A' = (C'_1 \triangleright I'_1 ; A'_1) \sqcap \dots \sqcap (C'_m \triangleright I'_m ; A'_m)$ then $A \parallel A' = \sqcap_{i=1..n}^{j=1..m} C_i \wedge C'_j \triangleright (I_i \bowtie I'_j) ; (A_i \parallel A'_j)$ $\sqcap_{i=1..n} C_i \triangleright I_i ; (A_i \parallel A')$ $\sqcap_{j=1..m} C'_j \triangleright I'_j ; (A \parallel A'_j)$

Fig. 3. Laws for aspects

The main law is *propag* which propagates the parallel operator inside the aspect definition. It produces a sequence of choices made of all the possible pairs

of crosscuts from A and A' and all the single crosscuts of A and A' independently. Conflicts are represented using the non-deterministic function $(I_1 \bowtie I_2)$ which returns either $I_1; I_2$ or $I_2; I_1$ (where “;” denotes the sequencing operator of the programming language). The law $elim_1$ uses the algorithm of [3] to check if a crosscut has no solution in which case the crosscut is replaced by *false*. The laws $elim_2$ and $elim_3$ remove unreachable parts of an aspect. The *priority* accounts for the priority rules implicit in the choice operator. This makes the analysis more precise (e.g., using this law and $elim_2$, $(true \triangleright I_1; A_1) \square A_2$ can be rewritten into $(true \triangleright I_1; A_1)$). The laws *assoc*, *commut* and *(un)fold* serve to rewrite an aspect so that the other laws can be applied.

Definition 2. *Two aspects A and A' are said to be strongly independent if $A \parallel A'$ can be expressed as a single aspect (i.e. without \bowtie and \parallel).*

For example, the parallel composition

$$A \parallel A' = (\mu a. C \triangleright I; a) \parallel (\mu a. C' \triangleright I'; a)$$

can be rewritten using *(un)fold* twice, *propag* and *fold* again into

$$\mu a. (C \wedge C' \triangleright (I \bowtie I'); a) \square (C \triangleright I; a) \square (C' \triangleright I'; a)$$

If C and C' are independent then, using $elim_1$ and $elim_2$, it can be rewritten into

$$\mu a. (C \triangleright I; a) \square (C' \triangleright I'; a)$$

a deterministic, sequential aspect.

AspectJ: As explained in the previous section, each rule in ASPECTJ is of the form: $\mu a. C_i \triangleright I_i; a$. So, the analysis of strong independence of two aspects boils down to check the independence of two crosscuts C_1 and C_2 . For example, the analysis detects that the two following crosscuts are unifiable and therefore not strongly independent:

`call(void *.move(*, int)) and call(* Point.*(int, *))` ■

3.2 Independence *w.r.t.* a Program

Strong independence may be too strong a condition. It is sufficient to check independence *w.r.t.* the set of possible observable execution traces of a program. These traces depend on whether inserts are visible or not. We first consider the case of silent aspects, i.e., inserts which are not subject to weaving.

The precise set of execution traces is not statically computable. We assume that we have a finite approximation taking the form of a finite set of join points $\mathcal{J}(P)$ and a function

$$\text{step}_P : \mathcal{J}(P) \rightarrow \mathcal{P}(\mathcal{J}(P))$$

giving for each join point a superset of the possible successors. When the join points are purely syntactic (and when new syntax cannot be dynamically created as it is possible, e.g., using Lisp’s backquote-construction), then a possible approximation is to take all the join points of the program for $\mathcal{J}(P)$ and

step_P $j = \mathcal{J}(P)$ for every join point. This crude approximation (all join points can follow each join point) is sufficient for stateless aspects. For stateful aspects, we may rely on techniques based on control flow to get more precise approximations. To be safe, such an analysis must take into account the impact that inserts may have on the control flow of the base program.

We can specialize the parallel composition of aspects *w.r.t.* the possible sequences of join points. The function I_w formalizes such a specialization. In the following definition, we assume that A is a parallel composition of two aspects (i.e., at most two crosscuts can match a join point).

$$\begin{aligned}
 Wrti(A, j) = & \\
 \text{if } \quad \text{sel } j \ A = \emptyset & \quad \text{then } \sqcap_{j' \in (\text{step}_P \ j)} I_w(A, j') \\
 \text{else if } \text{sel } j \ A = \{C \triangleright I\} & \quad \text{then } C \triangleright I; \sqcap_{j' \in (\text{step}_P \ j)} I_w(\text{next } j \ A, j') \\
 \text{else if } \text{sel } j \ A = \{C \triangleright I, C' \triangleright I'\} & \\
 \quad \text{then } C \wedge C' \triangleright (I \bowtie I'); & \sqcap_{j' \in (\text{step}_P \ j)} I_w(\text{next } j \ A, j')
 \end{aligned}$$

The process starts with an aspect and the entry of the program \downarrow . The crosscuts matching the current join point are extracted (**sel**). The process is iterated with the new aspects (computed by **next**) and all possible successors (given by **step**). The resulting aspects are combined with the choice operator. Due to the finite-state nature of aspects and join points, there are only a finite number of reachable pairs (A, j) and I_w terminates. The laws (*un*)fold, *elim*₁ and *elim*₂ are then used to simplify the expression.

Definition 3. *Two aspects A and A' are independent w.r.t. a program P if $I_w(A \parallel A', \downarrow)$ can be expressed as a single aspect (i.e. without \bowtie and \parallel).*

If inserts are visible, join points generated by inserts must be taken into account by the control flow analysis. This requires to compute an approximation of the set of join points and the possible insertions.

Note that since (visible) inserts produce new syntax dynamically, it is even possible that the weaving process loops and introduces an unbounded number of new join points. For instance, the following profiling aspect repeatedly crosscuts any method call in order to increment a counter:

$$\mu a. (\text{call}(x.y(z)) \triangleright \text{Profiler.incrCall}()); a$$

This aspect crosscuts its own insert and weaving loops: the first method call of the base program is crosscut, so `Profiler.incrCall()` is called, which is itself crosscut, etc.

AspectJ: The two crosscuts

`call(void *.move(*, int))` and `call(* Point.*(int, *))`

are independent w.r.t. to programs which do not contain call sites corresponding to the unification of the two patterns (i.e., `call(void Point.move(int, int))`). ■

3.3 Semantic Crosscuts

Most of the crosscuts we have considered so far match syntactic information such as method calls, etc. As already suggested, crosscuts can also match semantic information, such as dynamic values. For instance, the rule that matches only join points where the first argument of `move` is zero can be expressed as $x_1.\text{move}(0, x_3) \triangleright I$. In general, the dynamic information must be encoded as terms in join points. For example, let us consider the crosscut that matches method calls to `move` if the value of the first argument of the call is even. A simple solution would be to instrument the insert with a test, such as

$$x_1.\text{move}(x_2, x_3) \triangleright \text{if } (\text{even } x_2) \text{ then } I$$

The drawback of this approach is that the conflict analysis is not able to take the parity condition into account. A more precise solution is to encode the parity information in the join point model. For example, we may enhance the join point model of ASPECTJ with a constructor **Even** to denote the parity of the arguments of a call. The join point

```
call(void Point.move(int,int), ..., args(2,3), Even(true,false))
```

makes explicit that the first and second argument of the call to `move` are respectively even and odd.

Both independence analyses can take dynamic information into account. The interaction analysis *w.r.t.* a program can perform a static analysis of the semantic properties to improve its precision.

AspectJ: ASPECTJ provides a construction `cflow(C1) && C2`. It can be expressed in our framework as:

$$\mu a_1.C_1 \triangleright \text{skip} ; \mu a_2.(\text{Ret}C_1 \triangleright \text{skip} ; a_1) \square (C_2 \triangleright I ; a_2)$$

where C_1 defines a method call join point and $\text{Ret}C_1$ defines the corresponding method-return join point. This definition can be read as: “between C_1 and $\text{Ret}C_1$, occurrences of C_2 trigger execution of I ”. However, this definition is only valid when the method denoted by C_1 is not recursive. In general, such a crosscut is semantic. In ASPECTJ, `cflow`’s implementation requires a stack in order to count (i.e., store) pending calls to C_1 .

Similarly to the parity property above, a solution is to encode in the join point the presence/absence of (at least) one pending call in the execution stack for every method in the program (e.g., using a bit vector). The conflict analysis *w.r.t.* a program could approximate this information using static analysis. For example, when `cflow(C)` is involved, we can safely assume that there is at least one call to C in the stack for every join point in the set of reachable methods from C .

Analysis of strong independence cannot make assumption about the call graph of the application. So, we must assume that every method has pending calls in the stack, and when `cflow(C1) && C2` is involved, the analysis can only consider C_2 . However, there are special cases of crosscuts involving `cflow` (such as `cflow(C1) && C2` and `!cflow(C1) && C3`) which can be shown strongly independent. ■

4 Support for Conflict Resolution

When no conflicts have been detected, the parallel composition of aspects can be woven without modifications. Otherwise, the programmer must get rid of the nondeterminism by making the composition more precise. We present here some linguistic support aimed at resolving interactions. A first kind of commands serves to specify how inserts compose. A second kind allows the user to control visibility of inserts by restricting the scope of aspects. We describe a collection of useful commands which is, however, not meant to be complete.

4.1 Composition of Inserts

The conflict analyses of Sections 3.1 and 3.2 both return aspects as results. The occurrences of rules of the form $C \triangleright (I_1 \bowtie I_2)$ indicate potential interactions.

These interactions can be resolved one by one. For each $C \triangleright (I_1 \bowtie I_2)$, the programmer may replace each rule $C \triangleright (I_1 \bowtie I_2)$ by $C \triangleright I_3$ where I_3 is a new insert which combines I_1 and I_2 in some way.

This option is flexible but can be tedious. Instead of writing a new insert for each conflict, the programmer may indicate how to compose inserts at the aspect level. We propose parallel operators of the form \parallel_f to indicate that whenever a conflict occurs in the composition $A \parallel_f A'$, the corresponding inserts must be composed using f . Of course, these operators can be combined to compose several aspects (e.g., $A \parallel_f (A' \parallel_g A'')$)

For example, when an insert I_1 of A_1 conflicts with an insert I_2 of A_2 ,

- $A_1 \parallel_{seq} A_2$ inserts $I_1; I_2$, (where “;” denotes the sequencing operator of the programming language).
- $A_1 \parallel_{fst} A_2$ inserts I_1 only.

Let us consider two aspects whose composition produces conflicts: $A_{encryption}$ crosscuts some method calls and encodes their arguments and $A_{logging}$ logs some method calls.

- $A_{logging} \parallel_{seq} A_{encryption}$ generates logs for super users by logging method calls with original arguments,
- $A_{encryption} \parallel_{seq} A_{logging}$ generates logs for users by logging method calls with possibly encrypted arguments,
- $A_{encryption} \parallel_{fst} A_{logging}$ generates logs for basic users where the encrypted methods do not appear.

Another class of commands concerns spurious conflicts. Indeed, when inserts commute in a conflict (e.g., one of the insert is **skip**), the inserts can be executed in any order. The programmer may use the command $I_1 \text{ commute } I_2$ to allow the analyzer to produce an arbitrary sequence of I_1 and I_2 .

All these assertions can be taken into account by the analyzer. If there are still conflicts, the analyzer warns the programmer that the composition is not

yet completely specified. The process can be iterated until the composition of aspects can be rewritten into a single deterministic aspect.

AspectJ: In ASPECTJ, conflicting advice can be ordered with `dominate` which is equivalent to \parallel_{seq} . The programmer must manually implement other compositions. ■

4.2 Scope of Aspects

In the weaver defined in Figure 2, the inserts are subject to weaving. This option is conflict-prone. In order to control visibility, we propose a notion of scope for aspects. The command

$$\text{scope } id \text{ } Idset \text{ } A$$

declares an aspect A with name id which can match only join points coming from an aspect whose name belongs to $Idset$. The join points of inserts are supposed to be tagged by the name of the aspects the inserts belong to. The join points of the base program are supposed to be tagged by `base`.

Scope declarations allows us to define aspects of aspects. For instance, it becomes possible to compose a profiling aspect with a security aspect in order to evaluate the cost of security tests in an application:

$$(\text{scope sec } \{\text{base}\} \text{ } A_{security}) \parallel (\text{scope prof } \{\text{sec}\} \text{ } A_{profiling})$$

In order to profile both the security aspect and the base application, we should use the following declaration

$$(\text{scope sec } \{\text{base}\} \text{ } A_{security}) \parallel (\text{scope prof } \{\text{base, sec}\} \text{ } A_{profiling})$$

We pointed out in Section 3.2 that visible inserts may lead to an infinite loop in the weaver. Preventing cycles in the scope declarations (e.g., an aspect cannot see its own inserts) is sufficient to ensure that such non-terminating weaving never occurs.

AspectJ: As mentioned at the end of Section 2, aspects are silent *w.r.t.* one another in ASPECTJ and the base program is visible to all aspects. ASPECTJ does not provide the notion of scope and cannot define aspects of aspects. ■

Our static analyses can take scopes into account by transforming the declarations into regular aspects. If a tagged join point is represented by a term $(\text{tag } j \text{ } id)$ then,

$$\text{scope } id \{id_1, id_2, id_3\} \text{ } A$$

is transformed into A where all terms T occurring in the crosscuts of A are replaced by

$$(\text{tag } T \text{ } id1) \vee (\text{tag } T \text{ } id2) \vee (\text{tag } T \text{ } id3)$$

Analysis of strong independence as described in the previous section can be applied to such transformed aspect definitions. Independence analysis *w.r.t.* a program requires the base program and join points of inserts to be annotated similarly. Note that this encoding is for static analysis purposes only. In an actual

implementation, the join points do not need to be tagged because the identity of the current insert being executed could be recorded in the execution context.

Finally, let us mention that finer-grained scope annotations can be defined easily by allowing crosscuts and inserts to be named individually.

5 Related Work and Conclusion

Despite its importance, few work has previously been done on aspect interactions and conflict resolution.

Recent releases of ASPECTJ [9] provide limited support for aspect interaction analysis using IDE integration: the base program is annotated with crosscutting aspects. This graphical information can be used to detect conflicting aspects. However, the simple crosscut model of ASPECTJ would entail an analysis detecting numerous spurious conflicts. The reason is that the relationship between several crosscuts must be maintained by book-keeping code in advice (e.g., by incrementing a counter and check for the counter value later) [6]. In our case, this kind of relationship can (sometimes) be expressed by stateful aspects and taken into account by the analysis. In case of real conflicts, ASPECTJ programmers can resolve conflicts by reordering aspects using the keyword **dominate**.

DeVolder *et al.* [12] propose a meta-programming framework based on Prolog. They specify crosscuts by predicates on abstract syntax trees and define ad-hoc composition rules for specific aspects. However, this approach does not provide a general solution to aspect interaction analysis and resolution. DeVolder's work is extended by Gybels [7] to crosscut definitions depending on dynamic values (e.g. the value of a method call argument) and optimization opportunities are discussed. However, in this case the weaving process cannot be static anymore (i.e., the weaving cannot be performed by means of inserts inlining).

Andrews [1] models AOP by means of algebraic processes. He focuses on equivalence of processes and correctness of a weaving algorithm. Non-termination problems of weaving and a formal definition of **before** and **around** are discussed but aspect interaction is not treated.

Douence *et al.* [5,6] propose another model for AOP based on execution monitoring. In this model, the crosscut language is even more expressive, in fact Turing-complete, and independence or equivalence must be proven manually.

Other approaches to the formal definition of AOP — such as Wand's *et al.* denotational semantics for a subset of ASPECTJ [13] and, to a lesser extent, Lämmel's big-step semantics formalizing method-call interception [10] — could lead to alternative approaches to interaction analysis.

Finally, note that interaction properties arise in many fields of software engineering. For instance, Batori *et al.* [2] introduce “layers” which can be compared to aspects and study composition validation using semantic conditions. It would be interesting to study whether these techniques can be adapted to control semantic interactions between aspects.

We have proposed a general method for the static analysis of aspect interactions. The paper has presented three contributions. First, we have defined a

generic formal framework for AOP featuring expressive crosscuts. Second, we have given two general independence properties and have presented how to analyze them statically. Finally, we have proposed some useful commands for conflict resolution, which is based on and compatible with the presented static analyses.

As to the application of our framework, we started to formalize parts of ASPECTJ. This task should be completed. It would also be interesting to compare the framework precisely with the denotational semantics of Wand *et al.* [13] for a subset of ASPECTJ.

Other properties and analyses could be studied in our framework for AOP. For example, in some cases, the programmer may want to check that an aspect has terminated (i.e., keeps doing nothing) before another one starts. Several linguistic extensions of the aspect language are worth further study. For example, allowing crosscuts of the same aspect to share variables would make the aspect language more expressive. Also, the possibility of associating an aspect with a class or an instance would facilitate the instantiation of the framework to object-oriented languages.

References

1. J. H. Andrews. Process-algebraic foundations of aspect-oriented programming. In *Reflection*, pages 187–209, 2001.
2. D. Batory and B. J. Geraci. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Transactions on Software Engineering (special issue on Software Reuse)*, pages 62–87, February 1997.
3. H. Comon. Disunification: A survey. In *Computational Logic: Essays in Honor of Alan Robinson*. MIT Press, Cambridge, MA, 1991.
4. C. A. Constantinides, A. Bader, and T. Elrad. Separation of concerns in concurrent software systems. In *International Workshop on Aspects and Dimensional Computing at ECOOP*, 2000.
5. R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Proceedings of the 3rd International Conference on Reflection and Crosscutting Concerns*, volume 2192 of *LNCS*. Springer Verlag, September 2001.
6. R. Douence, O. Motelet, and M. Südholt. Sophisticated crosscuts for e-commerce. ECOOP 2001 Workshop on Advanced Separation of Concerns, June 2001.
7. K. Gybels. Aspect-oriented programming using a logic meta programming language to express cross-cutting through a dynamic joinpoint structure.
8. G. Kiczales et al. Aspect-oriented programming. In *Proc. of ECOOP*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.
9. G. Kiczales et al. An overview of AspectJ. In *ECOOP*, pages 327–353, 2001.
10. R. Lämmel. A semantics for method-call interception. In *1st Int. Conf. on Aspect-Oriented Software Development (AOSD'02)*, April 2002.
11. F. Nielson and H. R. Nielson. *Semantics with Applications - A Formal Introduction*. John Wiley and Sons, New York, NY, 1992.
12. K. De Volder. Aspect-oriented logic meta programming. In Pierre Cointe, editor, *Meta-Level Architectures and Reflection, Second International Conference, Reflection'99*, volume 1616 of *LNCS*, pages 250–272. Springer Verlag, 1999.
13. M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In *FOOL 9*, pages 67–88, January 2002.

Aspect-Oriented Modeling: Bridging the Gap between Implementation and Design

Tzilla Elrad¹, Omar Aldawud², and Atef Bader²

¹ Illinois Institute of Technology, elrad@iit.edu

² Lucent Technologies, {oaldawud, abader}@lucent.com

Abstract. Separation of Concerns is one of the software engineering design principles that is getting more attention from practitioners and researchers in order to promote design and code reuse. Separation of Concerns (SoC) separates requirements such as synchronization and scheduling from the core functionality. These requirements are often referred to as crosscutting-concerns. The implementation of such requirements is scattered throughout the system, which results in the code-tangling problem. Aspect Oriented Programming provides the user with the ability to modularize, and weave crosscutting-concerns in order to maximize code reusability and solves the code-tangling problem. Weaving is the process of combining crosscutting concerns with the core functionality. Using the UML to model and inter-weave these concerns is a craft that is hard to master due to the lack of formal modeling techniques based on SoC. In this paper we present a formal design methodology to model the system's concerns based on aspect-orientation.¹

1. Introduction

Aspect-oriented (AO) technology is a maturing technology [15,12,11] that complements object-oriented (OO) technology. Though there is a general agreement that this technology is rooted back to the separation of concerns [16] by which different concerns of the software system can be designed and reasoned about in isolation from each other. The modular representation and the weaving of these concerns are the main theme of AO Software Development. Recent work in AO design [1,3,4, 12] has demonstrated the need to deploy this technology early in the software life cycle in order to utilize the technology to its full potential. Once an initial decomposition of the problem identifies software components and the corresponding aspectual properties that cut through these components, we would like to be able to express and model this initial decomposition in a formal way and carry it to the next phase of the development life cycle.

OO modeling has many benefits outlined clearly in the literature [2]. But what OO does not address is the modeling of the different design concerns as a sub-models that are easy to maintain, and comprehend. Generally, traditional OO modeling techniques tend to produce one giant model at the end of the design phase. What makes these models so problematic is the fact that identifying one design concern would become a

¹ This research is supported in part by NSF grant 557624 NSF ELRAD 41901 18 TE 02

very time consuming task. The monolithic giant OO models have motivated researchers in the AO community to investigate remedies for this problem. In [1] we demonstrated that when aspects are identified at an early stage of the development life cycle their design components are made more reusable, and automatic code generation is made possible for AOP systems. In [3] it has been argued that capturing aspects at the design phase streamlines the process of AO development. In [4] it is shown that applying the AO design to the development life cycle can help maintain consistency between requirements, design, and code.

AO modeling should have the same relationship to AOP as OO modeling has to OOP. Since UML is "the" standard modeling language for OO it is natural to use it for AO. Most of the research in AO modeling has taken this approach. The Composition Patterns approach [4] extends UML to model aspectual behavior and separates the crosscutting requirements at the design phase and maintains it throughout the development life cycle. This approach is based on ideas from the subject-oriented model [15] for composing separate designs, combined with UML templates. In [13], the authors provide a modeling technique to achieve separation of concerns using statecharts, the approach associates a statechart for classes and each operation within the class will have its own statechart to describe its behavior. However, both [4] and [13] approaches require extending UML and the associated CASE tools. In [13] UML has to be extended to support associating operations to statecharts. In [4] UML has to be extended to support binding of the composition relationships and the UML template specifications. Our approach does not require extending UML or the associated CASE tools. This unification enables off-the-shelf CASE tools to be applied to our design.

The contributions of this paper are: (1) a design methodology for modeling aspect oriented software systems using standard object-oriented modeling language "UML"; (2) a realization of the benefits of AO modeling mainly reusability, maintainability, and trace-ability; and (3) a formal methods that enables semantic preservation through transformation between design and implementation. The paper is organized around the main design methodology components, which are concepts, notation and the modeling language component. They are presented in section 2. The process described by the design methodology is addressed in section 3. Conclusion and Future research are stated in section 4.

2. Concepts, Notation, and the Modeling Language

In this section we present the main concepts, the notation (syntax and semantics) and the modeling language our design methodology is based on.

2.1. Aspect Orientations

AOP provides mechanisms for decomposing a problem into functional components and the aspectual components called aspects. Aspects are concerns that cut across the functional components. Examples are: synchronization, scheduling, logging, security, fault tolerance etc. AOP attempts to modularize aspects, and uses the weaving mechanism to combine aspects with the main functional components. In Figure 1-(a)

and (b) we show different views between object-oriented technology and aspect-oriented technology. While these aspects can be thought about and analyzed relatively separate from the core functionality, at some point before or at run time they must be woven together. Researches have taken different approaches for carrying out the weaving process (Aspect J, Hyper/J, Composition Filters, etc.). In this Paper we will present a new approach for the weaving process; this process implicitly weaves aspects to the main functional components of the system through the use of statecharts communication mechanism.

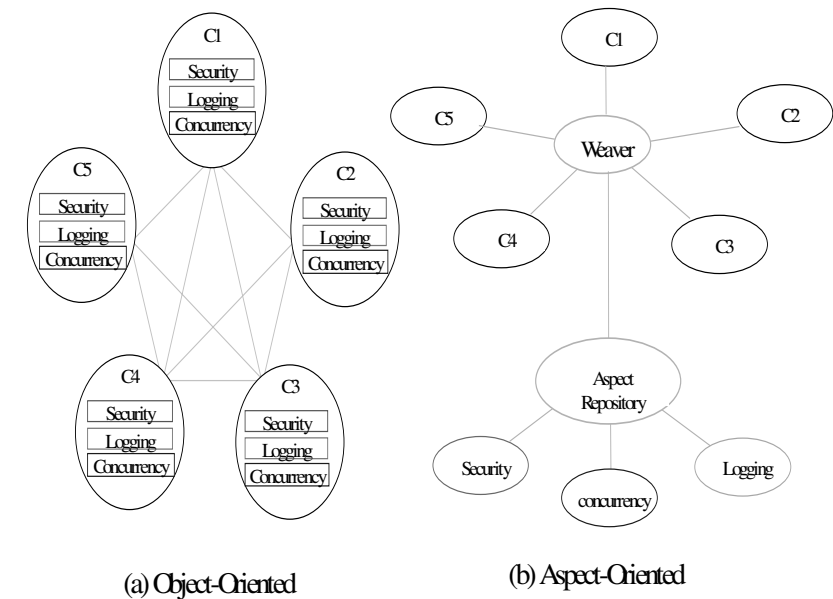


Fig. 1. Object-Oriented and Aspect Oriented Structure

2.2. UML

UML [2] is a standard modeling language for specifying, visualizing, constructing, and documenting the artifacts of object-oriented systems. UML provides several views to model the static and dynamic behavior of a software system.

Class Diagrams

UML class diagrams are used to describe the structural view (static) of classes. Both core concerns and crosscutting concerns (aspects) are represented as classes. Each core-class/aspect-class in the class diagram is associated with a statecharts that describes its dynamic behavior. When describing the structural view of the system, one must define classes and interfaces that will depict the role a class might play. Association relationships between core-classes and aspects-classes enable them to communicate. These relationships will specify how the different components of the system will be woven statically.

Statecharts

UML adopted David Harel's statecharts [7] for intra-object behavior modeling. Statecharts are "behavioral" language for the specification of real-time, event driven, and reactive systems. States in a statechart are "conditions of existence" that "persist" for a significant period of time [8]. Events trigger transitions from one state to another. They may be guarded by conditions, which must evaluate to "true" before transitioning to the next state. Actions can occur as the result of a state transition. We choose statecharts to model the behavior of objects and aspects because (1) they provide a rich set of semantics that are suitable for concurrency; (2) When using statecharts we assume a finite set of states [14], which will reduce the complexity of the system; and (3) Modeling with statecharts results in full behavioral specifications this enables semantics preserving transformation between the design and implementation and hence enables automatic code generation [5].

Modeling Orthogonal Concerns

Orthogonal regions, which are shown as dashed lines in statecharts, combine multiple simultaneous descriptions of the same object. Interactions between regions occur typically through shared variables, awareness of state changes in other regions, and message passing mechanisms [9] such as broadcasting, and propagating events. Broadcast events are events communicated to more than one orthogonal component. Propagate events are events that are signaled as a result of a transition occurring in one component. And-State is when an object is in its sub-states concurrently. Each of the concurrent sub-state is called an orthogonal component. Statechart models govern orthogonal states whereby an object must be in exactly one state from each of the orthogonal regions. For example in Figure 2 taken from [9], the object behavior is divided into two orthogonal regions (S, and T). S component must be in A or B state. T component must be in C, D, or E state. When an object receives an event, it is sent to all orthogonal regions (S, T) and each region responds to that event if it is impacted by it (i.e. when a region has a transition labeled with the event name).

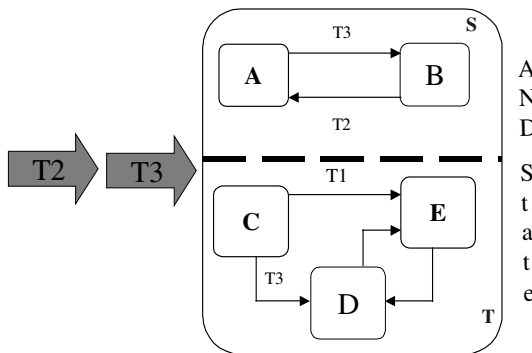


Fig. 2. Modeling Concurrency in UML

3. The Process Prescribed by Our Methodology

An important element of a sound behavioral modeling approach is a rigorous methodology that ensures that the semantics of the constructed model conform to its implementations and requirements. In this section we will introduce the process prescribed by the design methodology. Section 3.1 describes the activities that have to be carried out to apply the methodology. We will introduce the methodology by example in the following sections.

3.1. AO Modeling Steps

The Process prescribed by the methodology should describe the activities that have to be carried out to apply the methodology [6], and in what order they should progress. Also, It should describe how the steps progresses and what are the outputs of each step. Table 1 shows the modeling steps of our methodology. In the following sections we will outline how and when to apply each step and what are the output of each.

Table 1. Modeling Steps for our Methodology.

Step#	Step Description	Output
1	Identify key core objects in the system	Core-Objects
2	Identify crosscutting concerns.	Aspects-objects
3	Identify how objects and aspects relate to each other by defining association relationships between them.	Associations ²
4	Model each object and aspect as an autonomous class.	Classes
5	Draw the Class diagram based on previous 4 steps.	Class Diagram
6	Identify the set of states for each object in the object structure and the dependencies between them.	A set of states names and state dependencies
7	Use the output of step 6 to describe the behavior of each class in the class diagram using statecharts.	Statecharts (Behavioral description)
8	Use broadcasting mechanisms to send messages (events) between object's statechart using associations from Step 3.	Implicit Weaving
9	Introduce aspects as they arise orthogonal to the system and determine association relationships to existing objects and aspects.	Extensible Model that is easy to maintain and comprehend

² Messages, each message implies an association between participating objects.

3.2. The Concurrent Bounded Buffer Problem

To illustrate the practicality of our design methodology, we apply it on the design and development of the bounded-buffer example. Within the concurrent objects, state transitions may occur if a method is invoked or a timer has expired. Associating aspects that are evaluated when an object method is invoked is troublesome for the system modeler and developer. This is due to the fact that the specification and constraints of these aspects are state-dependent. Table 2 outlines the requirements that we placed on the bounded buffer system. In summary, the buffer has a limited capacity and can handle multiple read requests concurrently, while queuing all write requests. Write requests should block all other services; the blocked service will be queued until the buffer is done writing. The synchronization and scheduling requirement crosscut the core functionality of the bounded buffer.

Table 2. Bounded Buffer Requirements.

Req#	Requirement
R1	The buffer shall have a limited capacity.
R2	Items in the buffer shall be accessed via get and put methods.
R3	The system shall allow more than one reader to access the buffer at the same time.
R4	The system shall allow one writer to access the buffer at a certain time.
R5	The buffer shall Block all requests when a writer is accessing the buffer.
R6	Buffer shall block get requests when its Empty.
R7	Buffer shall block put requests when its Full.

3.3. Analysis and Design

In Table 2 requirements R6 and R7 state that we shall allow only the put method to proceed when the buffer is an EMPTY state. Either put or get can be issued when the buffer is in PARTIAL state. And we shall allow only get when the buffer is in FULL state. Requirements R6, and R7 are the synchronization requirements that control access to the bounded buffer. R3 through R5 are the main scheduling requirements, they impose certain access requirements on the system. For example, only one writer can access the system at a certain time, or more than one reader can access the system at a certain time. Synchronization constraints and scheduling specifications are the main crosscutting requirements (aspects) that influence the execution of the invoked methods based on the object state. Figure 3 shows the state machine for the system with the synchronization requirements and the scheduling policies intermixed with the main functionality of the buffer. Our approach will extract these scattered aspects and then code and model them as autonomous pieces of behavior. We found it very troublesome to associate a condition to a transition as shown in Figure 3. For example, when the buffer is in PARTIAL state and it receives a get event, the buffer

has to evaluate the synchronization constraint (if the number of items in the buffer is equal to zero) and the scheduling constraint (if there are any writers accessing the buffer, queue request) before proceeding with the transition. Depending on the outcome of both constraints the buffer might transit to the EMPTY state or remain in the PARTIAL state. We would like to eliminate the hardwiring of conditions to transitions. This hard wiring is the main cause for tangled design and hence tangled code as shown in Figure 3.

Applying Our Methodology Steps

Steps 1, and 2: So far we know that our system is composed of one main object that handles reading and writing from/to the buffer. We also know that the buffer requires synchronization and scheduling which shall be modeled as aspects. The output from step1 and step 2 is shown in Table 3.

Table 3. Output form Step1, and Step2.

Step1	Identify key objects in the system	BoundedBuffer object
Step2	Identify crosscutting concerns.	Synchronization Aspect, Scheduling Aspect

Step3, for objects to communicate, they must associate with each other. We have clearly mapped the states of the bounded buffer to the states of the scheduling aspect by a one-to-one relationship as shown in Table 4.

Table 4. Mapping Bounded Buffer states to Scheduling Aspect States.

BundedBuffer State	Scheduling Aspect State
IDLE	IDLE
Writing	Has-a-Writer
Reading	Has-a-Reader

We have also related the synchronization aspect states to the buffer’s states (FULL, PARTIAL, and EMPTY) depending on the number of items in the buffer. This mapping allows the synchronization aspect to block get requests when there are no items in the buffer (R6) and to block put methods when the buffer has reached its maximum capacity (R7). To prevent deadlocks synchronization concerns have to be resolved before scheduling concerns, which will imply temporal order between them. The output form Step 3 is shown in Table 5. Theses associations are required to enable object communication.

Table 5. Output from Step 3

Step3	Identify how objects and aspects relate to each other by defining association relationships between them.	Association: Scheduling Aspect – BoundedBuffer object. Synchronization Aspect – Scheduling Aspect.
-------	---	--

So far we have identified Objects, Aspects, and association relationships between them. Next step is to construct the Class Diagram.

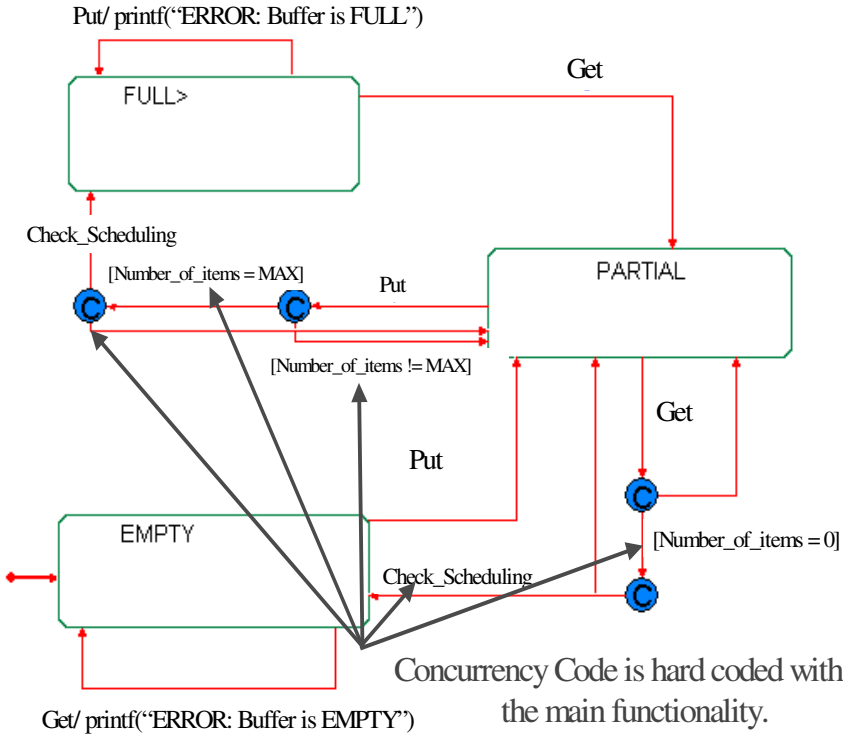


Fig. 3. The concurrent bounded buffer statecharts with crosscutting concerns intermixed with the main functionality.

3.4. Structural Description Using Class Diagrams

Class Diagrams can be used to describe the static structure of the system (classes, aspects) and define the static relationships between these objects. For objects to communicate they must associate with each other. Therefore we have to use the associations from step 3 to connect them (These associations are used to allow objects to exchange messages). We can study the structure of the model using the bounded buffer problem described earlier. The bounded buffer system consists of the main functionality object, the synchronization aspect, and the scheduling aspect (from Step 3). Figure 4 shows the class diagram for the bounded buffer, which is the output of Steps 4, and 5. Each class in the class diagram defines its own static structure including attributes, methods and interfaces. The lines between classes denote association relationships.

3.5. Modeling Crosscutting Concerns (Steps 6 through 8)

The model shown in Figure 3 does not align with the main principles of separation of concerns. It is clear that both synchronization and scheduling constraints cross cut the main functionalities of the bounded buffer. The conditions are hard wired to the transactions. We need to extract these conditions from the main functionalities of the buffer and model them as autonomous pieces of behavior (aspects). By doing so we guarantee that the bounded buffer main functionalities (design and the associated code) will not be tangled with the synchronization and scheduling aspects. Aspects now control transitions in the bounded buffer as shown in Figure 4.

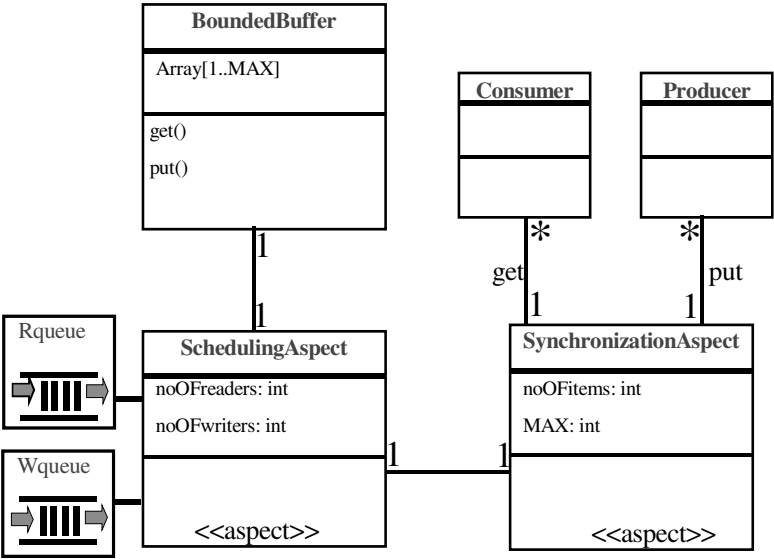


Fig. 4. Bounded Buffer Class Diagram.

Notice that extracting these cross cutting concerns from the main bounded buffer makes the main functionality clean, easy to understand. It also allows us to reuse them since the buffer deals only with reading and writing items from/to the buffer as shown in Figure 5. The states of the buffer, synchronization, and scheduling aspect are now controlled by the same events (put, and get). The synchronization aspect intercepts events by default; the arrow going into the aspect state machine region reflects this. State transitions are dependent on the aspect's state (and implicitly on the buffer's state). For example, when there are no items in the buffer the synchronization aspects should be in the EMPTY state. If a producer issues a put event, the following will occur:

- The synchronization aspect intercepts the event (Since its region is the only region that has a transition triggered by the put event) and it will change its state to PARTIAL, increase the number of items by one, and broadcast the PUT event as shown in Figure 6-A.
- Since the scheduling aspect region is the only region that has a transition triggered by the PUT event, it will change its state to hasAwriter (Assume its in IDLE state), increase the number of writers by one, and broadcast evPUT event as shown in Figure 6-B.
- Once the buffer receives the evPUT event it will change its state to writing; start writing the item into the buffer and once writing is complete, transition back to IDLE state, and broadcast event DONE as shown in Figure 6-C
- The Scheduling aspect will intercept event DONE and decrease the number of writers by one, and returns to IDLE state.

Being able to express concurrent object behavior as a statechart and augment that by notation that allows the modular to express aspects explicitly within the models is a step forward to automating the implementation and the verification of such objects.

3.6. Realizing the Benefits of Our Methodology

In this section we outline the benefits that we have realized using this design methodology mainly loose coupling, adaptability, trace-ability, and implicit weaving.

Implicit Weaving

We have demonstrated so far that we can decompose the overall system behavior into autonomous smaller pieces of behavior (aspects-objects and core-objects). Weaving is the process of combining different pieces of code (aspects code with core functionality code) into one executable module. We were able to achieve implicit weaving by means of the event broadcasting techniques that we used to send events from one state to another. Implicit weaving reduces if not eliminates coupling, which is desirable thing since it maintains separation of concern at the design as well as the code phase.

Loose Coupling

Propagating events from one object to another increases coupling between them. Broadcasting events on the other hand eliminates coupling but increases overhead. We use unique event names each time an object needs to signal an event to another object and do not want another object to respond to that event (synchronous). For example, in the bounded buffer example illustrated earlier, the synchronization aspect broadcast the GET event. The scheduling aspect implicitly (no coupling) intercepts this event since it is the only object that is affected by it. The scheduling aspect in turn will process the request and broadcast the event evGET.

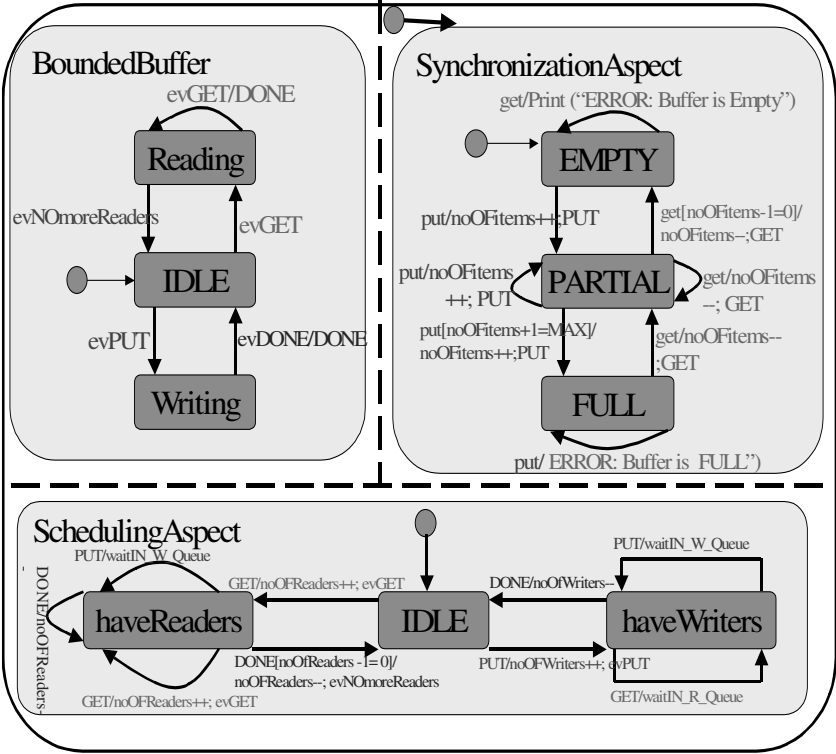


Fig. 5. Concurrent Bounded Buffer Statechart with Aspects.

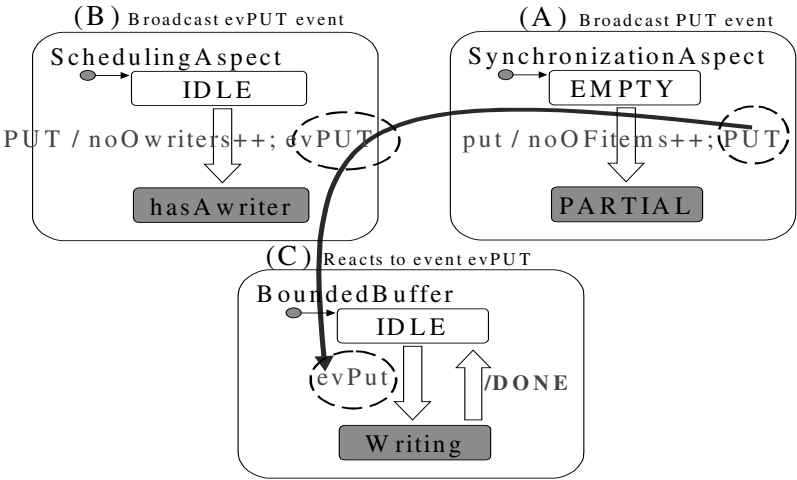


Fig. 6. Event Broadcasting

Adaptability

We can add more aspects orthogonally to the system without impacting any of the other orthogonal regions (existing aspects or main functionality). To demonstrate this, we can introduce the Error Handler aspect to the concurrent bounded buffer. The error handler aspect class and statechart are shown in Figure 7. We now need to define an association relationship(s) between the Error Handler aspect and the other aspects/objects in the system. We can define an association relationship between the synchronization aspect and the error aspect as shown in Figure 7-A.

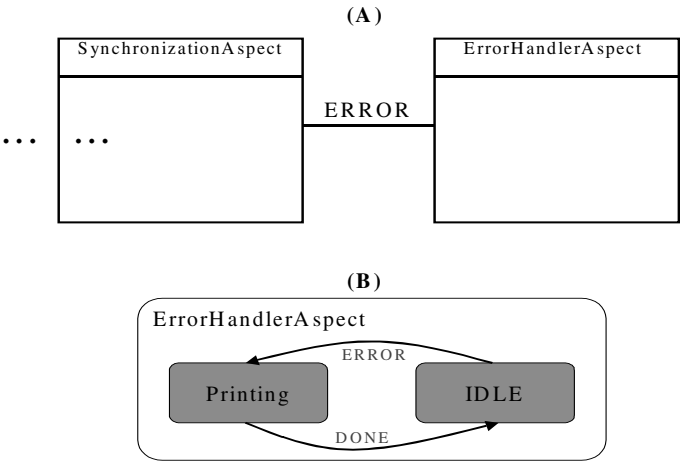


Fig. 7. Error Handler Aspect.

Figure 7-B shows the statechart for the error handler aspect. After the Error_Handler aspect has been introduced to the system’s class diagram via an association relationship with the synchronization aspect, the synchronization aspect can broadcast events to it. The synchronization aspect now broadcasts the ERROR event to resolve errors. Notice that nothing else has changed, i.e. no impact on the scheduling aspect or the buffer itself.

Trace-Ability

We have stated earlier that one of the benefits of introducing aspects at an early phase of the development life cycle is trace-ability, whereby we can trace crosscutting concerns requirements from design to code. We can easily trace the requirements in Table 2 to our design as shown in table 6.

Table 6. Requirement Trace-Ability

Req. #	Implementation
R2	Buffer Class
R3,R4,R5	Scheduling aspect
R1,R6,R7	Synchronization aspect

4. Conclusion and Future Research

AO modeling has the same relationship to AOP as OO modeling has to OOP. Maturity of AOP should lead to AO modeling and AO support at all phases of the software development life cycle. In this paper we presented a design methodology for modeling AOSD using UML's Class Diagram and Statecharts. We have identified the main components and the corresponding crosscutting components at the design phase with mechanisms to weave them. This design methodology captures these concerns at an early phase of the software life cycle and maintains their semantics across the life cycle using existing automated tools. Our initial experience shows that this approach maintains the benefits of AO across the development life cycle. In future research, we are planning to conduct an experimental study for a commercial software system to determine strengths and weaknesses of the presented approach.

References

1. Omar Aldawud, Tzilla Elrad, Atef Bader, "Aspect-Oriented Modeling to Automate the Implementation and Validation of Concurrent Software Systems", SIVOES'2001 Workshop, ECOOP 2001. <http://wooddes.intranet.gr/ecoop2001/sivoes2001.htm>
2. OMG official web site: <http://www.omg.org>
3. J. Suzuki, Y. Yamamoto, "Extending UML with Aspects: Aspect Support in the design phase" 3rd AOP Workshop at ECOOP 1999.
4. S. Claret, R.J. Walker, "Composition Patterns: An Approach to Designing Reusable Aspects", Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001). P 5-14.
5. David Harel, "From Play-In Scenarios To Code: An Achievable Dream." IEEE Computer. Preliminary version in Proc. FASE, Lecture Notes in Computer Science, Vol. 1783, Springer-Verlag, March 2000, pp. 22-34.
6. David Harel and M. Politi, "Modeling Reactive systems with StateCharts." McGraw-Hill, New York, 1994.
7. David Harel, "Statecharts": a visual formalism for complex systems, "Science of Computer Programming. Vol. 8 (1987), pp. 231-274.
8. Bruce Powel Douglass, "UML Statecharts", ESP Jan-1999." I-Logix
9. Bruce Powel Douglass, "Real-Time UML: Developing efficient Objects For Embedded Systems", Addison-Wesley, 1998.
10. Rhapsody User's Guide, I-Logix web site: <http://www.ilogix.com>
11. G. Kiczales, "An Overview of AspectJ", in proceedings of 15th ECOOP, 2001.
12. J. Gray, T. Bapty, S. Neema, and J. Tuck, "Handling Cross Cutting Constraints In Domain-Specific Modeling", Communication of the ACM October 2001/Vol. 44, No. 10 pp. 87-93.
13. C. Crichton, A. Cavarra, and Jim Davies, "A Pattern For Concurrency in UML". ICSE 2001. <http://wooddes.intranet.gr/uml2001>.
14. Pazzi, "Explicit Aspect Composition by Part-Whole Statecharts". Workshop on Separation of Concerns". ECOOP 1997.
15. T. Elrad, M. Aksit, G. Kiczales, K. Lieberherr, and Harold Ossher, "Discussing Aspects Of AOP". Communication of the ACM October 2001/Vol. 44, No. 10, pp. 33-38.
16. H. Ossher, M. Kaplan, W. Harrison, A. Katz, and V. Kruskal, "Subject-oriented composition rules," in Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices, Austin, Texas, October 1995, pp. 235-250, ACM Press.

Macros That Compose: Systematic Macro Programming

Oleg Kiselyov

Software Engineering, Naval Postgraduate School, Monterey, CA 93943
`oleg@pobox.com`, `oleg@acm.org`

Abstract. Macros are often regarded as a sort of black magic: highly useful yet abstruse. The present paper aims to make macro programming more like a craft. Using R5RS Scheme macros as an example, we develop and present a general practical methodology of building complex macros systematically, by combining simpler components by functional composition or higher-order operators.

Macro programming is complex because the systematic approach does not apply to many macros. We show that macros and other head-first normal-order re-writing systems generally lack functional composition. However, macros written in a continuation-passing style (CPS) always compose. Making CPS macros practical still requires an encoding for macro-continuations. We have found this missing piece, with an insight for anonymous macro-level abstractions.

In the specific case of R5RS macros, this paper presents a stronger result: developing R5RS macros by a translation from the corresponding Scheme procedures. We demonstrate the practical use of the technique by elaborating a real-world example.

1 Introduction

As a practical matter, macros are useful and often used, especially in large projects. Their applications range from conditional compilation to controlled inlining, to improving convenience and expressiveness of a small language core by adding syntax sugar. In the most compelling application, macros implement a domain-specific, “little” language by syntactically extending a general-purpose language [15]. Many programming systems officially or covertly include some kind of macro- or pre-processing facility. The unusual sophistication of macro systems of Scheme and Lisp is argued to be the most distinguished virtue of these languages, which is crucial for some projects [5][15]. MacroML extension for ML [4] and Camlp4 pre-processor for OCaml [13] are examples of macro facilities for modern functional languages. Macros turn out to be useful even in a non-strict language such as Haskell [16], to define and alter bindings, statements, declarations and other non-expression constructs.

However, macros have historically been poorly understood and unreliable [1]. Macros and pre-processors – i.e., abstract and concrete syntax transformers – may generate ill-typed, invalid or even unparseable code. Macros may insert

bindings that accidentally capture user variables. Macros are difficult to develop and debug. Some of these issues have been addressed: The MacroML system [4], designed from the viewpoint of macros as multi-staged computations, guarantees that the expanded code is well-formed and well-typed. Capture-free (i.e., hygienic) introduction of new bindings in generated code has been extensively investigated in the Scheme community [11]. The Revised⁵ Report on Scheme (R5RS) [7] defines a macro system with the assured hygiene.

This article however deals with a software-engineering aspect of writing macros. Embedding a domain-specific notation or otherwise syntactically extending a core language often require sophisticated macros. A complex artifact is easier to develop if it can be composed of smaller, separately designed and tested parts. Macro systems such as R5RS Scheme macros and Camlp4 quotations are head-first, call-by-name or normal order re-writing systems. Such systems are in general non-compositional. The familiar idioms of a function call and a functional composition – let alone higher-level combinators such as fold – do not easily apply. Therefore, macros tend to be monolithic, highly recursive, ad hoc, and requiring a revelation to write, and the equal revelation to understand. As Section 7.3 of R5RS demonstrates, we have to resort to contortions and hacks even in simple cases.

This paper pinpoints the stumbling block to macro composition and the way to overcome it. The resulting technique lets us program macros using traditional, well-understood applicative programming idioms of composition and combinators. The solution is a continuation-passing style (CPS) coupled with a *macro-level abstraction*. The latter is a first-class denotation for a future expansion, and is a novel insight of the paper.

We will limit the presentation to R5RS macros for the lack of space. However, the abstraction-passing technique is general and will apply to Camlp4, Tcl and other head-first code-generation systems. The application of the technique to R5RS macros leads to the other result of the paper: a systematic construction of R5RS macros by a translation from Scheme procedures. This is an unexpected result as typically R5RS macros look nothing like regular Scheme procedures.

In Section 2 we briefly outline R5RS macros as a representative head-first re-writing system. Section 3 demonstrates failures of macro compositions and identifies the reason. In the next section we introduce the main technical contribution, macro-level abstractions, and discuss their use to encode continuations in CPS macros. We establish the reason why CPS macros always compose. Section 5 introduces the systematic construction of R5RS macros. The section illustrates the translation technique by working out a real-world example. The last two sections discuss the related work and conclude.

2 R5RS Macros as a Head-First Re-writing System

Macro-systems are in general source-to-source transformations. Macro-phrases, or macro-applications, are introduced by a keyword. The keyword – an associated syntax transformer, to be precise – specifies how the rest of the phrase is to

be parsed or interpreted. When the macro processor encounters a keyword, it invokes the corresponding transformer, passes to it the macro-phrase as it is, and re-processes the result. Because the rest of a macro phrase is interpreted solely by a syntactic transformer, the phrase may contain constructs that appear invalid or ill-formed in the core language. This fact makes macros powerful tools for the extension of language syntax [12]. Throughout the paper, we will use the terms macros and head-first re-writing systems synonymously.

We will use R5RS macros as a representative head-first pattern-based re-writing system. R5RS-macro keywords are bound to transformers by `define-macro`, `let-syntax` or `letrec-syntax` forms, for example:

```
(define-syntax id
  (syntax-rules ()
    ((id x) x)))
```

A syntax transformer, specified with a `syntax-rules` form, is a sequence of patterns and the corresponding templates. Patterns may contain pattern variables and are linear. Given a form, the macro expander checks it for macro phrases, and if found, reduces them in the normal order, i.e., leftmost first. A reduction step is matching against a set of patterns and replacing the matched expression by a template. For example, given an expression `(id (id 42))`, the expander notices that it is a macro phrase, retrieves the transformer associated with the keyword `id`, and matches the expression against the only pattern in that transformer, `(id x)`. The matching process binds the pattern variable `x` to the form `(id 42)`, which, according to the template, becomes the result of the expansion. The result is itself a macro-application and is further re-written to `42`, which is the normal form of the original expression.

As another example, let us consider a macro that expands into a character-discriminator function:

```
(define-syntax mcase1
  (syntax-rules ()
    ((mcase1 chr)
     (lambda (v)
       (case v
         ((chr) #t)
         (else #f))))))
```

A macro-application `(mcase1 #\A)` expands into `(lambda (fresh-v) (case fresh-v ((#\A) #t) (else #f)))`, a predicate that checks if its argument is equal to the character `#\A`. R5RS macros are hygienic: when a macro-expansion introduces a new binding, the bound variable, e.g., `v`, is in effect renamed to prevent accidental captures of free identifiers [7].

3 Mis-composition of Macros

In the functional world, if f and g are two appropriately typed functions we can compose them to build a more complex function. If one of the functions is the

identity, the composition is equivalent to the other function: $f \cdot id \equiv f$. This property generally does not hold if f and id are macros.

Indeed, if we compose the previously introduced macros `mcase1` and `id` and evaluate `((mcase1 (id #\A)) #\A)` we get an unexpected result as if the character `#\A` is not equal to itself. The expansion of the composed form `(mcase1 (id #\A))`:

```
(lambda (vv)
  (case vv
    (((id #\A)) #t)
    (else #f)))
```

manifests the problem. According to R5RS [7], `case` is a special, library syntax form. It is defined as “(case <key> <clause1> <clause2> ...) where <key> may be any expression and each <clause> should have the form ((<datum1> ...) <expression1> <expression2> ...), where each <datum> is an external representation of some object. The last <clause> may be an `else` clause.” It must be stressed that <datum1> is not an expression and therefore is not subject to macro-expansion. The expansion of the form `(mcase1 (id #\A))` then is a procedure that matches its argument against a literal two-element-list constant `(id #\A)` rather than against the character `#\A`. The composition fails because the argument of `mcase1`, `chr`, appears in a *special class position* in `mcase1`’s expansion code template. We say that a phrase occurs in a special class position if it is not subject to macro expansion. A quoted phrase, <datum> in a `case` form, bindings in `let` forms are examples of special class position phrases [12]. A macro-argument to be deconstructed inside the macro also occupies a special class position. In Scheme, phrases in special class positions are those that are not parsed as expressions.

We argue that macros that place their arguments in special class positions constitute the most compelling class of macros. The paper [16] documents several reasons for using macros in functional languages. For example, in a strict language, we often employ macros to prevent, delay or repeat the evaluation of an expression. We can achieve the same result without macros however, by turning the expression in question into a thunk. Infix and especially distfix operators [16] can add significant amount of syntactic sugar without resorting to macros. However, transformations that: introduce bindings; insert type, module or other declarations; add alternatives to enumerated type declarations cannot be accomplished without macro facilities. Such transformations give the most compelling reason for the inclusion of macros in a language standard [16]. These transformations are also non-compositional, in general.

Re-writing systems that have special-class-position phrases and that cannot force the re-writing of argument expressions are generally non-compositional. For example, a macro may be invoked with an argument that is a macro-value such as an identifier or a string. Alternatively, a macro can be invoked with the argument that is a *macro-expression*, i.e., an application of another macro. The latter will yield a macro-value when expanded. If such an argument appears in

an expression position in the result of the original macro, the macro processor will expand the macro-expression when re-scanning the result. If an argument that is a macro-expression will eventually appear in a special class position, this macro-expression will *not* be further expanded. A macro that pattern-matches its argument against a known symbol or form will not work if it is applied to a macro-expression. This will cause syntax errors in the best case, or subtle semantic errors in the worst. The difference between head-first normal-order abstractions and applicative-order abstractions (functions) is that a function can force the evaluation of an expression by using it or (in strict systems) by passing it as an argument to a function. In a head-first normal-order re-writing system, a rule can cause the expansion of an expression only by returning it. In that case however, the rule loses the control and cannot obtain the expression's result.

The non-applicative nature of macros and other head-first systems has been tacitly acknowledged by language designers. To make macros functional, the designers often provide systematic or ad hoc ways for a macro-transformer to force the evaluation of its arguments. For example, Tcl has special evaluating parentheses [], Lisp has a `macroexpand` form, and shell has a backquote substitution. These approaches however make it difficult for the macro expander to offer hygiene or other guarantees. Therefore, R5RS syntax transformers are rather restricted; in particular, they cannot themselves invoke the macro expander – or other Scheme functions.

4 Macro-CPS Programming

We shall now show how to construct macros that are always compositional – even if they place their arguments in special positions. In some circumstances, we can compose with “legacy” macros written without following our methodology. As a side effect, the technique makes R5RS macros functional, restores the modularity principle, and even makes possible higher-order combinators. The present section introduces the technique and gives several small illustrations. The next section elaborates an extended example of a systematic, modular development of a real-life R5RS macro.

The first, novel ingredient to our macro programming technique is a notation for a *first-class parameterized future macro-expansion action*:

```
(??!lambda (bound-var ...) body)
```

Here **bound-var** is a variable, **body** is an expression that may contain forms (**??! bound-var**), and **??!lambda** is just a symbol. It must be stressed that **??!lambda** is not a syntactic or bound variable. Although the **??!lambda** form may look like a macro-application, it is not. This is a critical distinction: Hilsdale and Friedman [6] looked for a macro-level lambda as a macro, but did not succeed. Our macro-level abstractions are not macros themselves. The **??!lambda** form is first class: it can be passed to and returned from R5RS macros, and can be nested.

The `??!lambda`-form is interpreted by a macro `??!apply`. To be more precise, we specify that the following macro-application

```
(??!apply (??!lambda (bound-var ...) body) arg ...)
```

expands to `body`, with all non-shadowed instances of `(??! bound-var)` hygienically replaced by `arg`. In Scheme, question and exclamation marks are considered ordinary characters. Hence `??!lambda`, `??!apply` and `??!` are ordinary symbols – albeit oddly looking, on purpose. A `??!lambda` form can bind one or several variables; the corresponding `??!apply` form should have just as many arguments.

An implementation of `??!apply` that satisfies our specification is given on Fig. 1.

CPS macros first introduced in [6] are the second constituent of the technique. Our CPS macros must receive continuation argument(s), and must expand into an application of `??!apply` or another CPS macro. In particular, if a macro `foo` wants to ‘invoke’ a macro `bar` on an argument `args`, `foo` should expand into `(bar args (??!lambda (res) continuation))`. Here the form `continuation` includes `(??! res)` forms and thus encodes whatever processing needs to be done with the `bar`’s result.

This technique easily solves the problem of composing macros `id` and `mcase1`. We will first re-write the macro `id` in CPS:

```
(define-syntax id-cps
  (syntax-rules ()
    ((_ x k)
     (??!apply k x))))
```

This macro immediately passes its argument to its continuation. The composition of `id-cps` with `mcase1` will use a macro-level abstraction to encode the continuation:

```
(id-cps #\A (??!lambda (x) (mcase1 (??! x))))
```

We can even compose `mcase1` and `id-cps` twice:

```
(let ((discriminator
      (id-cps #\A
        (??!lambda (x)
          (id-cps (??! x)
                  (??!lambda (x) (mcase1 (??! x))))))))
  (display (discriminator #\A)))
```

When evaluated, this expression yields the expected result `#t`.

We observe that given the definition of `id-cps` and the specification of `??!apply`, the expression

```
(id-cps datum (??!lambda (x) (exp (??! x))))
```

```

(define-syntax ???!apply (syntax-rules (???!lambda)
  ((_ (???!lambda (bound-var . other-bound-vars) body)
    oval . other-ovals)
    (letrec-syntax
      ((subs
        (syntax-rules (???! bound-var ???!lambda)
          ((_ val k (???! bound-var)) (appl k val))
          ; check if bound-var is shadowed in int-body
          ((_ val k (???!lambda bvars int-body))
            (subs-in-lambda val bvars (k bvars) int-body))
          ((_ val k (x)) ; optimize single-elem list substitution
            (subs val (recon-pair val k ()) x))
          ((_ val k (x . y))
            (subs val (subsed-cdr val k x) y))
          ((_ val k x) ; x is an id other than bound-var, number&c
            (appl k x))))
        (subsed-cdr ; we've done the subs in the cdr of a pair
          (syntax-rules () ; now do the subs in the car
            ((_ val k x new-y)
              (subs val (recon-pair val k new-y) x))))
        (recon-pair ; reconstruct the pair out of subs comp
          (syntax-rules ()
            ((_ val k new-y new-x) (appl k (new-x . new-y))))
          (subs-in-lambda ; substitute inside the lambda form
            (syntax-rules (bound-var)
              ((_ val () kp int-body)
                (subs val (recon-l kp ()) int-body))
              ; bound-var is shadowed in the int-body: no subs
              ((_ val (bound-var . obvars) (k bvars) int-body)
                (appl k (???!lambda bvars int-body)))
              ((_ val (obvar . obvars) kp int-body)
                (subs-in-lambda val obvars kp int-body))))
            (recon-l ; reconstruct lambda from the substituted body
              (syntax-rules ()
                ((_ (k bvars) () result)
                  (appl k (???!lambda bvars result))))
              (appl ; apply the continuation
                (syntax-rules () ; add the result to the end of k
                  ((_ (a b c d) result) (a b c d result))
                  ((_ (a b c) result) (a b c result))))
              (finish
                (syntax-rules ()
                  ((_ () () exp) exp)
                  ((_ rem-bvars rem-ovals exps)
                    (???!apply (???!lambda rem-bvars exps) . rem-ovals))))
              )
            ; In the following, finish is the continuation...
            (subs oval (finish other-bound-vars other-ovals) body))))))

```

Fig. 1. The implementation of the ???!apply macro

yields `(exp datum)` for any expression `exp` with no free occurrence of `(?! x)`. This observation proves that `id-cps`, in contrast to the macro `id`, always composes with any macro. Furthermore, `id-cps` is the unit of the macro composition. Furthermore, the expression `(exp (?! x))` may even be an application of a legacy, non-CPS-style macro, e.g., `mcase1`. The last macro in a compositional chain may be a “third-party” macro, written without our methodology.

More examples of developing complex macros by macro composition can be found in [8]. One of the examples is a compile-time implementation of a factorial over Peano-Church numerals. No computer science paper is complete without working out the factorial. Figure 2 shows the factorial macro, excerpted from [8].

```
(define-syntax ?plc-fact
  (syntax-rules ()
    ((?plc-fact co k) ; pattern
     (?plc-zero? co
      (?!lambda (c) (?plc-succ (?! c) k)) ; k on c being zero
      (?!lambda (c) ; k on c being non-zero
        (?plc-pred (?! c) ; the predecessor
          (?!lambda (c-1)
            (?plc-fact (?! c-1)
              (?!lambda (c-1-fact)
                (?plc-mul (?! c) (?! c-1-fact) k))))))))))
```

Fig. 2. The factorial CPS macro

This fragment is quoted here to illustrate modularity: the factorial macro is built by composition of separately defined and tested arithmetic (`?plc-pred`, `?plc-mul`) and comparison macros. Selection forms such as `?plc-zero?` take several continuation arguments but continue only one. A paper [9] develops even more challenging R5RS macro: the normal-order evaluator for lambda calculus. The challenge comes from capture-free substitutions and repeated identifications of left-most redexes. The evaluator is truly modular: it relies on a separately designed and tested one-step beta-substitutor, and on a higher-order macro-combinator `map`. Our technique of CPS and macro-level abstractions guarantees composability of all pieces.

The reasons for such a guarantee are Plotkin’s simulation and indifference theorems discussed in [14][2]. Formally, composability can be defined as an observational equivalence of $f(gv)$ and $f\text{eval}(gv)$, where f , g , and v are values. The equivalence is a tautology in a call-by-value language: A call-by-value evaluator always evaluates arguments of an application. The equivalence is not assured in a call-by-name language (e.g., a macro system). For CPS terms, the equivalence that defines composability can be written as:

$$\mathcal{F}[f(gv)]\lambda k.k \equiv \mathcal{F}[gv]\lambda n.\Psi[f](\lambda k.k)n$$

$$\cong \Psi[f] (\lambda k.k) \text{eval}(\mathcal{F}[g v] \lambda k.k)$$

where \mathcal{F} and Ψ are Fisher's CPS transformers for expressions and values correspondingly [14]. The latter equivalence holds for both call-by-name eval_n and call-by-value eval_v evaluators, which means that CPS terms compose in both calculi. The proof of that assertion is based on the Plotkin's indifference and simulation theorems combined with a lemma

$$\begin{aligned} \text{eval}_v M \text{ defined} &\Rightarrow \mathcal{F}[M] =_v \lambda k.k \Psi[\text{eval}_v M] \\ \text{eval}_v M \text{ undefined} &\Rightarrow \text{eval}_v \mathcal{F}[M] k \text{ undefined, } \text{eval}_n \mathcal{F}[M] k \text{ undefined} \end{aligned}$$

The lemma is a re-statement of Plotkin's colon-translation (cf. also lemmas 3 and 4 of [2] for a more efficient CPS transformation).

5 Systematic Construction of R5RS Macros

The `?plc-fact` macro on Fig. 2 curiously looked rather similar to a regular Scheme procedure written in CPS. This observation raises a question if R5RS macros can be written systematically – by translating from the corresponding Scheme functions. This section answers affirmatively and elaborates an example of such a translation.

The problem, which was originally posed on a newsgroup `comp.lang.scheme`, is to write a macro `quotify` that should work as follows:

```
(quotify (i j k) (s4 k s5 l () (m i) (((i))))))
==> '(s4 ,k s5 l () (m ,i) (((',i))))
```

In other words, the problem is a non-destructive modification of selected leaves in an S-expression tree. We start by writing a regular Scheme procedure that implements the required transformation:

```
(define (quotify-fn syms-l tree)
  (define (doit tree)
    (map
      (lambda (node)
        (if (pair? node)
            ; recurse to process children
            (doit node)
            (if (memq node syms-l)
                ; replace the leaf
                (list 'unquote node)
                node)))
      tree))
  (list 'quasiquote (doit tree)))
```

After the procedure is verified we *mechanically* convert it to a CPS macro, using a Scheme-to-syntax-rules translator [10]. For clarity, this section will be developing the CPS version of `quotify-fn` and the corresponding macro manually

and side-by-side. A set of tables through the end of this section document all the steps. The tables juxtapose CPS procedures (in the left column) and the corresponding macros. Before we proceed with the CPS conversion of `quotify-fn`, we need CPS versions of Scheme primitives such as `car` and `cdr`. Table 1 shows these primitives and their macro counterparts. We distinguish the names of CPS macros with the leading character '?’.

Table 1. Primitive CPS functions and macros. The primitive `ifpair?` receives a value `x` and two continuations `kt` and `kf`. The primitive passes `x` to `kt` if `x` is a pair. Otherwise, `x` is passed to `kf`.

<code>(define (cons-cps a b k)</code> <code>(k (cons a b)))</code>	<code>(define-syntax ?cons</code> <code>(syntax-rules ()</code> <code>((_ x y k)</code> <code>(?!apply k (x . y))))</code>
<code>(define (car-cps x k)</code> <code>(k (car x)))</code>	<code>(define-syntax ?car</code> <code>(syntax-rules ()</code> <code>((_ (x . y) k) (?!apply k x))))</code>
<code>(define (cdr-cps x k)</code> <code>(k (cdr x)))</code>	<code>(define-syntax ?cdr</code> <code>(syntax-rules ()</code> <code>((_ (x . y) k) (?!apply k y))))</code>
<code>(define (ifpair? x kt kf)</code> <code>(if (pair? x) (kt x) (kf x)))</code>	<code>(define-syntax ?ifpair?</code> <code>(syntax-rules ()</code> <code>((_ (a . b) kt kf)</code> <code>(?!apply kt (a . b)))</code> <code>((_ non-pair kt kf)</code> <code>(?!apply kf non-pair))))</code>

The equality-comparison-and-branching primitive is a bit more complex, Table 2. The `?ifeq?` macro is the only one in this section that requires care. However, the macro does a simple task and can be written and tested independently of other code.

We can combine the primitives into more complex functions, for example, a CPS function that checks the membership of an item in a list, Table 3. The similarity between the function `memq-cps` and the corresponding macro `?memq` is striking. We also need another complex function: `map`, which is a higher-order combinator (Table 4).

This gives us all the pieces to write the `quotify-cps` function and the corresponding CPS macro, Table 5.

The macro `?quotify` is the sought answer. It is a modular macro and can be freely composed with other CPS macros. It is a R5RS, hence, hygienic macro. It is, however, ungainly, e.g.: `(?quotify (i j k) (s4 k s5 l () (m i) ((('i))) (?!lambda (r) (begin (?! r))))))`. The ease of use is an important issue for macros, since making the code look pretty is the principal

Table 2. The compare-and-branch primitive. It receives two values and two continuations, `kt` and `kf`. The primitive passes the first value to `kt` if the two values are the same. Otherwise, the first value is given to `kf`.

```
(define (ifeq? a b kt kf)
  (if (eq? a b) (kt a) (kf a)))

(define-syntax ?ifeq?
  (syntax-rules ()
    ((_ (x . y) b kt kf)
      (?!apply kf (x . y)))
    ((_ () b kt kf)
      (?!apply kf ()))
    ((_ a b _kt _kf)
      (let-syntax
        ((aux
          (syntax-rules (a)
            ((_ a kt kf)
              (?!apply kt a))
            ((_ other kt kf)
              (?!apply kf a))))))
        (aux b _kt _kf)))))
```

Table 3. Testing the list membership.

```
; if a occurs in lst, pass
; the sublist to kt. Otherwise,
; pass () to kf
(define (memq-cps a lst kt kf)
  (ifpair? lst
    (lambda (lst) ; it's a pair
      (car-cps lst
        (lambda (x)
          (ifeq? a x
            ; match
            (lambda (_)
              (kt lst))
            ; mismatch
            (lambda (_)
              (cdr-cps lst
                (lambda (tail)
                  (memq-cps
                    a tail kt kf)
                ))))))))
    (lambda (empty)
      (kf empty))))

(define-syntax ?memq
  (syntax-rules ()
    ((_ a lst kt kf)
      (?ifpair? lst
        (?!lambda (lst) ; it's a pair
          (?car lst
            (?!lambda (x)
              (ifeq? a (?! x)
                ; match
                (?!lambda (_)
                  (?!apply kt (?! lst)))
                ; mismatch
                (?!lambda (_)
                  (?cdr lst
                    (?!lambda (tail)
                      (?memq
                        a (?! tail) kt kf)
                    ))))))))
        (?!lambda (empty)
          (?!apply kf (?! empty)))))))
```

reason for their existence. Therefore, as the last step we wrap `?quotify` into a non-CPS macro:

```
(define-syntax quotify (syntax-rules ()
  ((_ args ...)
    (?quotify args ...
      ; the identity continuation: (lambda (x) x)
      (?!lambda (r) (begin (?! r))))))
```

Table 4. Higher-order CPS combinator `map-cps` and its R5RS macro counterpart `?map`

	(define-syntax ?map
	(syntax-rules ()
(define (map-cps f lst k)	((?map f lst k)
(ifpair? lst	(?ifpair? lst
; lst still has elements	; lst still has elements
(lambda (lst)	(???lambda (lst1)
(car-cps lst	(?car (??? lst1)
(lambda (x)	(???lambda (x)
(f x	(???apply f (??? x)
(lambda (fx)	(???lambda (fx)
(cdr-cps lst	(?cdr (??? lst1)
(lambda (tail)	(???lambda (tail)
(map-cps f tail	(?map f (??? tail)
(lambda (res)	(???lambda (res)
(cons-cps fx	(?cons (??? fx)
res k))	(??? res) k))
))))))))))))
; lst is empty	; lst is empty
(lambda (empty)	(???lambda (empty)
(k empty)))	(???!apply k (??? empty))))))

Table 5. CPS function `quotify-cps` and its R5RS macro counterpart `?quotify`

	(define-syntax ?quotify
	(syntax-rules (quasiquote unquote)
	((_ syms-l _tree _k)
	(letrec-syntax
	((doit
(define (quotify-cps syms-l tree k)	(syntax-rules ()
(define (doit tree k)	((_ tree k)
(map-cps	?map
(lambda (node k)	(???lambda (node mk)
(ifpair? node	(?ifpair? (??? node)
; recurse to process children	; recurse to process children
(lambda (node1)	(???lambda (node1)
(doit node1 k))	(doit (??? node1) (??? mk)))
(lambda (node1)	(???lambda (node1)
; node1 is not a pair	; node1 is not a pair
(memq-cps node1 syms-l	(?memq (??? node1) syms-l
; matches	; matches
(lambda (_	(???lambda (_
(k (list	(???apply (??? mk)
'unquote	(unquote
node1)))	(??? node1))))
; mis-matches: leave the node alone	; mis-matches
(lambda (_	(???lambda (_
(k	(???apply (??? mk)
node1))))))	(??? node1))))))
tree k))	tree k))))
(doit tree	(doit _tree
(lambda (conv-tree)	(???lambda (conv-tree)
(k (list	(???apply _k
'quasiquote	(quasiquote
conv-tree))))	(??? conv-tree))))))

The generic nature of the wrapper is noteworthy. The following expression is a usage example and a simple test of `quotify`:

```
(let ((i 'symbol-i) (j "str-j") (k "str-k"))
  (display (quotify (i j k) (s4 k s5 l () (m i) (((i))))))
  (newline))
```

When evaluated, the expression prints the expected result:

```
(s4 str-k s5 l () (m symbol-i) (((quote symbol-i))))
```

The macro `quotify` fulfills the original specification. It is easy to use on its own. If we want to compose it with other macros we should use its CPS version, `?quotify`, instead. As Tables 1-5 show, R5RS macros can indeed look just like regular Scheme procedures, and – more importantly – be systematically developed as regular, tail-recursive Scheme procedures.

6 Related Work

Employing the continuation-passing style for writing R5RS macros is not a new idea. This was the impetus of a paper “Writing macros in continuation-passing style” by Erik Hilsdale and Daniel P. Friedman [6]. The paper has noted that “The natural way to transform these [CPS] procedures into macros is to assume the existence of two macros that behave like `lambda` and `apply`. ... Unfortunately, it doesn’t work because of hygiene ... So it seems we cannot write a macro to express these syntactic continuations.” Therefore, Hilsdale and Friedman chose a different approach, which turns normally anonymous continuation abstractions into regular, named macros. The authors did not seem to regard that approach entirely satisfactory: they wrote that their technique is “akin to programming in an assembly language for macros, where we have given up not only all but one control structure (pattern selection) but also lexical closures.” The present paper demonstrated how to regain what was considered lost, including anonymous syntactic abstractions, branching and other control structures, and even higher-order combinators.

Keith Wansbrough [16] has made a compelling case for macros even in a non-strict functional language. The body of his paper however shuns function-like (i.e., parameterized) macros, which are only mentioned in a section discussing future research. The paper [16] never mentions macro compositions and the accompanying problems.

Camlp4 [13] is a Pre-Processor-Pretty-Printer for OCaml. One of its applications is extending the OCaml syntax by means of a *quotation*, which is essentially a macro. A quotation is a string-to-string or a string-to-AST transformation. The Camlp4 documentation mentions the possibility of nesting quotations but does not discuss composability. It is easy to show that a quotation that places its argument in a non-expression position cannot be composed with other quotations. The situation is analogous to that of R5RS macros. Therefore, the solution to

the composability problem proposed in the present paper applies to quotations as well.

Steve Ganz, Amr Sabry, and Walid Taha [4] introduced a novel macro system for a modern functional language that guarantees that the result of a macro-expansion is well-formed and *well-typed*. This is a highly appealing property of their MacroML system. The MacroML paper however does not address the issues of modularity and composability. It is not clear from the examples given in [4] if MacroML macros always compose.

Almost all implementations of Scheme offer another macro system, called `defmacro` or low-level macros. Like R5RS macros, `defmacro` is a head-first re-writing system. A `defmacro` syntax transformer however is a regular Scheme procedure that produces an S-expression. We can write `defmacro` transformers using all standard Scheme procedures and forms. Furthermore, some Scheme systems provide a non-standard procedure `macroexpand`, which a syntax transformer can use to force the expansion of arbitrary forms. Low-level macros in such systems are therefore composable. If the procedure `macroexpand` is not available, low-level macros can still be developed in a modular fashion, with the methodology proposed in the present paper.

Chez Scheme [3] generalized R5RS and low-level macro systems of Scheme into a syntax-case system. The latter is also a head-first re-writing system. Syntax-case transformers operate on an abstract datatype of code (called syntax-object) rather than strings or parsed trees. The transformers can analyze syntax objects with pattern matching; the transformers can also use standard Scheme procedures and syntax-object methods. Whether syntax-case macros are composable depends on the availability of a non-standard procedure `expander` for use in a transformer. The procedure `expander` is mentioned in the implementation section of [3]; however, this procedure is not a part of the syntax-case specification.

7 Conclusions

We have demonstrated that macro programming is in general non-functional. Macros that place their arguments in non-expression positions and cannot force expansion of their arguments do not compose. The fundamental engineering principle of modular design therefore does not generally apply to macros.

We have presented a methodology that makes macro programming applicative and systematic. The methodology is centered on writing macros in continuation-passing style. The novel element is the denotation for parameterized future macro-expansion actions: anonymous first-class syntactic abstractions. The insight that syntactic abstractions are not themselves macros made the development of such abstractions possible. Writing macros in CPS removes the obstacles to composability; anonymous macro abstractions encode macro-continuations and make the methodology practical. Therefore, complex macros can be constructed by combining separately developed and tested components, by composition and higher-order operators. The final result built by the CPS-

composition can be applied to the identity macro-continuation to yield an easy-to-use macro.

R5RS macros specifically admit a stronger result: a mechanical translation from the corresponding Scheme procedures. The technique consists of three steps: (1) write the required S-expression transformation in regular Scheme; (2) mechanically convert the Scheme procedure into CPS; and (3) translate the CPS procedure into a macro, replacing regular `lambdas` with macro-`lambdas` and annotating occurrences of bound variables. Coding of R5RS macros becomes no more complex than programming of regular Scheme functions. A real-world example elaborated in the paper has demonstrated the practical power and the convenience of the technique.

The approach of this paper makes programming of Scheme macros and of other related head-first re-writing systems more like a craft than a witchcraft.

Acknowledgments. I would like to thank anonymous reviewers for their helpful comments. This work has been supported in part by the National Research Council Research Associateship Program, Naval Postgraduate School, and the Army Research Office under contracts 38690-MA and 40473-MA-SP.

References

1. Clinger, W.: Macros in Scheme. *Lisp Pointers*, IV(4) (December 1991) 25–28
2. Danvy, O., Filinski, A.: Representing Control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, vol. 2, No. 4 (1992) 361–391
3. Dybvig, R. K.: Writing Hygienic Macros in Scheme with Syntax-Case. Computer Science Department, Indiana University (1992)
4. Ganz, S., Sabry, A., Taha, W.: Macros as Multi-Stage Computations: Type-Safe, Generative, Binding Macros in MacroML. *Proc. Intl. Conf. Functional Programming (ICFP'01)*, pp. 74–85. Florence, Italy, September 3–5 (2001)
5. Graham, P.: Beating the Averages. April 2001. <http://www.paulgraham.com/avg.html>
6. Hilsdale, E., Friedman, D. P.: Writing macros in continuation-passing style. Scheme and Functional Programming 2000. September 2000. <http://www.ccs.neu.edu/home/matthias/Scheme2000/hilsdale.ps>
7. Kelsey, R., Clinger, W., Rees J. (eds.): Revised5 Report on the Algorithmic Language Scheme. *J. Higher-Order and Symbolic Computation*, Vol. 11, No. 1, September 1998
8. Kiselyov, O.: Transparent macro-CPS programming. November 30, 2001. <http://pobox.com/~oleg/ftp/Scheme/syntax-rule-CPS-lambda.txt>
9. Kiselyov, O.: Re-writing abstractions, or Lambda: the ultimate pattern macro. December 16, 2001. <http://pobox.com/~oleg/ftp/Computation/rewriting-rule-lambda.txt>
10. Kiselyov, O.: A Scheme-to-syntax-rules compiler. July 5, 2002. <http://pobox.com/~oleg/ftp/Scheme/cps-macro-conv.scm>
11. Kohlbecker, E. E.: Syntactic Extensions in the Programming Language Lisp. Ph.D. Thesis. Indiana Classics University (1986)

12. Kohlbecker, E. E., Friedman, D.P., Felleisen, M., Duba, B.: Hygienic macro expansion. In Proc. ACM Conference on Lisp and Functional Programming, pp. 151–161 (1986)
13. de Rauglaudre, D.: Camlp4. Version 3.04+3, January 20, 2002.
<http://caml.inria.fr/camlp4/>
14. Sabry, A., Felleisen, M.: Reasoning about programs in continuation-passing style. Proc. 1992 ACM Conference on Lisp and Functional Programming, pp. 288–298. Technical Report 92-180, Rice University
15. Shivers, O.: A universal scripting framework, or Lambda: the ultimate ‘little language’. In: Jaffar, J., Yap, R. H. C. (eds.): Concurrency and Parallelism, Programming, Networking, and Security. Lecture Notes in Computer Science, Vol. 1179. Springer-Verlag, Berlin Heidelberg New York (1996) 254–265
16. Wansbrough, K.: Macros and Preprocessing in Haskell (1999).
<http://www.cl.cam.ac.uk/~kw217/research/misc/hssp-hw99.ps.gz>

Program Termination Analysis in Polynomial Time

Chin Soon Lee*

Datalogisk Institut, Copenhagen University, Denmark

Abstract. Recall the size-change principle for program termination: An infinite computation is *impossible*, if it would give rise to an infinite sequence of size-decreases for some data values. For an actual analysis, *size-change graphs* are used to capture the data size changes in possible program state transitions. A graph sequence that respects program control flow is called a *multipath*. The set of multipaths describe possible state transition sequences. To show that such sequences are finite, it is sufficient that the graphs satisfy *size-change termination (SCT)*: Every infinite multipath has infinite descent, according to the arcs of the graphs. This is an application of the size-change principle.

SCT is decidable, but complete for PSPACE. In this paper, we explore efficient approximations that are sufficiently precise in practice.

For size-change graphs that can loop (i.e., they give rise to an infinite multipath), and that satisfy SCT, it is usual to find amongst them an *anchor*—some graph whose infinite occurrence in a multipath causes infinite descent. The SCT approximations are based on finding and eliminating anchors, as far as possible. If the remaining graphs cannot loop, then SCT is established.

An efficient method is proposed to find anchors among *fan-in free* graphs, as such graphs occur commonly in practice. This leads to a worst-case quadratic-time approximation of SCT. An extension of the method handles general graphs and can detect some rather subtle descent. It leads to a worst-case cubic-time approximation of SCT. Experiments confirm the effectiveness and efficiency of the approximations in practice.

1 Introduction

1.1 Background and Motivation of This Work

This work began as an investigation into the problem of non-termination for the offline partial evaluation of functional programs. To achieve finite function specialization, various techniques have been proposed to ensure that static variables assume boundedly many values during specialization (see [7] in this volume and references therein). The intuition behind these techniques is: Unbounded sequences of increases in a static variable are *impossible*, if they would give rise to unbounded sequences of size-decreases for some bounded-variable values.

* This research is supported by DAEDALUS, the RTD project IST-1999-20527 of the IST Programme within the European Union's Fifth Framework Programme.

The conditions for variable boundedness seen in the literature are complicated. To understand better the forms of descent detected by them, Neil Jones proposed studying a related, but simpler condition for *program termination*, based on the reasoning: Infinite program execution is *impossible*, if it would give rise to an infinite sequence of size-decreases for some data values.

In joint work with Jones and Ben-Amram, the *size-change termination (SCT)* condition, based on the above principle, was formulated and studied for a first-order functional language [12]. SCT is stated in terms of simple structures called *size-change graphs*. They capture the data size changes in possible program state transitions (in this case, function-call transitions), and are obtained by dataflow analysis. SCT is decidable, but complete for PSPACE. In [12], we argued that this called for “sufficiently strong” approximations. They form the topic of the present article.

Fast and scalable methods for termination analysis are not widely discussed in the literature. The need for complicated supporting analyses in a practical analyzer makes exponential-time procedures difficult to avoid. For example, the Prolog analyzers Termilog [13] and Terminweb [3] implement procedures similar to a decision method for SCT, except that they manipulate procedure-call descriptions that capture the instantiatedness of variables as well as argument size relations. The abstract information can be exponentially large in the program size.

Interestingly, when this is *not* the case, fast analysis still cannot be guaranteed. This is because SCT can be tested using Termilog or Terminweb by a straightforward encoding. Such generality is not required in practice. In this article, we develop effective and efficient approximations of SCT. By studying fast supporting analyses, it might then be possible to design practical analyzers based on the approximations. Below are different reasons for this research, connected to the study of automatic program manipulation.

- **Finite function specialization** requires static variables to be bounded. This can be ensured by adapting our techniques [9,10,7].
- **Termination of unfolding** during the offline partial evaluation of functional and logic programs is addressed by our techniques [9,15,7].
- **More aggressive program transformation** may be possible given the termination of certain computations. These computations need not be retained in the transformed program just for their termination behaviour.
- **Choice of evaluation strategy** for some Prolog compilers may be guided by the results of termination analysis [14].
- **Independent verification of machine-generated programs** that are not completely trusted is desirable.

1.2 Generality of SCT

The SCT condition subsumes natural ways to reason about the termination of programs that manipulate well-founded data. It is particularly suited to functional and logic programs, for which the size relations among *inputs* and *outputs*

of subcomputations can be approximated by size analysis. The following untyped functional programs, operating on lists and natural numbers, are all terminating because every infinite sequence of function-call transitions would give rise to a sequence of parameter values whose sizes decrease infinitely. We label the function calls for easy reference.

1. **Non-recursive calls** do not affect size-change termination, although they may cause descent to begin late. Consider call 1 below.

```
rev0(zs) = 1rev1(zs, [])
rev1(xs, acc) = if xs=[] then acc else
               2rev1(tl(xs), hd(xs):acc)
```

2. **Indirect recursion**, as exhibited by the following tail-recursive program to multiply two numbers, is handled naturally.

```
mult(m,n) = 1loop1(m,0,n,0)
loop1(i1,j1,n1,a1) = if i1=0 then a1 else
                    2loop2(i1-1,n1,n1,a1)
loop2(i2,j2,n2,a2) = if j2=0 then 3loop1(i2,j2,n2,a2) else
                    4loop2(i2,j2-1,n2,a2+1)
```

3. **Nested calls** are handled by *size analysis*. Size analysis deduces that the return value of `sub(m,n)` below always has smaller size than the value of `m`.

```
quot(m,n) = if n=0 then false else
            if m<n then 0 else
            1quot(2sub(m,n),n)+1
sub(a,b) = if b=0 then a else
          3sub(a-1,b-1)
```

4. **Complex descent** in the *program states*: SCT induces a well-founded ordering on the program states in which every possible state transition decreases.

- a) *Lexical descent*. The parameters of `ack` below have sizes that descend lexically in every program state transition. To show termination, program state `ack(m,n)` is ordered below `ack(m',n')` whenever the sizes of `m` and `n` is a lexically smaller pair than the sizes of `m'` and `n'`.

```
ack(m,n) = if m=0 then n+1 else
           if n=0 then 1ack(m-1,1) else
           2ack(m-1, 3ack(m,n-1))
```

In Ex. 2, the sizes of `i` and `j`'s values also descend lexically in every sequence of program state transitions from `loop2` to `loop2`.

- b) *Descent in a sum of parameter sizes*. For the next program, the number `sx + sy + sz`, where `sx`, `sy` and `sz` are the sizes of `x`, `y` and `z`'s values, decreases on every program state transition. Plümer identifies this as an important form of descent [14].

```
p(x,y,z) = if z=[] then x else
           if y=[] then 1p(x,tl(z),y) else
           2p(z,tl(y),x)
```

- c) *Descent in the maximum over parameter sizes.* The number $\max(sx, sy)$, where sx and sy are the sizes of x and y 's values, decreases on every program state transition for the following program. This descent has been observed in a type inference algorithm [5,6].

$$\begin{aligned} q(x,y) = & \text{if } \text{hd}(\text{hd}(x)) \text{ or } \text{hd}(\text{hd}(y)) \text{ then true else} \\ & {}^1q(\text{hd}(\text{tl}(x)), \text{tl}(\text{tl}(x))) \text{ and} \\ & {}^2q(\text{hd}(\text{tl}(y)), \text{tl}(\text{tl}(y))) \end{aligned}$$

- d) *Combining different descent.* It is complicated to express the descent for the next program. Such descent is not expected in practice.

$$\begin{aligned} r(x,y,z,w) = & \text{if } w=[] \text{ then } x \text{ else} \\ & \text{if } y=[] \text{ or } z=[] \text{ then } {}^1r(x,x,1:z, \text{tl}(w)) \text{ else} \\ & {}^2r(\text{tl}(y), y, \text{tl}(z), 1:w) \end{aligned}$$

1.3 This Article

Following our earlier paper [12], we will work with a minimal first-order functional language with eager evaluation. In Sect. 2, we present its syntax and semantics. We define program state transitions (more precisely than in [12]), and relate the existence of an infinite state transition sequence to non-termination. *Size-change graphs* are bipartite graphs that describe the data size changes in possible program state transitions. In Sect. 3, we review the *size-change termination (SCT)* condition for a program's size-change graphs. As SCT is complete for PSPACE, we study viable PTIME approximations in Sect. 4. Section 5 contains some concluding remarks.

2 Programs and Abstraction of Their Behaviour

2.1 Syntax and Notations

Programs in our minimal subject language are generated by the following grammar, where $x, x_i \in \text{Par}$ and $f \in \text{Fun}$ are drawn from mutually exclusive sets of identifiers, and $op \in \text{Op}$ is a primitive operator.

$$\begin{aligned} p \in \text{Prog} & ::= d_1 \dots d_m \\ d, d_i \in \text{Def} & ::= f(x_1, \dots, x_n) = e^f \\ e, e_i, e^f \in \text{Expr} & ::= x \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mid \\ & \quad op(e_1, \dots, e_n) \mid f(e_1, \dots, e_n) \end{aligned}$$

The *definition* of function f has the form $f(x_1, \dots, x_n) = e^f$, where e^f is called the *body* of f . The number $n \geq 0$ of parameters is its *arity*, written $ar(f)$. Notation: $\text{Param}(f) = \{f^{(1)}, \dots, f^{(n)}\}$ is the set of f parameters. In examples they are named by identifiers. Parameters are assumed to be in scope when they are used. Call expressions are sometimes labelled, e.g., ${}^c f(e_1, \dots, e_n)$. Without loss of generality, function names, parameter names and callsite labels are assumed to be distinct in a program. Constants are regarded as 0-ary operators.

2.2 Programs and Their Semantics

Programs in the subject language are untyped, and interpreted according to the call-by-value semantics in Fig. 4.1. The semantic operator \mathcal{E} is defined as usual for a functional language: $\mathcal{E}[\![e]\!]\mathbf{v}$ is the value of expression e in the environment $\mathbf{v} = (v_1, \dots, v_n)$, a tuple containing the values of parameters $f^{(1)}, \dots, f^{(n)}$.

\mathcal{E} has type $\text{Expr} \rightarrow \text{Val}^* \rightarrow \text{Val}^\sharp$, where Val^* is the set of finite value sequences. Domain $\text{Val}^\sharp = \text{Val} \cup \{\perp, \text{Err}\}$ includes values, plus Err to model runtime errors, and \perp to model non-termination. Function $\text{lift} : \text{Val} \rightarrow \text{Val}^\sharp$ is the natural injection. Any input \mathbf{v} for f is assumed to have length $\text{ar}(f)$.

Definition 1. (*Termination*) Function f of program p terminates on input \mathbf{v} if $\mathcal{E}[\![e^f]\!]\mathbf{v} \neq \perp$. Function f is terminating if it terminates on all inputs. A program p is terminating if all of its functions are terminating.

Definition 2. (*Program states and computations*)

1. A local program state is a triple $(e, f, \mathbf{v}) \in \text{Expr} \times \text{Fun} \times \text{Val}^*$ such that e is a subexpression of e^f and \mathbf{v} has length $\text{ar}(f)$.
2. For local program states (e, f, \mathbf{v}) and (e', g, \mathbf{u}) , write $(e, f, \mathbf{v}) \rightsquigarrow (e', g, \mathbf{u})$ to express a subcomputation. Define the relation \rightsquigarrow by case analysis of e .
 - a) Case $e \equiv \text{if } e_1 \text{ then } e_2 \text{ else } e_3$
 - $(e, f, \mathbf{v}) \rightsquigarrow (e_1, f, \mathbf{v})$
 - $(e, f, \mathbf{v}) \rightsquigarrow (e_2, f, \mathbf{v})$ if $\mathcal{E}[\![e_1]\!]\mathbf{v} = \text{lift } u$ where $u = \text{True}$.
 - $(e, f, \mathbf{v}) \rightsquigarrow (e_3, f, \mathbf{v})$ if $\mathcal{E}[\![e_1]\!]\mathbf{v} = \text{lift } u$ where $u \neq \text{True}$.
 - b) Case $e \equiv \text{op}(e_1, \dots, e_n)$
 - $(e, f, \mathbf{v}) \rightsquigarrow (e_k, f, \mathbf{v})$ if for $1 \leq i < k : \mathcal{E}[\![e_i]\!]\mathbf{v} \notin \{\perp, \text{Err}\}$.
 - c) Case $e \equiv g(e_1, \dots, e_n)$
 - $(e, f, \mathbf{v}) \rightsquigarrow (e_k, f, \mathbf{v})$ if for $1 \leq i < k : \mathcal{E}[\![e_i]\!]\mathbf{v} \notin \{\perp, \text{Err}\}$.
 - $(e, f, \mathbf{v}) \rightsquigarrow (e^g, g, \mathbf{u})$ if each $\mathcal{E}[\![e_i]\!]\mathbf{v} = \text{lift } u_i$ and $\mathbf{u} = (u_1, \dots, u_n)$.
3. A program state is a pair $(f, \mathbf{v}) \in \text{Fun} \times \text{Val}^*$ where \mathbf{v} has length $\text{ar}(f)$.
4. A program state transition (or function-call transition) is a pair of program states $(f, \mathbf{v}) \hookrightarrow (g, \mathbf{u})$ such that there is a sequence of local program states $(e_0, f, \mathbf{v}) \rightsquigarrow (e_1, f, \mathbf{v}) \rightsquigarrow \dots \rightsquigarrow (e_T, f, \mathbf{v})$, where $e_0 = e^f$, expressions e_0, \dots, e_{T-1} are not function calls, e_T is a function call to g , each $(e_t, f, \mathbf{v}) \rightsquigarrow (e_{t+1}, f, \mathbf{v})$ is a local state transition, and $(e_T, f, \mathbf{v}) \rightsquigarrow (e^g, g, \mathbf{u})$.
5. A state transition sequence has the form $(f_0, \mathbf{v}_0) \hookrightarrow (f_1, \mathbf{v}_1) \hookrightarrow (f_2, \mathbf{v}_2) \hookrightarrow \dots$ such that each $(f_t, \mathbf{v}_t) \hookrightarrow (f_{t+1}, \mathbf{v}_{t+1})$ is a program state transition.

Remark: Clearly, neither \rightsquigarrow nor \hookrightarrow are computable relations in general. However, we will be *approximating* properties of the \hookrightarrow relation for the subject program.

Proposition 1. If program p is not terminating, there exists an infinite state transition sequence: $(f_0, \mathbf{v}_0) \hookrightarrow (f_1, \mathbf{v}_1) \hookrightarrow (f_2, \mathbf{v}_2) \hookrightarrow \dots$

2.3 Abstraction of Program Behaviour

Definition 3. (*Data sizes*)

1. We will assume a size function $|\bullet| : \text{Val} \rightarrow \mathbb{N}$.
2. For e a subexpression of e^f , we describe e as non-increasing on $f^{(i)}$ if for every input $\mathbf{v} = (v_1, \dots, v_n)$ to f , $\mathcal{E}[\![e]\!]\mathbf{v} = \text{lift } u$ implies $|v_i| \geq |u|$; we describe e as decreasing on $f^{(i)}$ if $|v_i| > |u|$.

Typically, the size of a list is the number of constructors in the list, and the size of a natural number is its value. With this size function, reducing the head of `ls` by computing `hd(hd(ls)) : tl(ls)` is size-decreasing on `ls`.

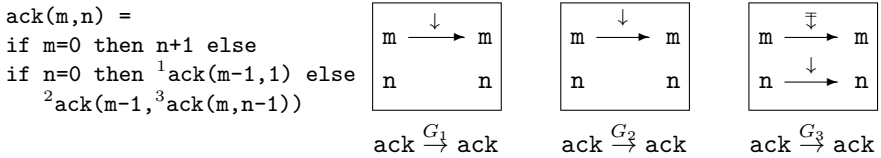
We will assume, for our examples, that `hd(x)`, `tl(x)` and `x-1` are all size-decreasing on x . This means that `hd([])` and `tl([])` should evaluate to `Err`, rather than `[]`. A semantic consequence: Attempting to apply an unbounded number of size-decreasing operations to a value results in erroneous termination.

The composition of size-decreasing operations is size-decreasing. For instance, `hd(tl(x))` is size-decreasing on x . An expression consisting of just the variable reference x is clearly non-increasing on x .

Definition 4. (*Size-change graphs*)

1. A size-change graph Γ is formally a triple $f \xrightarrow{G} g$, where G is a bipartite graph $(\text{Param}(f), \text{Param}(g), A)$, and A is a set of arcs of the form $x \xrightarrow{r} y$ with $x \in \text{Param}(f)$, $y \in \text{Param}(g)$ and $r \in \{\downarrow, \Downarrow\}$. We also loosely write $x \xrightarrow{r} y \in G$. It is assumed that $x \xrightarrow{\downarrow} y$ and $x \xrightarrow{\Downarrow} y$ are not both in G .
2. The finite set of possible size-change graphs for program p is denoted SCG_p .

A set of size-change graphs for Ex. 4a is depicted below. Each `ack` $\xrightarrow{G_i}$ `ack` describes any program state transition due to the function call at c . (Note that for a transition at callsite 2, the nested call at callsite 3 must have terminated successfully.) The graphs `ack` $\xrightarrow{G_1}$ `ack` and `ack` $\xrightarrow{G_2}$ `ack` (formally the same graph) capture function-call transitions where the size of `m` is decreased, as indicated by the arc $m \xrightarrow{\downarrow} m$. The graph `ack` $\xrightarrow{G_3}$ `ack` captures function-call transitions where the size of `m` is not increased, as indicated by the arc $m \xrightarrow{\Downarrow} m$, and the size of `n` is decreased, as indicated by the arc $n \xrightarrow{\downarrow} n$.



Definition 5. A size-change graph describes the program state transitions for which it is safe.

1. Graph $f \xrightarrow{G} g$ is safe for $(f, (v_1, \dots, v_n)) \hookrightarrow (g, (u_1, \dots, u_m))$ if for each $f^{(i)} \xrightarrow{\downarrow} g^{(j)} \in G$, $|v_i| > |u_j|$, and for each $f^{(i)} \xrightarrow{\overline{\downarrow}} g^{(j)} \in G$, $|v_i| \geq |u_j|$.
2. A set of graphs \mathcal{G} is safe for the program p if for every program state transition $(f, \mathbf{v}) \hookrightarrow (g, \mathbf{u})$, there is a graph $f \xrightarrow{G} g$ in \mathcal{G} that is safe for it.

The size-change graphs seen earlier for `ack` are clearly safe for the program.

Sets and sequences of size-change graphs. We may regard a set of size-change graph as arcs of an *annotated digraph*.

Definition 6. (*Annotated digraph*)

1. An annotated digraph $D = (V, A)$ is a pair, consisting of vertex set V , and annotated-arc set A . An annotated arc has the form $u \xrightarrow{a} v$ where $u, v \in V$ and $a \in \Sigma$ for some set of annotations Σ . For any set A , let $\text{vert}(A) = \{u, v \mid (u, v) \in A\}$.
2. Let \rightarrow_D (or just \rightarrow) denote single-step reachability: $u \rightarrow v$ if some $u \xrightarrow{a} v$ is in A . The reachability relation \rightarrow^* is the reflexive transitive closure of \rightarrow .
3. A subgraph of D is some $D' = (V', A')$ such that $V' \subseteq V$ and $A' \subseteq A$.
4. Annotated digraph D is strongly-connected if for all $u, v \in V : u \rightarrow^* v$. A set of arcs A is strongly-connected if $(\text{vert}(A), A)$ is strongly-connected.
5. For a set of arcs A , define $\text{SCC}(A)$ to be the set of maximal strongly-connect subsets, or strongly-connected components (SCCs), of A .

Definition 7. An annotated call graph (ACG) for program p is an annotated digraph whose vertex set is Fun and whose arc set is some $\mathcal{G} \subseteq \text{SCG}_p$.

Definition 8. (*Multipaths*)

1. A multipath \mathcal{M} is a non-empty, finite or infinite sequence of size-change graphs: $f_0 \xrightarrow{G_1} f_1, \dots, f_t \xrightarrow{G_{t+1}} f_{t+1}, \dots$. If the graphs are members of \mathcal{G} , then \mathcal{M} is known as a \mathcal{G} -multipath. We also write $f_0 \xrightarrow{G_1} f_1 \xrightarrow{G_2} f_2 \dots$.
2. A thread th in \mathcal{M} is any finite or infinite connected sequence of size-change arcs: $x_0 \xrightarrow{r_1} x_1, \dots, x_t \xrightarrow{r_{t+1}} x_{t+1}, \dots$ such that there exists $T \geq 0$, for each $d = 1, 2, \dots : x_{d-1} \xrightarrow{r_d} x_d \in G_{T+d}$. We also write $th = x_0 \xrightarrow{r_1} x_1 \xrightarrow{r_2} x_2 \dots$.
3. Thread th has infinite descent if $r_d = \downarrow$ for infinitely many d .
4. Multipath \mathcal{M} has infinite descent if it has a thread with infinite descent.
5. A thread is complete if it starts at the beginning of the multipath, and has the same length as the multipath.

Referring to the picture seen earlier for the size-change graphs of `ack`, consider the multipath $\text{ack} \xrightarrow{G_1} \text{ack} \xrightarrow{G_2} \text{ack} \xrightarrow{G_3} \text{ack}$. The entire picture may be regarded as depicting this multipath. Observe the complete thread: $m \xrightarrow{\downarrow} m \xrightarrow{\downarrow} m \xrightarrow{\overline{\downarrow}} m$.

Role of size analysis. To obtain a safe set of size-change graphs, we could include a graph for each callsite c , safe for the possible function-call transitions originating there. Suppose that ${}^c g(e_1, \dots, e_n)$ occurs in e^f . We would include some graph $f \xrightarrow{G_\varepsilon} g$. For the arcs of G_c , we include $f^{(i)} \xrightarrow{\downarrow} g^{(j)}$ if e_j is decreasing on $f^{(i)}$ and $f^{(i)} \xrightarrow{\Downarrow} g^{(j)}$ if e_j is non-increasing on $f^{(i)}$, as deduced by size analysis. Without global analysis, we could still obtain a safe set of graphs as follows.

Definition 9. The natural graphs \mathcal{G}_p for subject program p contains a size-change graph $f \xrightarrow{G_\varepsilon} g$ for each call ${}^c g(e_1, \dots, e_n)$ in e^f . The graph G_c contains the arc $f^{(i)} \xrightarrow{\Downarrow} g^{(j)}$ if e_j is $f^{(i)}$ and $f^{(i)} \xrightarrow{\downarrow} g^{(j)}$ if e_j is a sequence of destructive operators: $\text{hd}, \text{tl}, \dots - 1$ applied to $f^{(i)}$, e.g., $\text{hd}(\text{tl}(f^{(i)}))$.

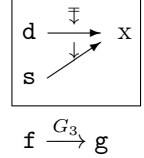
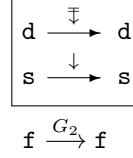
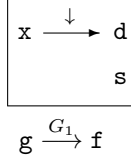
It is also *safe* to incorporate information from the conditions of **if** expressions. The code below has the structure of a program transformed by a binding-time improvement called The Trick [8]. Any transition $(\mathbf{f}, (v_1, v_2)) \hookrightarrow (\mathbf{g}, (u_1))$ is such that $|v_2| > |u_1|$ and $|v_1| = |u_1|$ (since for the transition to exist, $u_1 = \mathcal{E}[\text{hd}(\mathbf{s})](v_1, v_2) = \mathcal{E}[\mathbf{d}](v_1, v_2) = v_1$). This is a natural way for “fan-ins” (multiple arcs connected to the same parameter) to arise in our framework.

Example 5.

```

g(x) = if x=0 then 0 else
      1f(x-1, [0,1,2])+1

f(d,s) = if d=hd(s) then
          3g(hd(s)) else
          2f(d,tl(s))
    
```



3 Size-Change Termination

Definition 10. A set of size-change graphs \mathcal{G} satisfies size-change termination (SCT) if every infinite \mathcal{G} -multipath has infinite descent.

Theorem 1. Let \mathcal{G} be safe for p . If \mathcal{G} satisfies SCT, then p is terminating.

Proof. Suppose that \mathcal{G} satisfies SCT, but p is *not* terminating. By Proposition 1, there exists an infinite transition sequence $(f_0, \mathbf{v}_0) \hookrightarrow (f_1, \mathbf{v}_1) \hookrightarrow (f_2, \mathbf{v}_2) \dots$. By the safety of \mathcal{G} , there exists a \mathcal{G} -multipath \mathcal{M} : $f_0 \xrightarrow{G_1} f_1 \xrightarrow{G_2} f_2 \dots$ such that each $f_t \xrightarrow{G_{t+1}} f_{t+1}$ is safe for $(f_t, \mathbf{v}_t) \hookrightarrow (f_{t+1}, \mathbf{v}_{t+1})$. By SCT, \mathcal{M} has a thread th with infinite descent: $x_0 \xrightarrow{r_1} x_1 \xrightarrow{r_2} x_2 \dots$ such that for some T , and $d = 1, 2, \dots$, it holds that $x_{d-1} \xrightarrow{r_d} x_d \in G_{T+d}$. Define a value sequence u_0, u_1, \dots corresponding to th : Suppose x_i is $f_{T+i}^{(k)}$, and $\mathbf{v}_{T+i} = (v_1, \dots, v_n)$, then let $u_i = v_k$. By safety, the arc from x_{d-1} to x_d implies $|u_{d-1}| > |u_d|$ if $r_d = \downarrow$, and $|u_{d-1}| \geq |u_d|$ if $r_d = \Downarrow$. Since th has infinite descent, infinitely many of the r_d are \downarrow . This implies a sequence of data values whose sizes decrease infinitely, which is impossible. \square

Definition 11. The size-change graphs $f_0 \xrightarrow{G_1} f_1$ and $f_1 \xrightarrow{G_2} f_2$ can be composed to summarize the size changes among the variable values at the start and end of a state transition sequence modelled by $f_0 \xrightarrow{G_1} f_1 \xrightarrow{G_2} f_2$. Formally, the composition $f_0 \xrightarrow{G_1} f_1; f_1 \xrightarrow{G_2} f_2 = f_0 \xrightarrow{G} f_2$, where $G = G^\downarrow \cup G^\mp$ and

$$\begin{aligned} G^\downarrow &= \{x \xrightarrow{\downarrow} z \mid x \xrightarrow{r_1} y \in G_1, y \xrightarrow{r_2} z \in G_2, r_1 \text{ or } r_2 \text{ is } \downarrow\} \\ G^\mp &= \{x \xrightarrow{\mp} z \mid x \xrightarrow{\mp} y \in G_1, y \xrightarrow{\mp} z \in G_2, x \xrightarrow{\downarrow} z \notin G^\downarrow\} \end{aligned}$$

Definition 12. For $\mathcal{G} \subseteq SCG_p$, the composition closure of \mathcal{G} , denoted $\overline{\mathcal{G}}$, is the smallest set satisfying: $\overline{\mathcal{G}} = \mathcal{G} \cup \{f_0 \xrightarrow{G_1} f_1; f_1 \xrightarrow{G_2} f_2 \mid f_0 \xrightarrow{G_1} f_1, f_1 \xrightarrow{G_2} f_2 \in \mathcal{G}\}$.

Theorem 2. ([12]) \mathcal{G} satisfies SCT iff for every size-change graph $f \xrightarrow{G} f$ in $\overline{\mathcal{G}}$ such that $f \xrightarrow{G} f; f \xrightarrow{G} f = f \xrightarrow{G} f$, we have $x \xrightarrow{\downarrow} x \in G$ for some x .

It follows that SCT is decidable. Unfortunately, SCT is *intrinsically hard*.

Theorem 3. ([12]) SCT satisfaction is complete for PSPACE.

4 Polynomial-Time Approximations

The PSPACE-hardness result is motivation for studying viable polynomial-time approximations. The approximations we propose are centered around the ACG.

Definition 13. Let S be any finite set. The infinity set of an infinite sequence $s_1 s_2 \dots$ of S elements is the set $\{s \mid \{i \mid s_i = s\} \text{ is infinite}\}$.

Definition 14. Let \mathcal{H} be a set of size-change graphs. Call $f \xrightarrow{G} g \in \mathcal{H}$ an anchor for \mathcal{H} if every \mathcal{H} -multipath whose infinity set contains $f \xrightarrow{G} g$ has infinite descent. \mathcal{H} has an anchor means it contains some graph that is an anchor for it.

Lemma 1. Suppose that every non-empty strongly-connected $\mathcal{H} \subseteq \mathcal{G}$ has an anchor, then \mathcal{G} satisfies SCT.

Lemma 2. Suppose that \mathcal{H} is strongly-connected, and $\mathcal{A} \subseteq \mathcal{H}$ is a non-empty set of anchors for \mathcal{H} . If some non-empty strongly-connected subset \mathcal{H}^* of \mathcal{H} is without anchors, then \mathcal{H}^* is a subset of a member of $SCC(\mathcal{H} \setminus \mathcal{A})$.

Lemma 2 leads to the following procedure to approximate SCT for \mathcal{G} , provided we have some means to determine anchors for a strongly-connected \mathcal{H} . The procedure *SCP* (size-change polytime) below is called with each $\mathcal{H} \in SCC(\mathcal{G})$.

Procedure $SCP(\mathcal{H})$

1. Determine anchors \mathcal{A} for \mathcal{H} .
2. If \mathcal{A} is empty, fail.
3. Otherwise call SCP on each member of $SCC(\mathcal{H} \setminus \mathcal{A})$.

The procedure terminates, as \mathcal{H} is reduced on each recursive invocation. If it does not fail, argue that \mathcal{G} satisfies SCT as follows. If \mathcal{G} does *not* satisfy SCT, Lemma 1 guarantees a non-empty strongly-connected \mathcal{H}^* without anchors. At the start of the procedure, \mathcal{H} is equal to \mathcal{G} , so it includes \mathcal{H}^* . For each invocation of the procedure, if \mathcal{H} includes \mathcal{H}^* , the input to some recursive call of SCP is a superset of \mathcal{H}^* by Lemma 2. Therefore, if the procedure does not fail, \mathcal{H}^* is eventually discovered, and this would cause it to fail.

4.1 Deducing Anchors

For this, we introduce a few notions centered around the parameter set. Forward (backward) *thread preservers* for \mathcal{H} are parameters that are always forward (backward) connected to other thread preservers when they occur among the source (target) parameters in any graph of \mathcal{H} .

Definition 15. (*Thread preservers*) Let $P \subseteq \text{Par}$. For size-change graphs \mathcal{H} , define $\overrightarrow{TP}_P(\mathcal{H})$, $\overleftarrow{TP}_P(\mathcal{H})$ to be the largest sets such that:

$$\begin{aligned} \overrightarrow{TP}_P(\mathcal{H}) &= \{x \in P \mid \forall f \xrightarrow{G} g \in \mathcal{H} : \\ &\quad x \in \text{Param}(f) \Rightarrow \exists y \in \overrightarrow{TP}_P(\mathcal{H}) : x \xrightarrow{r} y \in G \text{ for some } r\} \\ \overleftarrow{TP}_P(\mathcal{H}) &= \{y \in P \mid \forall f \xrightarrow{G} g \in \mathcal{H} : \\ &\quad y \in \text{Param}(g) \Rightarrow \exists x \in \overleftarrow{TP}_P(\mathcal{H}) : x \xrightarrow{r} y \in G \text{ for some } r\} \end{aligned}$$

The subscript P is omitted if it is the set Par . $\overrightarrow{TP}_P(\mathcal{H})$ and $\overleftarrow{TP}_P(\mathcal{H})$ are called respectively the forward and backward preservers of \mathcal{H} with respect to P .

The *size-relation graph due to \mathcal{H}* is the digraph with vertex set Par and arcs corresponding to all the arcs in the size-change graphs of \mathcal{H} .

Definition 16. (*SRG*) The size-relation graph (SRG) due to \mathcal{H} , denoted $SRG_{\mathcal{H}}$, has the annotated arcs $\{x \xrightarrow{r}_{\Gamma} y \mid \Gamma = f \xrightarrow{G} g \in \mathcal{H}, x \xrightarrow{r} y \in G\}$.

Deducing anchors for fan-in free graphs.

Definition 17. (*Fan-in free*)

1. A size-change graph $f \xrightarrow{G} g$ is fan-in free if $x \xrightarrow{r} y, x' \xrightarrow{r'} y \in G \Rightarrow x = x'$.
2. A set of size-change graphs \mathcal{H} is fan-in free if all its graphs are fan-in free.

Experiments using Terminweb's size analysis indicate that both its polyhedron analysis and the simpler analysis using monotonicity and equality constraints frequently generate fan-in free graphs. This is despite the fact that equality constraints among variables (due to unifications) are common in Prolog.

Theorem 4. *Let \mathcal{H} be fan-in free and strongly-connected. If for $f \xrightarrow{G} g$ in \mathcal{H} , there exists $x \xrightarrow{\downarrow} y \in G$ with $x, y \in \overrightarrow{TP}(\mathcal{H})$, then $f \xrightarrow{G} g$ is an anchor for \mathcal{H} .*

Proof. As \mathcal{H} is strongly-connected, there is a multipath $f_0 \xrightarrow{G_1} f_1 \dots \xrightarrow{G_{n+1}} f_{n+1}$ with $f_0 = f_{n+1}$, containing every graph of \mathcal{H} . Let $P_i = \text{Param}(f_i) \cap \overrightarrow{TP}(\mathcal{H})$ for each i . By definition of $\overrightarrow{TP}(\mathcal{H})$ and the assumption that \mathcal{H} is fan-in free, $|P_i| \leq |P_{i+1}|$. Therefore $|P_0| \leq \dots \leq |P_{n+1}| = |P_0|$. Let $K = |P_0|$.

Consider any infinite \mathcal{H} -multipath $\mathcal{M} = f_0 \xrightarrow{G_1} f_1 \xrightarrow{G_2} f_2 \dots$. As before, let $P_t = \text{Param}(f_t) \cap \overrightarrow{TP}(\mathcal{H})$ for each t . Then each P_t has cardinality K . It is easy to see that the arcs from each P_t to P_{t+1} form K complete threads over \mathcal{M} . Suppose that $f \xrightarrow{G} g$ is in the infinity set of \mathcal{M} . By the premise, there exists some $x \xrightarrow{\downarrow} y \in G$ with $x, y \in \overrightarrow{TP}(\mathcal{H})$. This arc appears infinitely often among the K threads. Consequently, some thread has infinite descent. \square

Remark: The above proof gives insight into the form of descent inhibiting unbounded repetitions of $f \xrightarrow{G} g$ in any \mathcal{H} -multipath representing a state transition sequence. The sum of P_t parameter sizes is never increased, and strictly decreased for a transition described by $f \xrightarrow{G} g$, due to the decreasing arc in G .

Lemma 3. (Ben-Amram) *Let $\mathcal{H} \subseteq \mathcal{G}$ be non-empty, fan-in free and strongly-connected. And let N and M be the space requirement for \mathcal{G} and \mathcal{H} . Then $\overrightarrow{TP}(\mathcal{H})$ can be computed in time $O(M)$, with an $O(N^2)$ initialization performed on \mathcal{G} .*

The algorithm for computing $\overrightarrow{TP}(\mathcal{H})$ maintains a set of counts for the source parameters in each size-change graph of \mathcal{H} . These counts initially record the parameter out-degrees. Start by marking every parameter whose count is 0, and adding these parameters to a worklist. When processing x from the worklist, inspect those graphs with arcs connected to x . For such an arc $x' \xrightarrow{r} x$, if x' is not yet marked, reduce its count. If the count becomes 0, mark x' and insert it into the worklist. Terminate when the worklist is empty. We claim, without proof, that upon termination, the unmarked parameters form $\overrightarrow{TP}(\mathcal{H})$, and that the algorithm has $O(M)$ time-complexity, with an $O(N^2)$ initialization performed on \mathcal{G} . (Initialization is needed to set up the necessary indexing structures.)

The procedure *SCP* considers each graph of \mathcal{G} in no more than $|\mathcal{G}| \sim O(N)$ invocations. Each invocation of the procedure involves computing some $\overrightarrow{TP}(\mathcal{H})$, working out \mathcal{A} and computing $SCC(\mathcal{H} \setminus \mathcal{A})$. All these steps have linear time complexity. Therefore we have an $O(N^2)$ procedure to approximate SCT for fan-in free \mathcal{G} . We refer to the resulting approximation of SCT as SCP1.

An example. Consider the graphs \mathcal{G} for **ack** seen earlier. Now, \mathcal{G} is strongly-connected. We work out that $\overrightarrow{TP}(\mathcal{G}) = \{\mathbf{m}\}$ (\mathbf{m} is connected to \mathbf{m} in each graph of \mathcal{G}). We deduce that $\mathbf{ack} \xrightarrow{G_1} \mathbf{ack}$ and $\mathbf{ack} \xrightarrow{G_2} \mathbf{ack}$ are anchors for \mathcal{G} , as they have the arc $\mathbf{m} \xrightarrow{\downarrow} \mathbf{m}$. Intuitively, this means their occurrence in the infinity set of a \mathcal{G} -multipath would give the multipath infinite descent due to the decreasing arc.

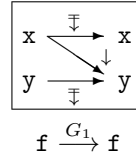
Removing these graphs from consideration leaves $\mathcal{H} = \{\mathbf{ack} \xrightarrow{G_3} \mathbf{ack}\}$. We work out that $\overrightarrow{TP}(\mathcal{H}) = \{\mathbf{m}, \mathbf{n}\}$, and deduce that $\mathbf{ack} \xrightarrow{G_3} \mathbf{ack}$ is an anchor for \mathcal{H} , as it has the arc $\mathbf{n} \xrightarrow{\downarrow} \mathbf{n}$. We conclude that \mathcal{G} satisfies SCT.

In general, if \mathcal{G} satisfies SCP1, then in every function-call transition sequence of the form $(f, \mathbf{v}) \hookrightarrow^* (f, \mathbf{u})$ for the subject program, the parameter tuples \mathbf{v} and \mathbf{u} exhibit lexical descent in various sums of parameter sizes. SCP1 subsumes most practical forms of descent, including those of Ex. 1 to Ex. 4b in Sect. 1.2.

General graphs considered. The above method to deduce anchors suffers from obvious defects. To begin with, there does not appear to be any simple extension to cope with fan-ins. We consider the issue in greater detail.

Example 6.

$\mathbf{f}(\mathbf{x}, \mathbf{y}) = \text{if } \mathbf{y}=\mathbf{tl}(\mathbf{x}) \text{ then } \mathbf{!f}(\mathbf{x}, \mathbf{y}) \text{ else } 0$



Now \mathcal{G} is strongly-connected, $\overrightarrow{TP}(\mathcal{G}) = \{\mathbf{x}, \mathbf{y}\}$, and in the graph $\mathbf{f} \xrightarrow{G_1} \mathbf{f}$, G_1 contains $\mathbf{x} \xrightarrow{\downarrow} \mathbf{y}$, but $\mathcal{M} = \mathbf{f} \xrightarrow{G_1} \mathbf{f} \xrightarrow{G_1} \mathbf{f} \dots$ has no infinite descent. Indeed, \mathcal{M} models the state transition sequence: $(\mathbf{f}, ([1], [])) \hookrightarrow (\mathbf{f}, ([1], [])) \hookrightarrow \dots$. Intuitively, when size-change graphs are not fan-in free, a descending arc between \overrightarrow{TP} parameters does not necessarily constitute “progress.” In the example, the arc $\mathbf{x} \xrightarrow{\downarrow} \mathbf{y}$ in G_1 indicates a size relationship between the values of \mathbf{x} and \mathbf{y} that persists throughout the state transition sequence.

It seems sensible to consider the thread behaviour in $\{\mathbf{x}\}$ and $\{\mathbf{y}\}$ separately. A little reflection reveals that arcs not in the same SCC of the SRG due to \mathcal{G} cannot give rise to anchor graphs. Therefore, we might define the following *critical parameter sets*, to be inspected independently for descent.

Definition 18. Define the critical parameter sets for size-change graphs \mathcal{H} as: $\text{Critical}(\mathcal{H}) = \{P \mid \overleftarrow{TP}_P(\mathcal{H}) = P, P \text{ strongly-connected in } \text{SRG}_{\mathcal{H}}, P \text{ maximal}\}$.

If P is a critical set of \mathcal{H} , then at the end of a state transition sequence described by any \mathcal{H} -multipath, the value of each P parameter has size no greater than the initial value of *some* P parameter. We will look for descent among such parameters. Observe that for fan-in free graphs, $\overleftarrow{TP}(\mathcal{H})$ at least includes $\overrightarrow{TP}(\mathcal{H})$. The use of a larger set of parameters makes it possible to detect more descent, but also means more complicated deductions (see Theorem 5).

Deducing anchors for general graphs.

Theorem 5. Let \mathcal{H} be strongly-connected. Then $f \xrightarrow{G} g$ in \mathcal{H} is an anchor for \mathcal{H} if there exists $P \in \text{Critical}(\mathcal{H})$ such that one of the following holds.

1. For $f_0 \xrightarrow{G_1} f_1$ in \mathcal{H} : $x \xrightarrow{r} y, x \xrightarrow{r'} y' \in G_1$ and $x, y, y' \in P$ implies $y = y'$ (i.e. no “fan-outs” among $x \in P$) AND there exists $x \xrightarrow{\downarrow} y \in G$ with $x, y \in P$.
2. The digraph with arcs $\{x \xrightarrow{\Gamma} y \in A \mid x, y \in P\}$, where A are $SRG_{\mathcal{H}}$ arcs, has no strongly-connected subgraph with an arc $x \xrightarrow{\Gamma} y$ where $\Gamma = f \xrightarrow{G} g$.

Refer to [11] for a proof. The first condition detects the same form of descent as the condition for fan-in free graphs in Theorem 4. When the backward thread-preserving P contains fan-outs, it is difficult to know whether the decreasing arcs among P parameters are significant for descent. The second condition implements a weak principle for identifying anchors: Any infinite \mathcal{H} -multipath has an infinite thread th remaining in just the P parameters (easily shown), so if there are no *non-descending* threads remaining infinitely among $x \in P$ in any multipath whose infinity set has $f \xrightarrow{G} g$, then th is infinitely descending in any multipath whose infinity set *does* contain $f \xrightarrow{G} g$, i.e., this graph is an anchor for \mathcal{H} . This reasoning is effective when there are sufficiently many decreasing arcs among $x \in P$. For example, the natural graphs of Ex. 4c in Sect. 1.2 are handled. Note the rather subtle descent in this example.

Lemma 4. *Let $\mathcal{H} \subseteq \mathcal{G}$ be non-empty and strongly-connected. Let N and M be the space requirement for \mathcal{G} and \mathcal{H} . Then $Critical(\mathcal{H})$ can be computed in time $O(M)$ for fan-in free \mathcal{H} and $O(M^2)$ in general, with an $O(N^2)$ initialization performed on \mathcal{G} .*

The critical sets of \mathcal{H} are computed as follows. Initialize \wp to $\{\overleftarrow{TP}(\mathcal{H})\}$. In the iterative step, replace each $P \in \wp$ with possibly several non-empty $\overleftarrow{TP}_{P'}(\mathcal{H})$, where P' are strongly-connected components in the SRG due to \mathcal{H} , restricted to P . This is repeated until no further change occurs. The calculation of $\overleftarrow{TP}_{P'}(\mathcal{H})$ is similar to the calculation of $\overleftarrow{TP}(\mathcal{H})$. The total \overleftarrow{TP} calculations for all P' in the current \wp can be accomplished in time $O(M)$, so the critical sets of \mathcal{H} can be worked out in time $O(M^2)$, as there are at most M iterative steps. The required initialization can be performed in time $O(N^2)$. To see that critical sets can be computed in time $O(M)$ for fan-in free graphs, we require the following.

Lemma 5. ([11]) *Let \mathcal{H} be non-empty, fan-in free and strongly-connected. If $P = \overleftarrow{TP}(\mathcal{H})$ and P' is a strongly-connected component in the SRG due to \mathcal{H} , restricted to P , then either $\overleftarrow{TP}_{P'}(\mathcal{H}) = \{\}$ or $\overleftarrow{TP}_{P'}(\mathcal{H}) = P'$.*

Thus no iteration of $\overleftarrow{TP}_{P'}(\mathcal{H})$ calculations is necessary for fan-in free graphs. The computation of critical sets is the dearest part of the procedure to deduce anchors. The conditions of Theorem 5 can be checked in time $O(M)$. It follows that the *SCP* procedure can be performed in time $O(N^2)$ for fan-in free graphs, and $O(N^3)$ in general. We refer to the resulting approximation of SCT as *SCP2*.

An example with fan-ins. Consider the graphs \mathcal{G} of Ex. 5 seen earlier. The initial step to determine anchors for \mathcal{G} finds the only critical set $\{\mathbf{x}, \mathbf{d}\}$. This blocks out \mathbf{s} , which cannot contribute to infinite descent in any multipath whose infinity set contains $\mathbf{g} \xrightarrow{G_1} \mathbf{f}$ or $\mathbf{f} \xrightarrow{G_3} \mathbf{g}$ (any thread visiting \mathbf{s} at some point in such a multipath is eventually continued to the $\{\mathbf{x}, \mathbf{d}\}$ component of the SRG, never to return to \mathbf{s}). By either condition in Theorem 5, $\mathbf{g} \xrightarrow{G_1} \mathbf{f}$ is determined to be an anchor for \mathcal{G} . Next, consider the strongly-connected $\{f \xrightarrow{G_2} f\}$. The critical parameter sets here are $\{\mathbf{d}\}$ and $\{\mathbf{s}\}$. The second critical set allows us to conclude that $f \xrightarrow{G_2} f$ is an anchor. It follows that \mathcal{G} satisfies SCT.

Domains	$u, v, v_i \in Val$ $w, w_i \in Val^\# = Val \cup \{\perp, Err\}$, where $\perp \sqsubseteq w$ for all w .
Types	$\mathcal{E} : Expr \rightarrow Val^* \rightarrow Val^\#$ $\mathcal{O} : Op \rightarrow Val^* \rightarrow Val^\#$ $lift : Val \rightarrow Val^\#$ (the natural injection) $strictapp : (Val^* \rightarrow Val^\#) \rightarrow (Val^\#)^* \rightarrow Val^\#$
Semantic operator	$\mathcal{E}[\![f^{(i)}]\!](v_1, \dots, v_n) = lift\ v_i$ $\mathcal{E}[\![\text{if } e_1 \text{ then } e_2 \text{ else } e_3]\!]\mathbf{v} = \mathcal{E}[\![e_1]\!]\mathbf{v} \rightarrow \mathcal{E}[\![e_2]\!]\mathbf{v}, \mathcal{E}[\![e_3]\!]\mathbf{v}$ $\mathcal{E}[\![op(e_1, \dots, e_n)]\!]\mathbf{v} = strictapp\ (\mathcal{O}[\![op]\!])\ (\mathcal{E}[\![e_1]\!]\mathbf{v}, \dots, \mathcal{E}[\![e_n]\!]\mathbf{v})$ $\mathcal{E}[\![f(e_1, \dots, e_n)]\!]\mathbf{v} = strictapp\ (\mathcal{E}[\![e^f]\!])\ (\mathcal{E}[\![e_1]\!]\mathbf{v}, \dots, \mathcal{E}[\![e_n]\!]\mathbf{v})$
Auxiliary	$w_1 \rightarrow w_2, w_3 =$ $\begin{cases} w_1, & \text{if } w_1 \in \{\perp, Err\}, \\ w_2, & \text{if } w_1 = lift\ u \text{ where } u = True, \\ w_3, & \text{if } w_1 = lift\ u \text{ where } u \neq True. \end{cases}$ $strictapp\ \psi(w_1, \dots, w_n) =$ $\begin{cases} \psi(v_1, \dots, v_n) & \text{if each } w_i = lift\ v_i; \text{ else} \\ w_i & \text{where } i = \text{least index such that } w_i \in \{\perp, Err\} \end{cases}$
Assume	$\mathcal{O}[\![op]\!]\mathbf{v} \neq \perp$

Fig. A.1. Program semantics. *True* is a distinguished element of *Val*

4.2 Assessment of the SCT Approximations

Lemma 6. ([11]) *Let \mathcal{H} be non-empty, fan-in free and strongly-connected. For $x, y \in \overrightarrow{TP}(\mathcal{H})$, if there exists $f \xrightarrow{G} g$ in \mathcal{H} such that $x \xrightarrow{r} y \in G$ for some r , there is a critical set P such that $x, y \in P$ and $\overrightarrow{TP}_P(\mathcal{H}) = P$.*

Ref	program	·	size	<i>SCT</i> (ms)	<i>SCP1</i> (ms)	<i>SCP2</i> (ms)	procedure(s)
[2]	read	ts	56 N	1103.53	-	- N	4.68 read_expt1 ...
[2]	qplan	ts	28 N	25.10	-	- N	2.35 schedule_plan ...
[2]	tictactoe	ts	27 N	12.37	-	- N	1.99 alpha_beta ...
[13]	vangelder	ts	24 Y	59.66	-	- N	2.37 q t r p
[2]	rdtok	ts	23 N	68.94	N	1.22 N	2.49 read_tokens ...
[2]	warplan	ts	22 N	13.25	-	- N	1.62 plan_achieve ...
[13]	sicstus3	ts	15 N	8.89	N	0.51 N	1.20 put_assoc
[2]	rdtok	ts	15 N	7.49	-	- N	1.17 read_string ...
[2]	aiakl	ts	15 Y	7.47	Y	0.04 Y	0.07 intersect
[2]	qplan	ts	14 Y	1.96	Y	0.17 Y	0.36 incorporate ...
[2]	serialize	ts	12 Y	1.66	Y	0.02 Y	0.05 split0
[2]	ann	ts	12 Y	9.67	-	- Y	0.08 collect_info
[13]	sicstus3	ts	11 N	7.00	N	0.45 N	0.95 get_assoc
[2]	browse	ts	11 N	20.19	N	0.23 N	1.10 get_pats
[2]	browse	ts	10 N	20.94	N	0.34 N	0.83 init
[2]	ann	ts	9 Y	2.57	Y	0.03 Y	0.09 intersect
[2]	ann	ts	9 Y	2.51	Y	0.03 Y	0.08 merge
[13]	yale_s_p	ll	8 Y	1.40	-	- Y	0.23 ab_holds
[2]	warplan	ts	8 Y	0.96	Y	0.03 Y	0.05 retract1
[2]	qplan	ts	8 Y	0.52	-	- Y	0.08 best_goal
[2]	hanoiapp	ts	8 Y	19.47	Y	0.03 Y	0.07 shanoi
[2]	ann	ts	8 N	2.49	N	0.39 N	0.88 un_number_vars_2
[13]	arit_exp	ts	7 Y	18.52	Y	0.14 Y	0.43 e g f
[14]	pl8.4.2	ts	7 Y	6.38	Y	0.17 Y	0.50 e n t
[2]	tak	ts	7 N	4.00	N	0.38 N	0.78 tak
[2]	rdtok	ts	7 N	7.93	N	0.29 N	0.70 read_digits
[2]	qplan	ts	7 N	5.86	N	0.38 N	0.88 variablise
[2]	qplan	ts	7 N	3.85	N	0.38 N	0.82 quantificate
[2]	qplan	ts	7 N	5.25	-	- N	0.78 cost
[2]	boyer	ts	7 N	1.72	N	0.38 N	0.81 tautology
[13]	ack	ts	6 Y	1.99	Y	0.02 Y	0.06 ack
[14]	mergesort_t	ll	6 Y	0.56	Y	0.09 Y	0.23 split2 split
[2]	warplan	ts	6 N	1.93	N	0.28 N	0.62 mkground ...
[2]	tictactoe	ts	6 N	0.60	N	0.16 N	0.74 choose_legal
[2]	qplan	ts	6 Y	2.74	Y	0.03 Y	0.08 mark
[2]	qplan	ts	6 N	5.92	N	0.37 N	0.85 variables
[2]	qplan	ts	6 Y	0.64	Y	0.04 Y	0.10 instantiate ...
[2]	peephole	ll	6 Y	2.56	Y	0.04 Y	0.20 popt3 popt31
[2]	peephole	ll	6 Y	1.01	Y	0.09 Y	0.21 popt1a popt1a1
[2]	peephole	ll	6 Y	1.01	Y	0.04 Y	0.09 popt2 popt21
[2]	hanoiapp	ts	6 N	16.62	N	0.20 N	0.73 shanoi
[2]	fib_t	ts	6 Y	1.70	Y	0.03 Y	0.07 add
[2]	ann	ts	6 N	5.99	N	0.39 N	0.87 numbervars_2
[2]	ann	ts	6 Y	0.94	Y	0.03 Y	0.09 subtract

Fig. B.1. Experiments conducted on a P3/800 running Redhat Linux 6.1

Lemma 7. *Let SCT , $SCP1$ and $SCP2$ denote those \mathcal{G} satisfying SCT , $SCP1$ and $SCP2$ respectively. Then $SCP1 \subset SCP2 \subset SCT$.*

Proof. It follows from Lemma 6 that any anchor deduced for some strongly-connected, fan-in free \mathcal{H} by the condition of Theorem 4 is also deduced to be an anchor by the first condition of Theorem 5. Therefore, $SCP1 \subseteq SCP2$. The other containment $SCP2 \subseteq SCT$ is clear. The containments are strict: $SCP1$ does not handle the natural graphs for Ex. 4c of Sect. 1.2, whereas $SCP2$ does; $SCP2$ does not handle the natural graphs for Ex. 4d, whereas SCT does. \square

Experimental results. Finally, we report some empirical findings. The size-change graphs used for our experiments have been generated using Terminweb’s size analysis [3], applied to the test suites in [14,1,2,4,13]. For each program, a set of size-change graphs \mathcal{G} is compiled using either the *list-length norm* (ll) or the *term-size norm* (ts). Instead of timing the analysis of each \mathcal{G} , we consider their SCCs, since our analyses scale linearly in this number. For each SCC, we time 1000 runs each of the SCT , $SCP1$ and $SCP2$ tests. The table in Fig. 4.1 shows the benchmark reference, program name, size norm used, total number of arcs in the tested component’s graphs (the problem size), results and timings of the various tests, and names of some procedures in the component.

A total of 103 programs with 281 components have been analyzed. For our experiments, the $SCP2$ test has given the same results as SCT in all but one instance, a contrived example due to Van Gelder. Apart from 19 components where the $SCP1$ test does not apply due to fan-ins (this situation is indicated by - in the table), it has also given the same results, suggesting that SCT is “overkill” in practice. Timing-wise, $SCP1$ is consistently and significantly faster than SCT , when it is applicable. For 19 small components, $SCP2$ is actually slower than SCT (probably due to its more difficult implementation), but it gives up quickly on complicated examples that eventually cause SCT to fail. For our table, we have simply ordered the experiments according to problem size, and listed the results at the top.

5 Concluding Remarks

We have reviewed the size-change condition for program termination. In a previous article, we established that SCT is complete for PSPACE. In this article, we have explored, theoretically and empirically, polynomial-time approximations of SCT . In particular, we have proposed two tests that may establish SCT . The first is simpler but restricted to fan-in free graphs. It has worst-case quadratic-time complexity and is very fast in practice. The general test has worst-case cubic-time complexity and accurately predicts SCT in practice.

There remain a number of obstacles to the analysis of “real programs.”

- **For functional programs:** We require a convincing extension of the basic principles to higher-order programs.

- **For Prolog programs:** Fast supporting analyses are needed. They have to be combined with SCP2 sensibly to produce a useful tool for Prolog.
- **For imperative programs:** Research is on-going. It is hard to deal with imperative programs, as they do not use well-founded data. Further, sequential programming encourages nested loops that maintain complex invariants among variable values. This makes the job of size analysis difficult.

References

1. K. R. Apt and D. Pedreschi. Modular termination proofs for logic and pure prolog programs. In *Advances in Logic Programming Theory*, pages 183–229. Oxford University Press, 1994.
2. F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of global analysis in strict independence-based automatic program parallelization. In *International Symposium on Logic Programming*, pages 320–336. MIT Press, 1994.
3. Michael Codish and Cohavit Taboch. A semantic basis for termination analysis of logic programs and its realization using symbolic norm constraints. In Michael Hanus, Jan Heering, and Karl Meinke, editors, *Algebraic and Logic Programming, 6th International Joint Conference, ALP '97-HOA '97, Southampton, U.K., Sep 3–5, 1997*, volume 1298 of *LNCS*, pages 31–45. Springer, 1997.
4. D. De Schreye and S. Decorte. Termination of logic programs: The never-ending story. *The Journal of Logic Programming*, 19–20:199–260, 1994.
5. Carl C. Frederiksen. SCT analyzer. At <http://www.diku.dk/~xeno/sct.html>.
6. Carl C. Frederiksen. A simple implementation of the size-change principle. Technical Report D-442, Datalogisk Institut Københavns Universitet, 2001.
7. Neil Jones and Arne Glenstrup. Program generation, termination, and binding-time analysis. In *Principles, Logics, and Implementations of High-Level Programming Languages, PLI 2002 (invited paper)*. Springer, 2002.
8. Chin Soon Lee. Partial evaluation of the euclidean algorithm, revisited. *Higher-Order and Symbolic Computation*, 12(2):203–212, Sep 1999.
9. Chin Soon Lee. *Program Termination Analysis, and Termination of Offline Partial Evaluation*. PhD thesis, UWA, University of Western Australia, Australia, 2001.
10. Chin Soon Lee. Finiteness analysis in polynomial time. In M. Hermenegildo and G. Puebla, editors, *Proceedings of Static Analysis Symposium*, volume 2477 of *LNCS*. Springer, 2002.
11. Chin Soon Lee. Program termination analysis in polynomial time (with proofs). To be available as a technical report at DIKU, Denmark, 2002.
12. Chin Soon Lee, Neil D. Jones, and Amir Ben-Amram. The size-change principle for program termination. In *Proceedings of the ACM Symposium on Principles of Programming Languages, Jan 2001*. ACM, 2001.
13. Naomi Lindenstrauss and Yehohua Sagiv. Automatic termination analysis of logic program (with detailed experimental results). Article available at <http://www.cs.huji.ac.il/~naomil/>, 1997.
14. Lutz Plümer. *Termination Proofs for Logic Programs*, volume 446 of *LNAI*. Springer-Verlag, 1990.
15. Wim Vanhoof and Maurice Bruynooghe. Binding-time annotations without binding-time analysis. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR), 8th International Conference, Havana, Cuba, Dec 3-7, 2001*, volume 2250 of *LNCS*, pages 707–722, 2001.

A Program Semantics

Programs are interpreted according to the semantic operator \mathcal{E} of Fig. A.1.

B Results of Experiments

As instantiatedness information is not accounted for, it is *ground termination behaviour* that is analyzed. A positive SCT result for a component does not indicate the termination of procedures in the component. It indicates that no looping occurs among these procedures. The ground termination of particular procedures can be recovered from this information and the call graph. SCT termination deductions are comparable with Termilog and Terminweb's.

Generators for Synthesis of QoS Adaptation in Distributed Real-Time Embedded Systems

Sandeep Neema¹, Ted Bapty¹, Jeff Gray², and Aniruddha Gokhale¹

¹ Institute for Software Integrated Systems, Vanderbilt University, Nashville TN 37211
{sandeep, bapty, gokhale}@isis-server.vuse.vanderbilt.edu
<http://www.isis.vanderbilt.edu>

² Department of Computer and Information Sciences, University of Alabama at Birmingham
1300 University Boulevard, Birmingham AL 35294-1170
gray@cis.uab.edu
<http://www.gray-area.org>

Abstract. This paper presents a model-driven approach for generating Quality-of-Service (QoS) adaptation in Distributed Real-Time Embedded (DRE) systems. The approach involves the creation of high-level graphical models representing the QoS adaptation policies. The models are constructed using a domain-specific modeling language - the Adaptive Quality Modeling Language. Multiple generators have been developed using the Model-Integrated Computing framework to create low-level artifacts for simulation and implementation of the adaptation policies that are captured in the models. A simulation generator tool synthesizes artifacts for Matlab Simulink/Stateflow® (a popular commercial tool), providing the ability to simulate and analyze the QoS adaptation policy. An implementation generator creates artifacts for Quality Objects, a QoS adaptation software infrastructure developed at BBN, for execution of QoS adaptation in DRE systems. A case study in applying this approach to an Unmanned Aerial Vehicle – Video Streaming application is presented. This approach has goals that are similar to those specified in the OMG's Model-Driven Architecture initiative.

1 Introduction

Rapid advances in middleware technologies have given rise to a new generation of highly complex, object-oriented Distributed Real-Time Embedded (DRE) systems based on technologies such as RT-CORBA, COM+, RMI, among others. As observed in [9], middleware solutions promote software reuse resulting in better productivity. However, within the context of DRE systems, specifying and satisfying quality-of-service (QoS) requirements demand ad-hoc, problem-specific code optimizations. Such modifications go against the very principles of reuse and portability.

There have been some attempts to improve this situation by introducing a programmable QoS adaptation layer on top of the middleware infrastructure. The key idea behind the adaptation layer is to offer *separation of concerns* with respect to QoS requirements. Such separation provides improved modularization for separating QoS

requirements from the functional parts of the software. Quality Objects (QuO), developed by BBN, is one such adaptation layer [6]. QuO provides an extension to the Object Management Group's (OMG) Interface Definition Language (IDL). This extension, known as the Contract Definition Language (CDL), supports the specification of QoS requirements and adaptation policies. Contracts written in CDL are compiled to create stubs that monitor and adapt the QoS parameters when the system is operational. These stubs are integrated into the QuO kernel.

From a software engineering standpoint this approach works well; however, there are a few drawbacks to using CDL alone:

1. The control-centric nature of the QoS adaptation extends beyond the software. QoS parameters in a DRE system have a significant impact on the dynamics of the overall physical system. Owing to the complex and non-linear dynamics, it is very difficult to tune the QoS parameters in an ad-hoc manner without compromising the stability of the underlying physical system. The QoS adaptation software is, in effect, equivalent to a controller for a discrete, non-linear system. Therefore, sophisticated tools are needed to design, simulate, and analyze the QoS adaptation software from a controls perspective.
2. For scalability reasons, the CDL offers a much lower level of abstraction (textual code-based). Even for a moderately large system, the CDL specifications can grow quite large. Instrumenting a small change to the adaptation policy requires making several changes manually, while ensuring that all these changes are consistent in the CDL specification.

In this paper, we present an approach based on the principles of Model-Integrated Computing (MIC) [5], which addresses these issues. Our approach involves creating a graphical modeling environment that allows a DRE system designer to graphically model the DRE system and the QoS adaptation policy using standardized notations, such as Statecharts [3] and Dataflow. A generator tool synthesizes artifacts for Matlab Simulink/Stateflow® (a popular commercial tool) providing the ability to simulate and analyze the QoS adaptation policy. This gives significant assurance that the system will perform as desired. A second generator tool creates CDL specifications from the QoS adaptation models. The generated CDL is then compiled into executable artifacts. The approach described in this paper has goals that are similar to those specified in the OMG's Model-Driven Architecture (MDA) [1, 2].

The rest of this paper is organized as follows. Section 2 presents the modeling environment and the modeling concepts required by our approach. Section 3 introduces the generator that creates simulation artifacts from the models. Section 4 describes another generator that creates CDL specifications from the models. A short case study in applying the approach to a video streaming application, within the context of an Unmanned Aerial Vehicle (UAV), is presented in section 5. The paper ends with some concluding remarks in section 6.

2 Modeling Paradigm

The MIC infrastructure provides a unified software architecture and framework for creating a Model-Integrated Program Synthesis (MIPS) environment [5]. The core components of the MIC infrastructure are: a customizable *Generic Model Editor* for

creation of multiple-view, domain-specific models; *Model Databases* for storage of the created models; and, a *Model Interpretation* technology that assists in the creation of domain-specific, application-specific model interpreters for transformation of models into executable/analyzable artifacts. The new environment is domain-specific and includes tools and functionality to support the creation and storage of system models, in addition to generation of executable/analyzable artifacts from these models.

In the MIC technology, the modeling concepts to be instantiated in the MIPS environment are specified in a *meta-modeling* language [8]. A *metamodel* of the modeling paradigm is constructed that specifies the syntax, static semantics, and the presentation semantics of the domain-specific modeling paradigm. The metamodel uses a Unified Modeling Language (UML) class diagram to capture information about the objects that are needed to represent the system information and the inter-relationships between different objects. The meta-modeling language also provides for the specification of visual presentation of the objects in the graphical model editor [8].

The Adaptive Quality Modeling Language (AQML) presented here models the following key aspects of a DRE system:

1. *QoS Adaptation Modeling* – In this first category, the adaptation of QoS properties of the DRE system is modeled. The designer can specify the different state configurations of the QoS properties, the legal transitions between the different state configurations, the conditions that enable these transitions (and the actions that must be performed to enact the change in state configuration), the data variables that receive and update QoS information, and the events that trigger the transitions. These properties are modeled using an extended finite-state machine (FSM) modeling formalism [3].
2. *Computation Modeling* – In this category, the computational aspect of a DRE system is modeled. A dataflow model is created in order to specify the various computational components and their interaction. An extended dataflow modeling formalism is employed.
3. *Middleware Modeling* – In this category, the middleware services, the system monitors, and the tunable “knobs” (i.e., the parameters being provided by the middleware) are modeled.

The metamodel of each of these modeling categories and their interaction in the AQML is described below.

2.1 QoS Adaptation Modeling

Stateflow models capture the QoS adaptive behavior of the system. A *Discrete Finite State Machine* representation, extended with hierarchy and concurrency, is selected for modeling the QoS adaptive behavior of the system. This representation has been selected due to its scalability, universal acceptability, and ease-of-use in modeling. Figure 1 illustrates the QoS adaptation aspect of the AQML.

The main concept in a Finite State Machine (FSM) representation is a *state*. States define a discretized configuration of QoS properties. Hierarchy is enabled in the representation by allowing States to contain other States. Attributes define the decomposition of the State. The State may be an *AND* state (when the state machine contained

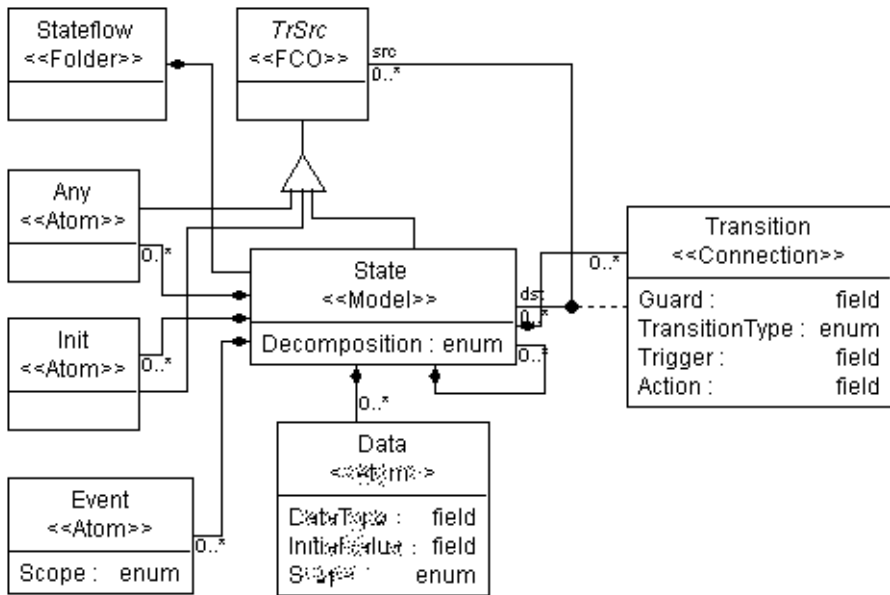


Fig. 1. Metamodel of QoS Adaptation Modeling

within the State is a concurrent state machine), or, the State can be an *OR* state (when the state machine contained within the State is a sequential state machine). If the State does not contain child States, then it is specified as a *LEAF* state. States are stereotyped as *models* in the MIC framework.

Transition objects are used to model a transition from one state to another. The attributes of the transition object define the trigger, the guard condition, and the actions. The trigger and guard are Boolean expressions. When these Boolean expressions are satisfied, the transition is enabled and a state change (accompanied with the execution of the actions) takes place. Transitions are stereotyped as a *connection* in the MIC framework. To denote a transition between two States, a connection has to be made from the source state to the destination state.

In addition to states and transitions, the FSM representation includes data and events. These can be directly sampled external signals, complex computational results, or outputs from the state machine. In the modeling paradigm, *Event* objects capture the Boolean event variables, and the *Data* objects capture arbitrary data variables. Both the Events and Data have a *Scope* attribute that indicates whether an event (or data) is either local to the state machine or is an input/output of the state machine.

2.2 Computation Modeling

This modeling category is used to describe the computational architecture. A dataflow representation, with extensions for hierarchy, has been selected for modeling computations. This representation describes computations in terms of computational compo-

nents and their data interactions. To manage system complexity, the concept of hierarchy is used to structure the computation definition. Figure 2 illustrates the computation aspect of the AQML. The different objects and their inter-relationships are described below.

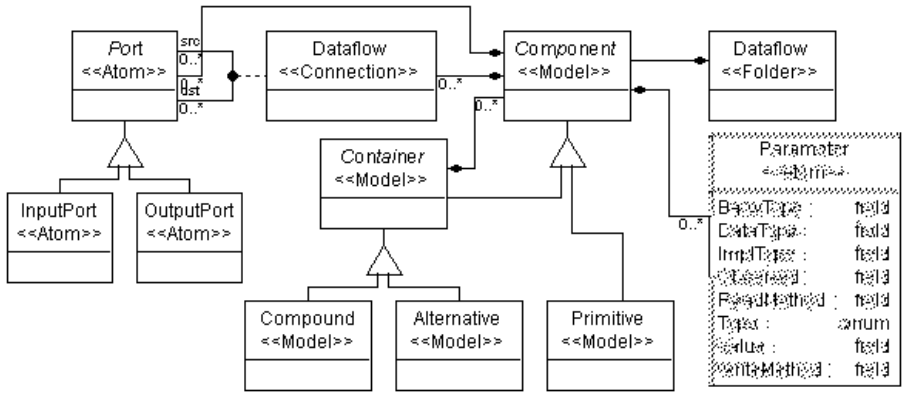


Fig. 2. Metamodel of Computation Architecture Modeling

The computational structure is modeled with the following classes of objects: *Compounds*, *Alternatives*, and *Primitives*. These objects represent a computational component in a dataflow representation. *Ports* are used to define the interface of these components through which the components exchange information. Ports are specialized into *InputPorts* and *OutputPorts*.

A *Primitive* is a basic modeling element that represents an elementary component. A *Primitive* maps directly to a processing component that will be implemented as a software object or a function. A *Compound* is a composite object that may contain *Primitives* or other *Compounds*. These objects can be connected within the compound to define the dataflow structure. *Compounds* provide the hierarchy in the structural description that is necessary for managing the complexity of large designs. An *Alternative* captures “design choices” – functionally equivalent designs for a rigorously defined interface, providing the ability to model design spaces instead of a single design.

An important concept relevant to QoS adaptive DRE systems is the notion of parameters. Parameters are the tunable “knobs” that are used by the adaptation mechanism to tailor the behavior of the components such that desired QoS properties are maintained. Parameters can be contained in *Compounds* and *Primitives*. The *Type* attribute defines whether a *Parameter* is read-only, write-only, or read-write. The *DataType* attribute defines the data type of the parameter.

2.3 Middleware Modeling

In this category, the components of the middleware are modeled. These components include the services and the system conditions provided by the middleware. Examples of services include an Audio-Video Streaming service, Bandwidth reservation service,

Timing service, and Event service, among others. System conditions are components that provide quantitative diagnostic information about the middleware. Examples of these include observed throughput, bandwidth, latencies, and frame-rates. Figure 3 illustrates the middleware modeling aspect of the AQML.

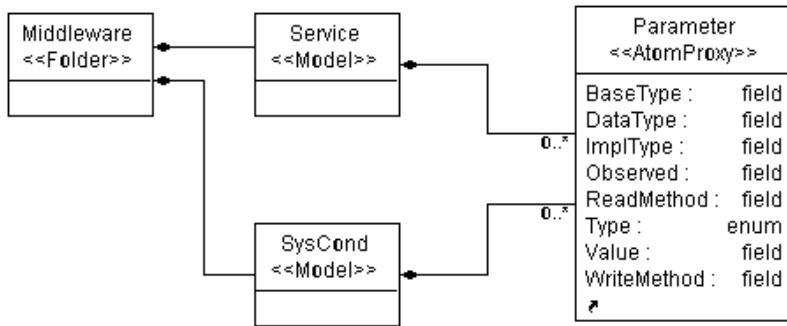


Fig. 3. Metamodel of Middleware Modeling

The *Service* object represents the services provided by the middleware. Services can contain parameters that are the tunable knobs provided by the service. In addition to being tunable “knobs,” parameters play a second role as instrumentation, or probes, by providing some quantitative information about the service.

The *SysCond* object represents the system condition objects present in the middleware layer. SysConds can also contain parameters.

Observe that we do not facilitate a detailed modeling of the middleware components or the dataflow components. This is because the focus of AQML is on the QoS adaptation. We model only those elements of the dataflow and middleware that facilitate the QoS adaptation (namely, the tunable and observable Parameters).

2.4 Interaction of QoS Adaptation with Middleware and Computation Modeling

In this category, the interaction of the previous three modeling categories (illustrated in Figure 4) is specified. As described earlier, the Data/Event objects within the Stateflow model form the interface of the state machine. Within the Computation and the Middleware models, Parameters form the control interfaces. The interaction of the QoS adaptation (captured in Stateflow models), and the middleware and application (modeled in the Middleware/Computation models), is through these interfaces. The interaction is modeled with the *Control* connection class, which connects the Data object of a State model to a Parameter object of a Middleware/Computation model.

In the MIC framework, connections between objects that are not contained within the same model are not supported. Therefore, we create references, which are equivalent to a “reference” or a “pointer” in a programming language. These are contained in the same context (model) such that a connection can be drawn between

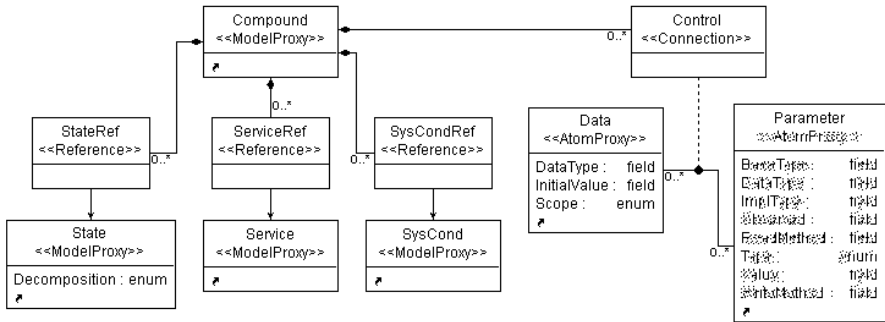


Fig. 4. Metamodel of Interaction of QoS Adaptation with Middleware/Computation Modeling

them. The *StateRef*, *ServiceRef*, and the *SysCondRef* objects are the reference objects that are contained in a Compound (Computation) Model.

3 Simulation Generator

One of the primary goals of our approach, as identified earlier, is to be able to provide integration with tools that can analyze the QoS adaptation from a control-centric viewpoint. Matlab Simulink/Stateflow® is an extremely popular commercial tool that is routinely used by control engineers to design and simulate discrete controllers. Simulink provides the ability to model hybrid systems (mixed continuous and discrete dynamics) in a block-diagrammatic notation. Stateflow provides the ability to model hierarchical parallel state machines in a Statechart like notation [3]. A Stateflow model can be inserted in a Simulink model as a block and the Simulink blocks can provide input stimulus and receive outputs from the Stateflow block. The Simulink/Stateflow model can be simulated within the Matlab framework for a desired time period, which in effect steps through the state-machine for the given input excitation. The responses from the state machine can be graphically plotted and the trajectory of the state machine can be visually observed, as well as recorded, for post-analysis. Additional analyses are possible in terms of the time spent in different states, the latency from the time of a change in excitation, to the time of change in outputs in the state machine, stability of the system, etc. Thus, the Matlab Simulink/Stateflow tool suite provides an extremely convenient and intuitive framework for observing and verifying the behavior of the system.

We have implemented a generator, using our model interpretation technology, which can translate the QoS adaptation specifications (modeled using the AQML) into a Simulink/Stateflow model. Matlab provides an API that is available in the Matlab scripting language (M-file), for procedurally creating and manipulating Simulink/Stateflow models. The simulation generator produces an M-file that uses the API to create Simulink/Stateflow models.

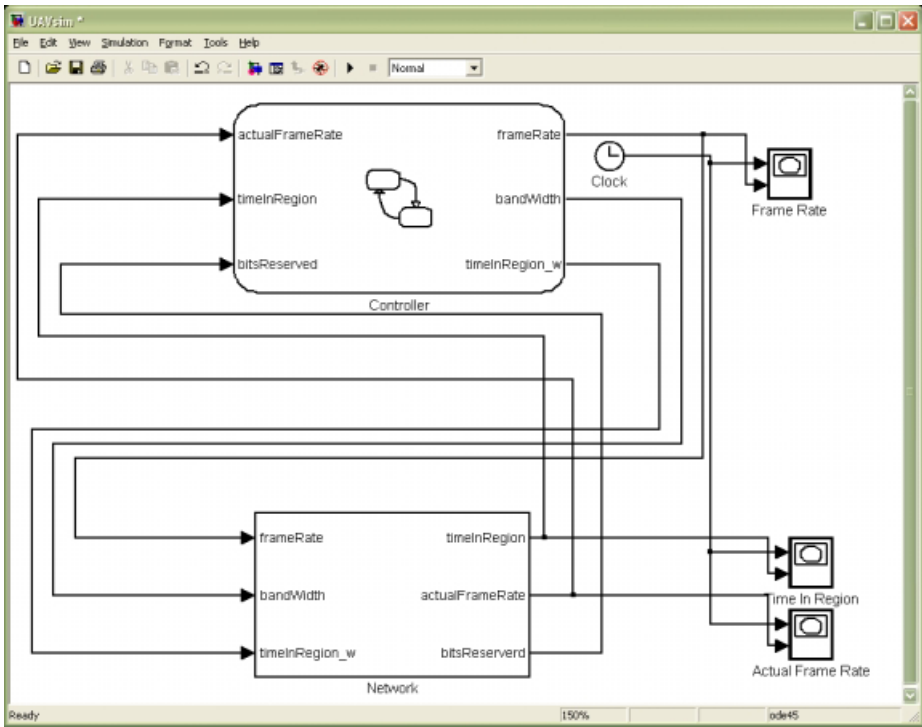


Fig. 5. Top-level Structure of the Generated Simulink/Stateflow Model

The structure of the generated Simulink/Stateflow model is shown in Figure 5. There are two main blocks in the generated Simulink model. A Stateflow block is generated that represents the QoS adaptation engine (labeled Controller in Figure 5). A Subsystem block is generated that represents the simulated middleware and the application (labeled Network in Figure 5). Only the interface of this block is generated. The user models the dynamics of the physical network, the middleware, and the application composite, as mathematical equations created using Simulink blocks within this Subsystem. The fidelity of this model is dependent upon the user. A variety of models may be created, ranging from coarse and simple, to highly fine-grained and extremely complex. It must be noted here that these models are dependent on the underlying physical network, and not dependent on the controller itself. Thus, for a new network, a user may have to create a new model; however, while simulating and verifying the modeled controller, there is no need to recreate the network simulation model every time there is a change in the controller.

The main work of the simulation generation algorithm is in synthesizing the Stateflow block. The Stateflow representation within Matlab has almost a one-to-one equivalence with the Stateflow representation within AQML. However, the representation of hierarchy is somewhat complicated in Stateflow. Although the AQML hierarchy is established by an actual containment relationship over objects, the Matlab hierarchy is enforced strictly through graphical containment. Thus, all the rectangles

representing child states are graphically bounded by the rectangle symbolizing the parent state. Any overlap is flagged as a syntax error by the Stateflow simulator.

The Stateflow block generation is essentially a two-pass interpretation. The first-pass calculates the bounding rectangles for each state according to the containment relationships. The second-pass emits the API calls in the M-file to create states with appropriate bounding rectangles. The second-pass also emits API calls for creating data variables, event variables, and transitions. An example of the generated M-file and the Simulink/Stateflow diagram is shown in the case study.

4 CDL Generator

Research at BBN on DRE systems and QoS adaptation resulted in the CDL – a domain-specific language (based on OMG IDL) for specifying QoS contracts. This research also produced a CDL compiler and a QoS adaptation kernel that can process specifications (contracts) expressed in CDL. The CDL compiler translates QoS contracts into artifacts that can execute the adaptation specifications at run-time. CDL is a textual language, and it has a state-machine like flavor (see [6] for details). Our research efforts build upon their work, and we utilize their infrastructure to affect the adaptation specifications captured in the AQML models. That is, we generate CDL specifications from the AQML models and use the BBN QuO tools to actually instantiate these adaptation instructions at runtime.

Although the CDL has a state-machine like flavor and can implement hierarchical finite state machines, it does not support the notion of parallelism in the state machine description. To compensate, the AQML CDL generator procedurally explores the state-space captured by the hierarchical, parallel, FSMs in the AQML, and translates it into flat FSM. A simple algorithm to procedurally explore the state-space captured by a hierarchical, parallel, FSM could be exponential in complexity. Instead, we use an approach based on symbolic methods to explore the state-space. We use a variant of a prefix-based encoding scheme to assign an encoding (a string of Boolean values) to each state in the state machine (see [7] for details). Given this encoding, each state can be represented as a Boolean formula, and the complete state-space can be composed symbolically using these Boolean formulae. The satisfying valuations of the Boolean formula represent the entire state-space corresponding to a state in the flattened state machine representation. Transitions in the flat FSM representation can be resolved procedurally by determining the constituent components of a satisfying valuation from the encoding.

A second challenge in translating the AQML models to CDL is in the synthesis of the transition triggers, guards, and actions. In AQML, the triggers, guards, and actions, are written using a standard expression language (from Statecharts). Further, the expressions involve the data and event variables declared within the state machine description. In the CDL, however, the expressions are written in an IDL/C++ flavor. The expressions involve method calls over the SysCond objects, which are passed as arguments to the contract. Therefore, the CDL generator parses the trigger, guard, and action expressions. The abstract syntax tree created by the parser is traversed by the generator. During traversal, when data or event variables are referenced in the expression, the generator determines the middleware or application parameter that the data variable is connected to and emits the equivalent of a *get* or *set* method call on the pa-

parameter object (depending on the context of the reference). The exact name of the method is available to the generator via the `ReadMethod` and `WriteMethod` attributes of the `Parameter` object in the `Middleware` and `Computation Model` (see Figure 2).

The CDL generator performs a multi-pass interpretation. The first pass parses the trigger, guard, and action expressions of the transitions, and builds individual abstract syntax trees. A second pass creates a flat FSM representation from the hierarchical, parallel, FSM representation. The final pass traverses the flat FSM representation, evaluates the abstract syntax tree created for trigger, guard, and action expressions of the transitions, and emits CDL specifications that represent an equivalent flat FSM representation. The CDL generation presented above is further illustrated with an example in the following section.

5 Case Study

The case study presented in this section is based upon a UAV video streaming application [4]. There are several things that make this a complex and challenging problem (i.e., the real-time requirements, resource constraints, and the distributed nature). Figure 6 shows a UAV video application scenario. This is a Navy application where a number of UAVs are simultaneously transmitting surveillance video from different regions in a battlefield. A distributor node on a shipboard receives these video streams and sends it to different receivers on the ship that are interested in monitoring those video streams and responsible for making tactical decisions about deployment and guidance. There are a few interesting observations to make: 1) the link between the UAV to the ship is a wireless link imposing some strict bandwidth restrictions; 2) there is a need for prioritization between different video streams coming from different UAVs owing to the region of interest, nature of threat, etc; 3) latency is a higher concern than throughput because it is important to get the latest changes in the threat scenario at the earliest possible time; 4) there may be a wide-variety of computational resources (processors, networks, switches) involved in the entire application; and 5) the scenario is highly dynamic (i.e., UAVs frequently enter and leave the battle field). Given these complex requirements, a QoS-enabled middleware solution has been proposed for this application [4, 6]. Due to the highly dynamic nature of the application scenario, the adaptation of the QoS properties is mandatory.

In this application, the goal of QoS adaptation is to minimize the latency on the video transmission. When the communication resources are nominally loaded, it may be possible to transmit the video stream at the full frame rate with a minimal basic network delay. However, when the communication and computational resources are loaded, the delays in transmission expand for the same frame rate resulting in increased latencies. The adaptation scheme attempts to compensate for the increased load by reducing the rate of transmission, thus improving the latency again. There are several ways of reducing the transmission rate: a) reduce the frame rate by dropping frames, b) reduce the image quality per frame, or c) reduce the frame size. Depending on the actual scenario, one or more of these situations may apply. In the example of this section, we consider dropping the frame rate only.

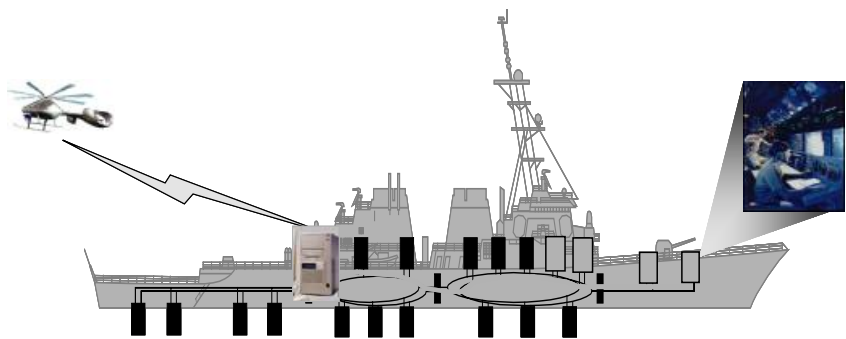


Fig. 6. UAV Video Streaming Application (reprinted from [4], with permission from BBN)

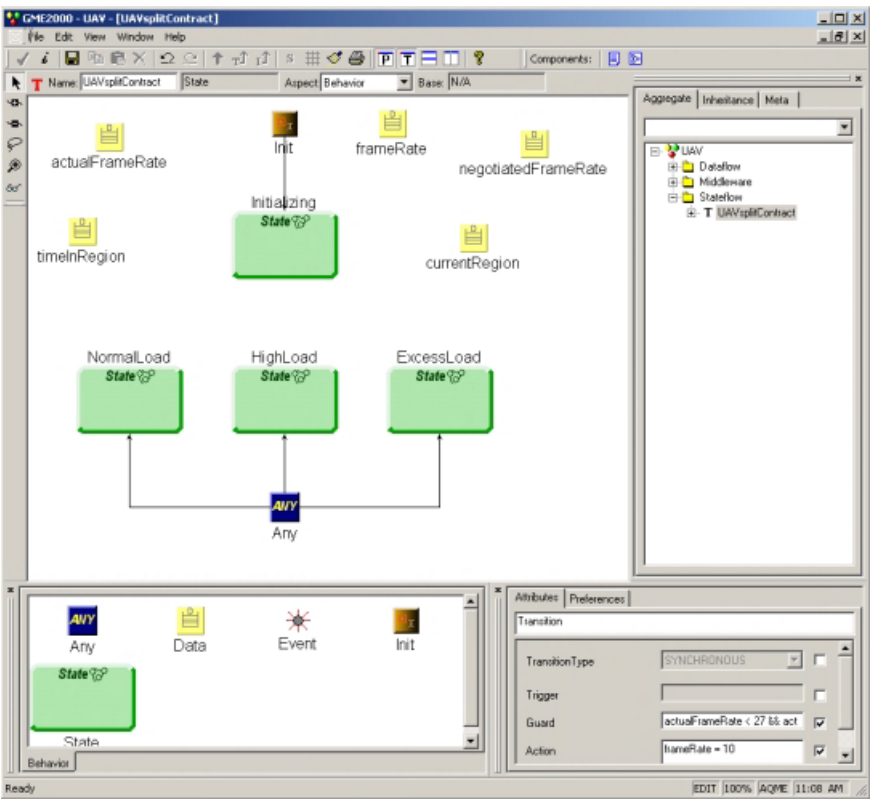


Fig. 7. Model of QoS Adaptation in AQML

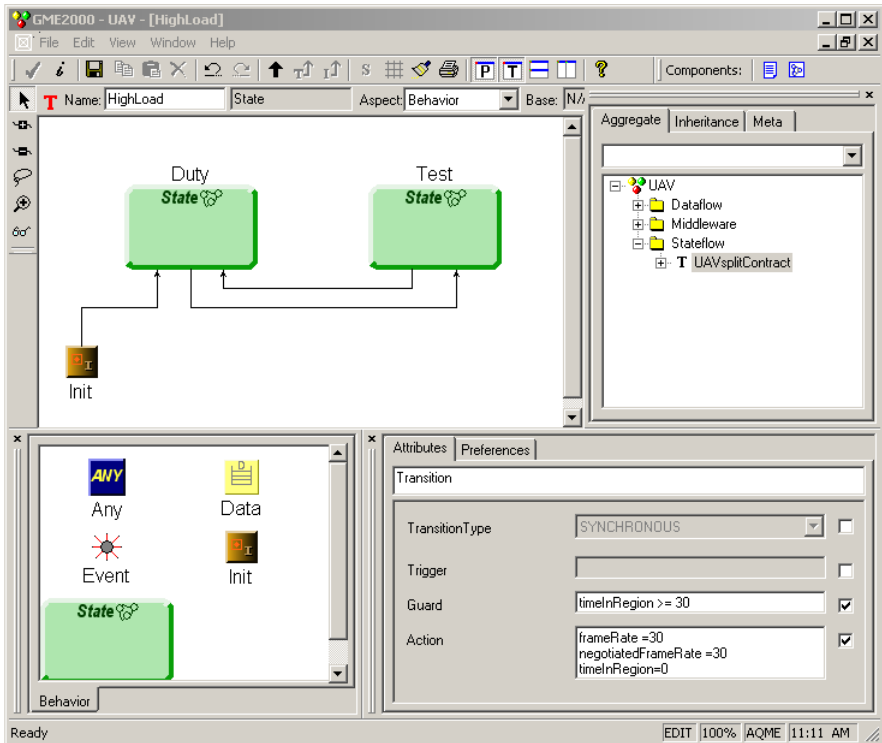


Fig. 8. The HighLoad State Model

Figure 7 shows a QoS adaptation model of the UAV scenario in the AQML. The three states NormalLoad, ExcessLoad, and HighLoad capture three different QoS configurations of the system. A few data variables (actualFrameRate, frameRate, timeInRegion) can also be seen in this figure. These data variables provide the state-machine with sensory information about the network. At the same time, some other data variables may enact the adaptation actions that are being performed in the transitions. Notice that the attribute window at the bottom-right corner of the figure shows the trigger, guard, and action attributes of a transition. An example guard expression is visible in the attribute window of the figure (i.e., “actualFrameRate < 27 and actualFrameRate >= 8”). When this expression evaluates to true, the transition is enabled and the DRE system enters the HighLoad state. An example action expression can be seen in this figure (i.e., “frameRate = 10”). This sets the frameRate data variable to a value of 10. Figure 8 shows an exploration into the hierarchy of the HighLoad state. A Duty and Test state are being depicted in this figure. The general idea is that when the system enters a HighLoad state, the frame rate is reduced. However, the adaptation periodically probes the network by entering a Test state, which bumps up the frame rate. If the transient load has disappeared, then the actualFrameRate variable goes up (indicating that the network can sustain the desired frame

rate). When this happens, the system switches into the NormalLoad state; otherwise, it goes back into the Duty state of the HighLoad state.

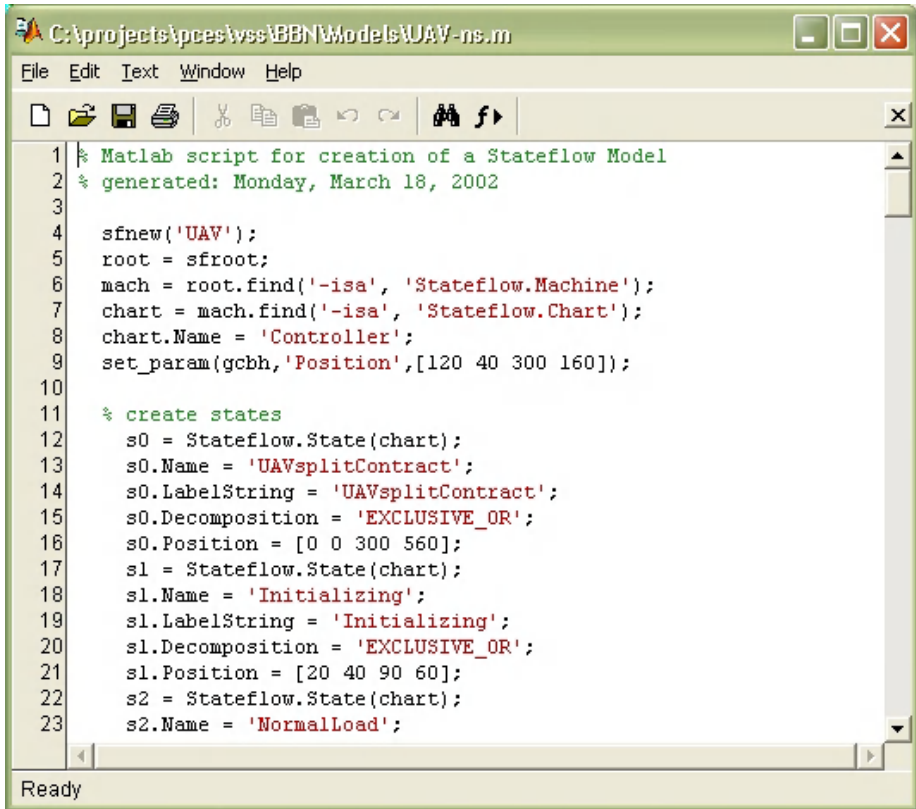


Fig. 9. Matlab M-file Generated from UAV Model

Figure 9 shows an M-file that was generated from the model in Figure 8. Figure 10 shows the generated Stateflow model, which in effect is equivalent to the state machine captured in the AQML models.

Figure 11 presents the results of the simulation (X-axis shows simulation time, and Y-axis shows the frame rate in frame-per-sec). The behavior of the adaptive system can be intuitively observed and understood from these plots. It can be seen that when the load crosses a threshold, the adaptation engine causes the frame rate to reduce adaptively (thus minimizing the latency). When the load vanishes, the frame rate is enhanced again. Some important information about the adaptation (in terms of lead times, response times, and stability) can be gathered from these plots. If the response is too quick, then the network may enter into oscillations. The network may also get into oscillatory behavior if the frame-rate reduction step is too high. On the other hand, if the frame-rate reduction step is too low, the duration for which the latency of the network stays high may be very large.

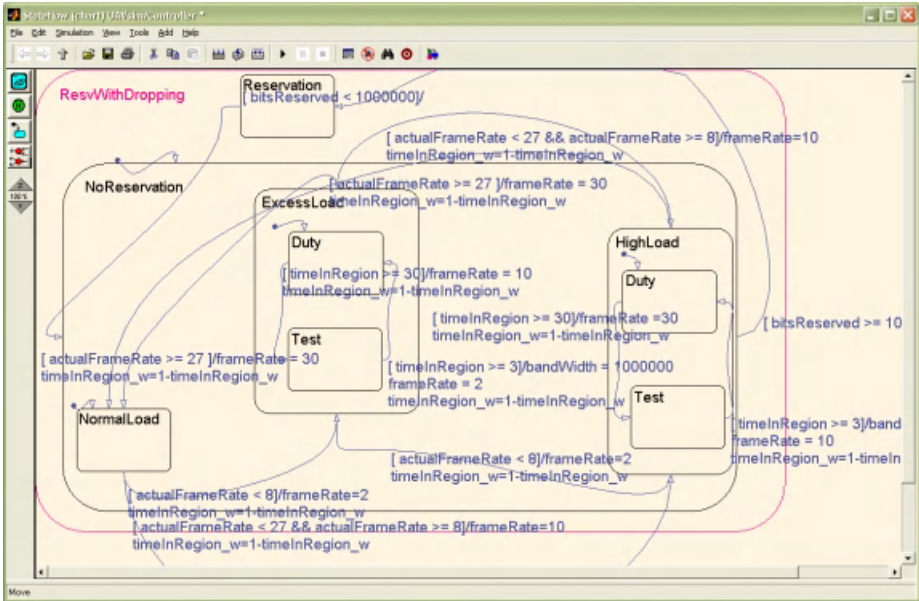


Fig. 10. Generated Matlab Stateflow Model

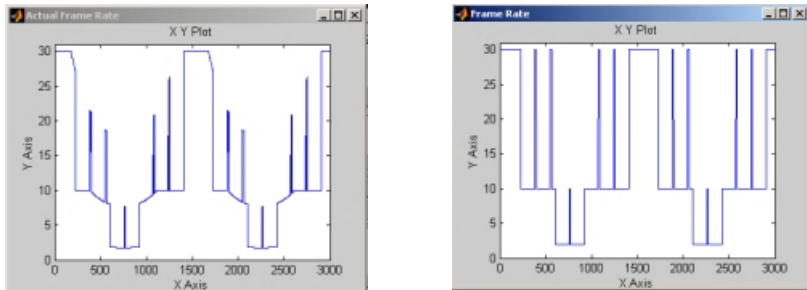


Fig. 11. Simulation Results – Actual (left) and Desired (right) Frame Rate

Figure 12 illustrates a CDL file that was generated from the same model. This CDL file is processed by the QuoGen compiler to generate runtime artifacts for the Quo kernel.

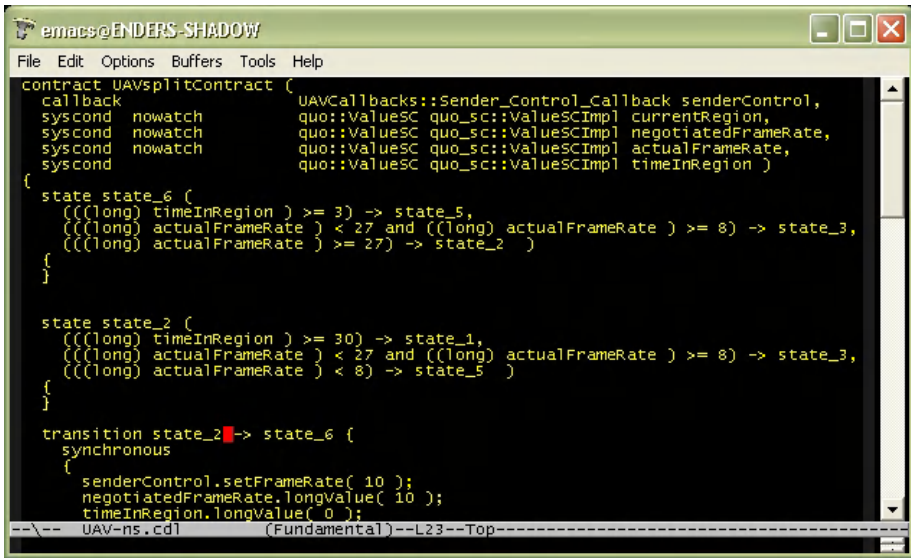


Fig. 12. CDL File Generated from Models

6 Conclusions

This paper presented an approach based on MIC for simulating and generating QoS adaptation software for DRE systems. The key focus of the approach is on raising the level of abstraction in representing QoS adaptation policies, and providing a control-centric design for the representation and analysis of the adaptation software. Using a model-based representation and employing generators to create low-level artifacts from the models, we have been successful in raising the level of abstraction, and providing better tool support for analyzing the adaptation software. At the same time, our approach results in increased productivity as we 1) shorten the design, implement, test, and iterate cycle by providing early simulation, and analysis capabilities, and 2) facilitate change maintenance as minimal changes in the models can make large and consistent changes in the low-level (CDL) specifications. This approach has similar goals to the OMG's MDA initiative [1, 2].

The approach has been tested and demonstrated on a UAV Video Streaming application described as a case study in this paper. The case study presented a simple scenario; however, we have been able to model a much larger scenario in the developed modeling environment, with a higher degree of variability and adaptability. In our experience, the simulation capabilities have been particularly helpful in fine-tuning the adaptation mechanism.

The tool is still in the prototype state and several enhancements are planned. We plan to integrate a symbolic model-checking tool for formal reasoning about the adaptation mechanism. With the aid of this tool we can establish various properties (such as liveness, safety, reachability, etc.), about the state-machine implementing the

adaptation policy. We also plan to strengthen the computation and middleware modeling in order to facilitate analysis of the application and middleware components.

Acknowledgments. This work is supported by DARPA under the Program Composition of Embedded System (PCES) program within the Information Exploitation office (DARPA/IXO). The authors also acknowledge Craig Rodrigues, Joseph Loyall, and Richard Schantz at BBN Technologies for valuable discussions and comments.

References

- [1] Jean Bézivin, "From Object Composition to Model Transformation with the MDA," *Technology of Object-Oriented Languages and Systems (TOOLS)*, Santa Barbara, California, August 2001.
- [2] Carol Burt, Barrett Bryant, Rajeev Raje, Andrew Olson, Mikhail Auguston, "Quality of Service Issues Related to Transforming Platform Independent Models to Platform Specific Models," *The 6th International Enterprise Distributed Object Computing Conference (EDOC)*, Switzerland, September 2002.
- [3] David Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, June 1987, pp. 231–274.
- [4] David Karr, Craig Rodrigues, Joseph Loyall, Richard Schantz, Yamuna Krishnamurthy, Irfan Pyarali, and Douglas Schmidt, "Application of the QuO Quality-of-Service Framework to a Distributed Video Application," *International Symposium on Distributed Objects and Applications*, Rome, Italy, September 2001.
- [5] Akos Lédeczi, Arpad Bakay, Miklos Maroti, Peter Volgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gábor Karsai, "Composing Domain-Specific Design Environments," *IEEE Computer*, November 2001, pp. 44–51.
- [6] Richard Schantz, Joseph Loyall, Michael Atighetchi, and Partha Pal, "Packaging Quality of Service Control Behaviors for Reuse," *The 5th IEEE Symposium on Object-oriented Real-time distributed Computing (ISORC)*, April 2002, Washington, DC.
- [7] Sandeep Neema, "System-Level Synthesis of Adaptive Computing Systems," Ph.D. Dissertation, Vanderbilt University, May 2001.
- [8] Greg Nordstrom, Janos Sztipanovits, Gábor Karsai, and Ákos Lédeczi, "Metamodeling - Rapid Design and Evolution of Domain-Specific Modeling Environments," *International Conference on Engineering of Computer-Based Systems (ECBS)*, Nashville, Tennessee, April 1999, pp. 68–74.
- [9] Richard E. Schantz and Douglas C. Schmidt, "Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications," *Encyclopedia of Software Engineering*, Editors John Marciniak and George Telecki, Wiley and Sons, New York, 2001.

Optimizing Content Management System Pipelines

Separation and Merging of Concerns

Markus Noga¹ and Florian Krüper²

¹ Universität Karlsruhe
Program Structures Group
Adenauerring 20a
76133 Karlsruhe, Germany

`markus@noga.de`

² Sparkasse.de
Rotherstr. 9
10245 Berlin, Germany
`florian@krueper.net`

Abstract. Content management systems support the dissemination and maintenance of documents. In software engineering terms, they separate the concerns of content, application logic and visual styling. Current systems largely maintain this separation of concerns after document deployment. Their runtime processing pipeline is a composition of generators, or document transformations. We exploit commutativity to enable new static evaluations of the composite during document deployment. Unlike traditional caching, we arrive at closed-form composites even for styled, database-driven documents. This eliminates the runtime penalties of a separation of concerns while preserving their software engineering benefits.

1 Introduction

A *content management system*, short *CMS*, supports the process of publishing documents in software. Formerly used exclusively by large organizations, e.g., newspapers, they have become the mainstay of web publishing for medium-size businesses. This paper discusses optimizations that enable economic deployment of a CMS by even smaller businesses and individuals.

Content management is a complex affair. Users have distinct preferences when editing documents. In many cases, publishing these documents also requires significant amounts of application logic. In a web context, e.g., navigation must be generated. Finally, entire professions build a living around visualizing documents. Thus, the notion of a *separation of concerns* [1] is as central to content management as it is to software engineering. A CMS primarily separates the concerns of *content*, *logic* and *presentation*.

Separating concerns allows their development to be temporally, spatially and personally separated as well. E.g., visuals take a significant fraction of initial

development time for customer-specific CMS solutions. In terms of remuneration, web designers are cheaper than application logic programmers. Thus, separation of concerns in design and development yields immediate economic benefits.

Unfortunately, webspace providers do not charge for traffic and space alone. They also apply the *utility pricing model* to the runtime environment, i.e., the more complex the runtime, the higher the charges. Therefore, separation of concerns in a deployed system imposes economic penalties. Removing these software engineering artifacts from the deployed system is a competitive advantage.

We aim to reap this advantage. This paper describes an experimental CMS based on composed transformations of documents. On a restricted domain of transformations, we show the composition to be commutative and exploit this property to optimize the deployed system with static evaluation. This eliminates the penalties for separation of concerns during system operation while preserving their design and development benefits. Additionally, our approach increases on-line throughput by an order of magnitude. This enables professional content management on inexpensive webspace accounts.

The remainder of the paper is organized as follows: We examine the state of the art in section 2. In section 3, we introduce an abstract processing model and discuss optimizations on it. Our prototype system implementing these optimizations is described in section 4. section 5 summarizes our results and outlines directions for future work.

2 State of the Art

To clarify a CMS' runtime environment, we first look at technical and economical aspects of renting webspace and examine LAMP, the most popular and economic variant of webspace in more detail. Then, we examine some professional CMS of the open-source and commercial variety. We look at traditional build managers used in programming and examine their suitability for CMS. Finally, we summarize our findings.

2.1 Webspace

The options for renting webspace are bewildering. Offerings range from limited amounts of plain HTML pages, typically offered for free, to personal dedicated servers. Managed service providers like Loudcloud offer a wide array of associated services, from simple email support over 24/7 hotlines up to full site management.

For the purposes of this paper, we disregard managed service providers and employ a simple model of increasing technical complexity. We distinguish webspace, virtual servers and dedicated servers. *Webspace* is storage space on a server. A simple application server or parts thereof may be available, but webspace users have no control over configuration. *Virtual servers* are virtual sandboxes within a server. They support more complex environments like Enterprise JavaBeans [2]. Additionally, virtual servers offer some control over server configurations, but fall short of supporting the installation of arbitrary software. This requires a server entirely under user control, that is, a *dedicated server*.

Rental fees increase in line with technical complexity. Depending on the configuration, webspace is available at low or no cost. Virtual servers impose moderate costs, and dedicated servers even higher ones. These costs are duplicated with the user: configuring virtual or dedicated servers requires technical expertise that must be bought. These types of servers may also require additional maintenance personnel.

As more and more smaller companies and individuals come online, the growth of the internet occurs mainly within low-cost webspace. This market segment restricts server providers to free software environments as the most economical choice of implementation. We explore that environment in more detail below.

2.2 LAMP

What technologies are used in cheap webspace accounts? Usually, free open-source ones, the foremost example being *LAMP* [3]. The acronym refers to its constituent parts: the Linux operating system, the Apache web server, the MySQL relational database and one of the scripting languages PHP, Perl or Python, usually PHP.

Linux, Apache and MySQL are fairly well-known. The newest addition to LAMP is *PHP* [4], a pre-processor for XML and HTML that seamlessly integrates scripting into the XML and HTML files. Because of its simplicity, PHP is open to less-sophisticated developers for whom Perl or Python present formidable challenges. In a LAMP environment, PHP is the application server. Together with the other programs, it enables the provision of dynamic content and database-backed services.

Monthly web surveys can convey a measure of LAMP's popularity [5]. Currently, roughly $\frac{2}{3}$ of all servers run Apache. PHP is the most popular Apache module, running in about $\frac{1}{2}$ of all Apache servers. These numbers make LAMP the most widely deployed web environment. Most large ISPs provide LAMP environments in the very lowest price segment, although with restricted configurability. In Germany, the two largest providers Strato and Puretec do so [6, 7].

2.3 Open-Source CMS

Open-source CMS abound. On Sourceforge, there are some 800 CMS projects, most of them in combination with PHP or LAMP. We discuss only the technology leader, Apache Cocoon, which runs in a more sophisticated Java environment.

Cocoon focuses on plain content management [8]. The Cocoon architecture follows the architectural pattern *Pipeline* [9]. Here, we present the standard case of the configurable pipeline. Each step is a document transformation and encapsulates a concern in the domain of content management: document content, addition and evaluation of application logic, and finally visual formatting or presentation (see fig. 1).

Technically, Cocoon relies on XML and server-side Java. Content is encoded in XML. Application logic libraries are XSLT scripts that replace XML logic

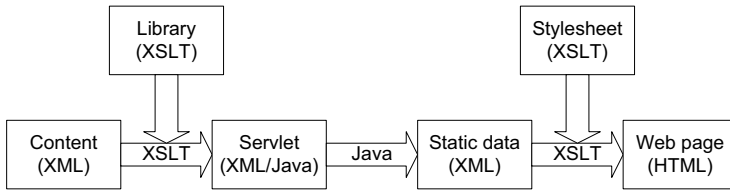


Fig. 1. The Cocoon processing pipeline

tags with server-side Java code embedded in XML. Styling for presentation is also expressed as an XSL transformation. In effect, this is an infrastructure for XML-based domain-specific languages extensible through XSLT rule definitions.

Cocoon requires server-side Java and access to the server configuration. Thus, a virtual server is the minimal runtime environment for Cocoon. In most cases, dedicated servers are required.

2.4 Commercial CMS

Prominent examples of commercial CMS are BroadVision, Gauss, Imperia and Vignette. We discuss Vignette as a classical system and BroadVision as a modern XML-based one.

Vignette Content Suite V6 is a large system split into six modules [10]. Apart from content management and sophisticated caching strategies, the system also caters to personalization, integration and content analysis. Vignette achieves a separation of concerns between content and logic. A separation between logic and presentation is not evident. Content is stored in relational databases and files. Logic takes the form of proprietary modules, TCL scripts or JSPs. Presentation is performed in those scripts. Vignette requires a dedicated server to run.

BroadVision also offers a wide array of tools, which cover personalization, e-commerce and portals in addition to plain CMS [11]. The system achieves a separation of concerns between content, logic and presentation. BroadVision employs XML and relational databases to store content. Logic and scripting interfaces are accessible from Java, JavaScript, COM and others. Like Cocoon, BroadVision uses XSLT for presentation and styling. Like Vignette, BroadVision requires a dedicated server to run.

In general, commercial CMS deal with many issues besides plain content management. They tend towards high complexity and include many proprietary components, although XML technology is being adapted steadily. Unfortunately, heavyweight commercial CMS usually require dedicated server environments.

2.5 Build Managers

Build managers automate the process of building a target from sources. Both targets and sources are software artifacts. *make* [12], *Odin* [13] and *ClearCase* [14] are some well-known tools. They organize builds according to *derivation trees*,

whose nodes are software artifacts and whose edges are individual derivators like compilers or linkers. The three tools make increasingly complex efforts to cache intermediate results: make uses regular files in the user directory, Odin maintains a private cache where source names and tools configurations are encoded into names and ClearCase relies on custom file systems with view capabilities.

However, all three consider the order of individual derivation steps as fixed. Consider fig. 1, where visualization occurs after interpretation. If interpretation hinges on the runtime environment, e.g. because of database accesses, caching must stop before interpretation. Build managers cannot eliminate the final transformations because they do not support commutativity. In the CMS domain, that prohibits deployment of to lightweight platforms.

2.6 Summary

All professional CMS achieve a separation of concerns in design and development to varying degrees. Unfortunately, they maintain that separation in deployment. Their complex runtime environments require virtual or even dedicated servers to run. This makes professional CMS unsuitable for the majority of current and future web page operators. For them, it is necessary to support LAMP environments while preserving software engineering benefits. Thus, the separated concerns must be merged in deployment. Conventional build managers cannot achieve this because they regard the order of individual derivation steps as fixed.

3 Model

In this section, we present an abstract model of a content management system pipeline. We introduce a set of restrictions on pipeline steps that demonstrably ensures commutative composition. Then, we discuss how the resulting preprocessing steps can be expressed in closed form. In the final subsection, we discuss applications to document deployment.

3.1 Processing Model

We model the processing pipeline P of a CMS as a composition of operators $P = O_1 \circ O_2 \circ \dots \circ O_n$. The individual operators $O_i: X \rightarrow X$ are maps on the set of XML documents X .

The classical processing pipeline consists of XSL transformations and language interpretation. Let $X_\sigma = XSLT(\cdot, \sigma)$ be the XSLT transformation operator with script σ and $I_e = interpret(\cdot, e)$ be the programming language interpretation with runtime environment e , which may include file systems and databases. We define interpretation to replace special interpretation elements with well-formed, computed XML fragments.

Thus, the classical processing pipeline is can be expressed as $P = X_s \circ I_e \circ X_l$, where s is the styling transformation and l is the logic library transformation.

Many optimization strategies for database query optimizations rely on the algebraic properties of query operators (see [15]). Here, we focus on associativity and commutativity.

Operator composition is associative. Now, let us assume the composition in $X_s \circ I_e$ to be commutative as well. If that were the case, we could transform the pipeline equation to $P = P' = I_e \circ X_s \circ X_l$. As $X_s \circ X_l$ is independent of the runtime environment e , we could precompute $d' = X_s \circ X_l(d)$ when deploying a document d . The online computation would be reduced to $P''(d) = I_e(d')$.

Unfortunately, $X_s \circ I_e$ does not commute in general. While the identity stylesheet does commute with any logic, there are numerous counterexamples. Consider, e.g., application logic generating a list of random numbers and a stylesheet sorting them. A list of numbers not yet generated cannot be sorted. For commutative composition to hold, we must thus restrict ourselves to a subset of all stylesheets and applications. There is a family of such restrictions: the more restrictions are made on stylesheets, the less restrictions on logic are necessary, and vice versa. Here, we are interested in restrictions on s that leave many degrees of freedom to l .

3.2 Restrictions

We proceed by partitioning the set of admissible XML elements by name into three sets, the structural \mathcal{S} , logical \mathcal{L} and generated \mathcal{G} ones. Additionally, we know the set of interpreter elements \mathcal{I} and the set of HTML elements \mathcal{H} . W.l.o.g., we assume all five sets to be mutually disjoint.

In this framework, X_l transforms XML documents over $\mathcal{S} \cup \mathcal{L}$ to documents over $\mathcal{S} \cup \mathcal{G} \cup \mathcal{I}$. I_e transforms the results to documents over $\mathcal{S} \cup \mathcal{G}$. Finally, X_s transforms these documents to documents over \mathcal{H} . With this framework in place, we are ready to impose restrictions.

1. Interpreter scripts may not generate or reference XML elements $\mathcal{S} \cup \mathcal{G}$, save direct copies of well-formed XML elements in the stylesheet with optional generation of attribute values.
2. Attribute value computations are free of side-effects.
3. Styling transformations are restricted to statically:
 - copying by default
 - changing element or attribute names
 - adding arbitrary XML elements immediately before or after opening or closing tags, provided the result is well-formed. It may contain copies of attribute values.

Do these restrictions guarantee commutative composition? To the XSLT processor, interpreter scripts are simply text interspersed with elements. Therefore, the XSLT script s can run on $X_l(d)$. As it copies by default, the interpreter markup \mathcal{I} remains untouched, but structural and generated elements are transformed. The result of $X_s \circ X_l(d)$ is an XML document over $\mathcal{H} \cup \mathcal{I}$.

XML elements are generally not valid in interpreter scripts, so they must be contained in interpreter string literals. We enumerate the string literals l_i containing XML elements sequentially.

When changing names or adding XML content as per 3, this transforms the l_i to l'_i . The l'_i are valid interpreter literals because they can contain well-formed XML elements to begin with (per 1). The XSLT processor may reorder attributes arbitrarily within an element, but per 2, this cannot be observed on the outside. As 3 prohibits reordering or deletion of element contents, the number and order of the l'_i and the interspersed code remains invariant.

Thus, $X_s \circ X_l(d)$ can be interpreted with I_e . Their composition, we recall, was P' . Moreover, when simultaneously replacing the l_i, l'_i with "li" in this P and P' , their static and dynamic semantics are the same, as there are no control dependencies on the contents of the l_i per 1. By construction of the l'_i , the resulting output documents are equal.

Therefore, the above restrictions are sufficient for commutative composition.

3.3 Closed Form

Commutative composition allows to preprocess the application library and styling transformations at document deployment, reducing on-line processing to a single interpreter run. We now ask if the initial composed transformations may be expressed in closed form. As they are both XSL transformations, we are interested in combining them into a single one.

We exploit the property that XSL transformation scripts are themselves XML documents. If \mathcal{X} is the set of XSLT tags, the script l of the application library transformation X_l is an XML document over $\mathcal{X} \cup \mathcal{S} \cup \mathcal{G} \cup \mathcal{I}$. As X_s transforms elements in $\mathcal{S} \cup \mathcal{G}$ only, $X_s(l)$ is an XML document over $\mathcal{X} \cup \mathcal{I} \cup \mathcal{H}$. Because X_s does not reorder, $X_s(l)$ is a valid XSLT script. It replaces elements in \mathcal{L} with interpreter elements in \mathcal{I} containing script text and styled elements from \mathcal{H} .

A transformation with script $X_s(l)$ still leaves elements in $\mathcal{S} \cup \mathcal{G}$ invariant. However, as \mathcal{S}, \mathcal{G} and \mathcal{L} are mutually disjoint, the transformation rules in s and $X_s(l)$ may be combined without conflicts. We name the resulting script l' .

Analogous to the preceding section, we impose the following restriction on l :

4. The library transformation may not generate XML elements in $\mathcal{S} \cup \mathcal{G}$, save by copying input or stylesheet elements with optional generation of attribute values. It may not reference result tree fragments.

As XSL transformations are functional except for output, 2 of the preceding section always holds for the transformation script, too. Together with 3, the rationale of the preceding proof applies analogously. Thus, $X_{l'} = X_s \circ X_l$ holds.

l' is a static evaluation of s on all possible outputs of l . In terms of [16], we can also interpret l as a program family and s as a generator specification to obtain the individual family member l' .

3.4 Applications

In content management, deployment makes data on a production system available on a server. Simplifying things to text, there are three kinds of data: content, logic and presentation styling. When deploying them, commutative composition and closed form can be profitably employed to reduce runtime system requirements and enhance processing throughput.

When deploying new logic or presentation styling, we compute the closed form l' of the composite transformation on the client. We then redeploy the affected content.

When deploying content documents d , we apply the closed form of the composite transformation and upload $X_{l'}(d)$ to the server. Even for database-driven documents, this reduces online processing to a single step, which traditional CMS and build managers cannot. If I_e is independent of e , we may even upload $I_e \circ X_{l'}(d)$. This is equivalent to traditional caching of generated documents.

Under this scheme, content management systems may target inexpensive LAMP environments. Simultaneously, the software engineering benefits of a separation of concerns are fully retained.

4 Prototype

Our experimental system is a modification of the original Cocoon design. To support LAMP, we replaced server-side Java with PHP scripting. Content is still encoded in XML. Application logic libraries are XSLT scripts that replace XML logic tags with PHP scripts embedded in XML. Styling is again expressed as an XSL transformation.

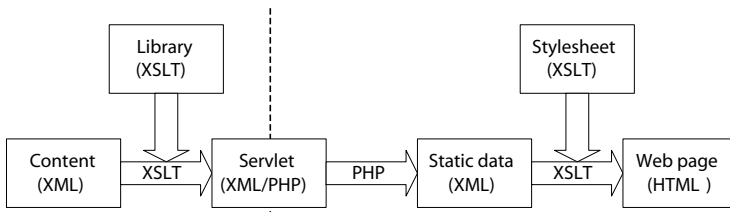


Fig. 2. The experimental processing pipeline. The dashed line represents the caching boundary for dynamic documents.

Fig. 2 shows the resulting design. The dashed line separates cacheable and non-cacheable parts of the system for dynamic documents. We refer to the full pipeline as the *standard* one and the pipeline to the right of the dashed line as the *caching* one.

Because the results in the previous section were obtained using an abstract notion of a CMS pipeline, they apply to our system as well as to Cocoon. Fig. 3 shows the *optimized* version of our system.

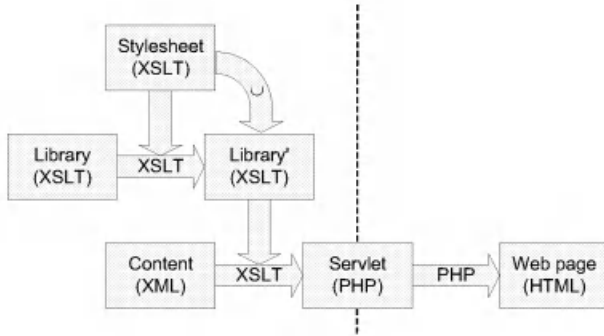


Fig. 3. The optimized processing pipeline. To the left the production system, to the right the deployed system.

4.1 Benchmarks

We prepared five benchmarks to evaluate the performance of the optimized pipeline as compared to the caching and standard ones. They are designed to closely mirror a variety of usage scenarios.

The *hello* benchmark is a simple hello world document. It prints a short message and applies basic styling to it. This benchmark is designed to establish a lower bound on the individual pipelines' execution times.

The *simplifiedb* benchmark is a database-driven document. It connects to a MySQL database of college classes, retrieves the list of classes in the current semester and applies plain styling to it (see the following subsection). *simplifiedb* represents first-generation database-driven web pages.

The *db* benchmark is a more sophisticated cousin of *simplifiedb*. In addition to database operations, it also performs fairly complex visual styling, including cascading stylesheets and table layouts. This benchmark should be fairly representative for contemporary database-driven web pages.

The *bib* benchmark visualizes a list of bibliography entries given as an XML file. There is comparatively little logic, but styling complexity is comparable to or slightly higher than for *db*. This benchmark represents contemporary dynamic web pages that do rely on databases.

Finally, the *fract* benchmark computes part of the Mandelbrot set. It visualizes the results as a colored HTML table. This benchmark represents applications with complex logic but small to negligible libraries and styling.

4.2 A Benchmark in Depth

To demonstrate the optimized pipeline, we reproduce the *simplifiedb* in full and discuss its execution. Standard namespaces, default copying rules and error handling code have been removed to maintain readability.

Consider the following document *d* built from the structural elements **page** and **title** along with the logic element **classes**, which represents a database query to retrieve college classes in a given semester.

```
<page>
  <title>Current Classes</title>
  <classes year="2002" semester="S"/>
</page>
```

The following library transformation l replaces the logic tag in d with server-side PHP code that actually performs the query. Notice how attributes of `classes` are mixed into the code.

```
<xsl:stylesheet version="1.0">
  <xsl:template match="classes">
    <script language="php">
      print "<classlist>";
      $db=mysql_connect("localhost","root","");
      mysql_select_db("phptest",$db);
      $query=mysql_query("select name from classes
        where year=<xsl:value-of select="@year"/>
        and semester='<xsl:value-of select="@semester"/>',$db);
      while($row=mysql_fetch_array($query))
        printf('<class name="%s"/>',$row['name']);
      print "</classlist>";
    </script>
  </xsl:template>
</xsl:stylesheet>
```

As l amounts to simple text substitution, we do not reproduce the transformation result $X_l(d)$ here. The following XSL transformation s performs simple presentation styling on $I_e \circ X_l(d)$:

```
<xsl:stylesheet version="1.0">
  <xsl:output method="html"/>

  <xsl:template match="page">
    <html><head><title>
      <xsl:value-of select="title"/>
    </title></head><body>
      <xsl:apply-templates/>
    </body></html>
  </xsl:template>

  <xsl:template match="title">
    <h1><xsl:value-of select="."/></h1>
  </xsl:template>

  <xsl:template match="classlist">
    <ul><xsl:apply-templates/></ul>
  </xsl:template>
```

```

<xsl:template match="class">
  <li><xsl:value-of select="@name"/></li>
</xsl:template>
</xsl:stylesheet>

```

The restrictions in section 3.2 hold by design for l and s , so commutative composition can be applied. Computing the closed form according to section 3.3 yields this styled version l' of the logic library:

```

<xsl:stylesheet version="1.0">
  <xsl:output method="html"/>

  <xsl:template match="classes">
    <script language="php">
      print "<ul>";
      $db=mysql_connect("localhost","root","");
      mysql_select_db("phptest",$db);
      $query=mysql_query("select name from classes
        where year=<xsl:value-of select="@year"/>
        and semester='<xsl:value-of select="@semester"/>'",$db);
      while($row=mysql_fetch_array($query))
        printf('<li>%s</li>',$row['name']);
      print "</ul>";
    </script>
  </xsl:template>

  <xsl:template match="page">
    <html><head><title>
      <xsl:value-of select="title"/>
    </title></head><body>
      <xsl:apply-templates/>
    </body></html>
  </xsl:template>

  <xsl:template match="title">
    <h1><xsl:value-of select="."/></h1>
  </xsl:template>

  <xsl:template match="classlist">
    <ul><xsl:apply-templates/></ul>
  </xsl:template>

  <xsl:template match="class">
    <li><xsl:value-of select="@name"/></li>
  </xsl:template>
</xsl:stylesheet>

```

Applying l' to d yields the styled document $X_{l'}(d)$:

```
<html>
  <head><title>Current Classes</title></head>
  <body>
    <h1>Current Classes</h1>
    <script language="php">
      print "<ul>";
      $db=mysql_connect("localhost","root","");
      mysql_select_db("phptest",$db);
      $query=mysql_query("select name from classes
        where year=2002 and semester='S',$db);
      while($row=mysql_fetch_array($query))
        printf('<li>%s</li>',$row['name']);
      print "</ul>";
    </script>
  </body>
</html>
```

The above l' is deployed on the server. At runtime, only PHP interpretation remains to be done. If the database contains a fictitious list inspired by classes offered at our institute, interpretation results in the following static document $I_e \circ X_{l'}(d)$ for client consumption:

```
<html>
  <head><title>Current Classes</title></head>
  <body>
    <h1>Current Classes</h1>
    <ul>
      <li>Compiler Construction</li>
      <li>IA64 DivX Lab</li>
      <li>Software Components</li>
    </ul>
  </body>
</html>
```

4.3 Measurements

The optimized pipeline eliminates all but one stage of the transformation process. The caching pipeline eliminates the initial pipeline stage. Therefore, measurements in a client-side environment would skew the results in favour of the optimized and caching variants due to startup costs associated with the tools implementing individual pipeline stages, i.e., XSLT processors and PHP interpreters. Only a server-side environment, where the associated tools persist in main memory, allows a fair comparison.

This prevents classic build managers like make from participating in measurements. Instead, we implemented a custom framework for performance measurements in PHP. Using `eval`, output capturing and experimental XSLT extensions, startup costs can be avoided. To cope with system clock jitter, the framework loops each benchmark for a minimum of two seconds execution time.

Measurements were taken on an 1.2 GHz Mobile Pentium III-M with 512 MB of memory under Windows XP Professional. We executed our benchmarks with PHP 4.1.2 using version 0.9 of the Sablotron XSLT processor [17] and the NT executable of MySQL 3.23.49. Fig. 4 shows the results. The horizontal axis shows the individual benchmarks. Processing times in percentages of standard pipeline processing time are plotted along the vertical axis. The standard pipeline has been decomposed into its constituent phases, represented by bar graphs. Due to instruction cache effects, totals may differ from 100%. The caching and optimized pipelines are plotted as lines.

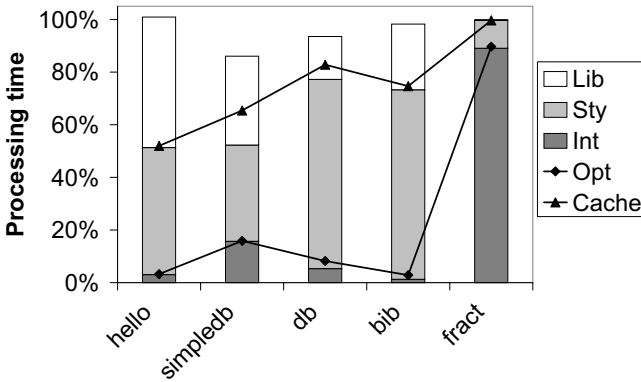


Fig. 4. Performance comparison: The standard, caching and optimized pipelines.

Looking at the curves, we see why caching is a mainstay of traditional CMS. It removes the overhead for library transformation, outperforming standard processing by a factor of 1.0–1.9x on our benchmarks. Clearly, caching is a profitable strategy.

The optimized pipeline outperforms both the standard and caching variants on our benchmarks. The speedups over caching range from 1.1 – 26.2x, those over standard processing from 1.1 – 35.1x. Visually, benefits appear to be largest for the benchmarks using complex visual styling. The fact that our optimization renders a separate styling stage obsolete would support this interpretation. However, numerical speedups for *hello* exceed those for the database examples. Those two benchmarks demonstrate that increases in computing time have a proportionally larger impact on the optimized system than on the other variants.

We postulate that execution time for the single remaining optimized pipeline step is dominated by logic execution time. Reporting is reduced to simple copying, so reporting time is almost irrelevant for typical web-based document sizes.

4.4 Tools and Quoting

In preparing our benchmark, we noticed that quoting is an issue with two of the tools used, PHP and Sablotron.

XML mandates that the `<` and `&` characters be escaped with `<` and `&` in regular text, respectively. Unfortunately, the PHP interpreter does not recognize these representations as operators. Thus, PHP requires non-well-formed XML for input. This is probably due to its origins with the HTML community which has long been accustomed to laxer syntactical constraints. In our benchmarks, we resolved this issue through automatic postprocessing. For the long run, we would like the PHP community to address this issue, as it is critical to XML and XHTML embedding of PHP.

In a similar vein, the Sablotron XSLT processor does not preserve the quoting style used in element attributes. Whether single or double quotes are used, transformation results always employ double quotes. This is not a Sablotron issue alone — the XSLT transformation model and the underlying XML information set model simply do not contain quoting information. Most applications are unaffected by this. However, due to the embedding of XML attributes in language string constants, any change in quotation characters may break language syntax. For our benchmarks, we resolved this issue through manual postprocessing.

More viable alternatives come in many shapes. Of course, we would welcome the addition of quotation information to the XML information set and XSLT processing models. But this may take a long time. Maybe adding a new switch to XSLT processors that allows users to toggle the standard quotation character would already be sufficient. Alternatively, languages like PERL can use arbitrary characters as quotes via the `q` and `qq` prefixes, respectively. However, these quick suggestions address individual processors and languages only.

4.5 Impact of Restrictions

How restrictive are the restrictions from section 3.2 in practice?

Restriction 1 prohibits the interpreter to generate or reference XML elements in $S \cup G$, other than copying them. In practice, this means that element names cannot be constructed in a piecemeal way, e.g. from string variables. However, as DTD grammars are finite, the possible elements names can be enumerated and used in a switch construct. Thus, restriction 1 may lead to an inconvenient notation, but it does not limit the expressive power of interpreter scripts.

Restriction 2 mandates that attribute value computations must be free of side effects. It is necessary because XML transformations may not preserve attribute order. In practice, attribute values can be precomputed and stored in variables, so restriction 2 does not limit the power of interpreter scripts, either. However, the quoting issues discussed in the previous subsection still render the process of using variables somewhat cumbersome.

Restriction 3 restricts styling to stateful tree decoration. This is a true restriction that destroys the Turing-completeness of the styling phase. Some restriction on styling is necessary, because it can only commute with interpretation if it does

not rely on values computed by the interpreter. Any such restriction is bound to mingle logic and styling concerns to some degree.

In practice, we noticed restriction 3 several times. E.g., we initially tried to convert recursion depths to HTML colors in the styling phase of the fractal example, only to realized that recursion depths are unavailable in that phase.

Finally, restriction 4 is a restriction of the library transformation equivalent to restriction 1 on the interpreter stage. Thus, the same rationale applies.

5 Conclusions

The separation of concerns inherent to CMS may be beneficially dissolved during deployment using commutative composition and closed forms. This requires a pair of restrictions to application logic and presentation styling transformations. We devised restrictions that are lax on logic and stricter on styling. We showed them to be sufficient for commutativity and the computation of a closed form, enabling us to apply static preprocessing in the deployment phase.

Our approach lowers software requirements on the deployment platform. By eliminating software technology artifacts in the deployed system, we defeated the webspace utility pricing mechanism. In theory, it is now possible to deploy a modern content management system with inexpensive or free providers.

While some of our benchmarks are synthetic, we chose them in a representative manner. On these benchmarks, commutative composition and closed forms increase processing performance by an order of magnitude on average.

We explain this acceleration with the elimination of all but one pipeline stage from online execution. Reporting apart, this reduces online processing times to logic execution time. Simultaneously, the benefits of separating logic from styling concerns are preserved. Historically, the web progresses to ever more visually intricate document representations. The importance of all optimizations that address or eliminate late pipeline stages should increase in line with that trend.

Of cause, there is plenty of room for future work. The presented set of restrictions are not minimal. Techniques like partial evaluation and abstract interpretation may be capable of tightening them further. Entirely different tradeoffs between logic and presentation restrictions are conceivable and remain to be explored. Practical experience in preparing the benchmarks showed the styling restrictions to be the most pressing ones, so relaxing them is a priority.

We have not addressed the question of validating conformance to a set of restrictions. In this paper and benchmarks therein, we manually asserted conformance by careful design. I.e., we looked hard at the sources, a process prone to errors. Automatic validation would remove that potential for errors. However, to grapple problems of decidability, any such validation of application logic must by nature be aggressive. Whether the resulting fake negatives are relevant in practice remains to be evaluated.

The question of validation leads to other questions about real-world usability. We did not address any of the issues raised by multiple, distributed users in this paper. Neither did we address asset management, interface concerns or any of

the more advanced modules found in commercial CMS. To truly bridge the gap between content management and the end user, we have to extend our prototype into a more complete environment.

References

1. Dijkstra, E.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs, NJ (1976)
2. SUN Microsystems: Enterprise JavaBeans 1.1 specification. <http://java.sun.com/products/ejb/docs.html> (1999)
3. O'Reilly: LAMP. <http://www.onlamp.net/> (2002)
4. The Apache Foundation: PHP hypertext processor. <http://www.php.net/> (2002)
5. Security Space: Internet research reports. http://www.securityspace.com/s_survey_data/ (2002)
6. Strato Medien AG: Strato Medien AG. <http://www.strato.de/> (2002)
7. Puretec: 1&1 WebHosting. <http://www.puretec.de/> (2002)
8. The Apache Foundation: Apache cocoon. <http://xml.apache.org/cocoon2/> (2001)
9. Shaw, M., Graham, D.: Software Architecture in Practice – Perspectives on an Emerging Discipline. (1996)
10. Vignette: Vignette content suite v6. <http://www.vignette.com/> (2002)
11. BroadVision: Broadvision. <http://www.broadvision.com/> (2002)
12. Feldman, S.I.: Make-a program for maintaining computer programs. *Software - Practice and Experience* **9** (1979) 255–65
13. Clemm, G.M.: The Odin system. *Lecture Notes in Computer Science* **1005** (1995) 241–262
14. Leblang, D.B.: The CM challenge: Conguration management that works. In Tichy, W.F., ed.: *Configuration Management*, Wiley (1994)
15. Silberschatz, A., Korth, H.F., Sudarshan, S.: 12. In: *Database System Concepts*. 4th edn. McGraw-Hill (2001)
16. Eisenecker, U.W., Czarnecki, K.: *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley (2000)
17. Ginger Alliance: Sablotron XSLT, DOM and XPath processor. http://www.gingerall.com/charlie/ga/xml/p_sab.xml (2002)

Component-Based Programming for Higher-Order Attribute Grammars

João Saraiva

Department of Computer Science,
University of Minho, Braga, Portugal
`jas@di.uminho.pt`

Abstract. This paper presents techniques for a component-based style of programming in the context of higher-order attribute grammars (HAG). Attribute grammar components are “*plugged in*” into larger attribute grammar systems through higher-order attribute grammars. Higher-order attributes are used as (intermediate) “gluing” data structures.

This paper also presents two attribute grammar components that can be re-used across different language-based tool specifications: a visualizer and animator of programs and a graphical user interface AG component. Both components are reused in the definition of a simple language processor. The techniques presented in this paper are implemented in LRC: a purely functional, higher-order attribute grammar-based system that generates language-based tools.

1 Introduction

Recent developments in programming languages are changing the way we construct programs. Programs are now a collection of generic, reusable, off-the-shelf program components that are “*plugged in*” to form larger and powerful programs. In such an architecture, intermediate gluing data structures are used to convey information between different program components: a component constructs (produces) an intermediate data structure which is used (consumed) by other component.

In the context of the design and implementation of language-based tools, attribute grammars provide powerful properties to improve the productivity of their users, namely, the static scheduling of computations. Indeed, an attribute grammar writer is neither concerned with breaking up her/his algorithm into different traversal functions, nor is she/he concerned in conveying information between traversal functions (*i.e.*, how to pass intermediate values computed in one traversal function and used in following ones). A second important property is that circularities are statically detected. Thus, the existence of cycles, and, as a result, the non-termination of the algorithms, is detected statically. That is to say that for (ordered) attribute grammars the termination of the programs for all possible inputs is statically guaranteed. A third characteristic is that attribute grammars are declarative. Furthermore, they are executable: efficient

declarative (and non-declarative) implementations (called attribute evaluators) are automatically derived by using well-known AG techniques. Finally, incremental implementations of the specified tools can be automatically generated from an attribute grammar.

Despite these advantages, attribute grammars are not of general use as a language-based tool specification formalism. In our opinion, this is due to two main reasons: firstly, there is no efficient, clear and elegant support for a component-based style of programming within the attribute grammar formalism. Although an efficient form of modularity can be achieved in AGs when each semantic domain is encapsulated in a single AG component [GG84,LJPR93,KW94][CDPR98,SS99b,dMBS00], the fact is that there is no efficient support within the AG formalism for an easy reuse of such components. That is, how can a grammar writer “*plug in*” an AG component into her/his specification? How are those AG components *glued* together? How is information passed between different AG components? How can the separate analysis and compilation of components be achieved? Obviously we wish to provide answers to these questions within the attribute grammar formalism itself. Secondly, there is a lack of good generic, reusable attribute grammar components that can be easily “*plugged in*” into the specifications of language-based tools. Components that are themselves written in the AG formalism.

The purpose of this paper is two-fold: firstly, to propose a component-based style of programming in the (higher-order) attribute grammar formalism. This means that attribute grammar components are efficiently and easily “*plugged-into*” an AG specification via higher-order attributes. In this approach, one AG component defines a higher-order attribute which is decorated according to the attribute equations defined by another AG component.

Secondly, to introduce two generic, reusable and off-the-shelf AG components. These components are themselves defined in the HAG formalism and provide modern and powerful properties to visualize, animate and interact with language-based tools.

This paper is organized as follows: Section 2 presents higher-order attribute grammars, its notation and provides a simple example that will be used throughout the paper. Section 3 introduces HAG component-based programming and presents two generic AG components: a visualization and animation component (Section 3.1) and graphical user interface component (Section 3.2). Section 4 discusses related work and Section 5 contains the conclusions.

2 Higher-Order Attribute Grammars

The techniques presented in this paper are based on the *higher-order attribute grammar* formalism [VSK89]. Higher-Order Attribute Grammars are an important extension to the attribute grammar formalism. Conventional attribute grammars are augmented with *higher-order attributes*, the so-called *attributable attributes*. Higher-order attributes are attributes whose value is a tree. We may associate, once again, attributes with such a tree. Attributes of these so-called

higher-order trees, may be higher-order attributes again. Higher-order attribute grammars have four main characteristics:

- First, when a computation can not be easily expressed in terms of the inductive structure of the underlying tree, a better suited structure can be computed before. Consider, for example, a language where the abstract grammar does not match the concrete one. Consider also that the semantic rules of such a language are easily expressed over the abstract grammar rather than over the concrete one. The mapping between both grammars can be specified within the higher-order attribute grammar formalism: the attribute equations of the concrete grammar define a higher-order attribute representing the abstract grammar. As a result, the decoration of a concrete syntax tree constructs a higher-order tree: the abstract syntax tree. The attribute equations of the abstract grammar define the semantics of the language.
- Second, semantic functions are redundant. In higher-order attribute grammars every computation can be modelled through attribution rules. More specifically, inductive semantic functions can be replaced by higher-order attributes. For example, a typical application of higher-order attributes is to model the (recursive) lookup function in an environment. Consequently, there is no need to have a different notation (or language) to define semantic functions in AGs. Moreover, because we express inductive functions by attributes and attribute equations, the termination of such functions is statically checked by standard AG techniques (*e.g.*, the circularity test).
- The third characteristic is that part of the abstract tree can be used directly as a value within a semantic equation. That is, grammar symbols can be moved from the syntactic domain to the semantic domain.
- Finally, as we will describe in this paper, attribute grammar components can be “glued” via higher-order attributes.

These characteristics make higher-order attribute grammars particularly suitable to model language-based tools [TC90, Pen94, KS98, Sar99].

2.1 The Block Language

Consider a very simple language that deals with the scope rules of a block structured language: a definition of an identifier x is visible in the smallest enclosing block, with the exception of local blocks that also contain a definition of x . In the latter case, the definition of x in the local scope hides the definition in the global one.

We shall analyse these scope rules via our favorite (toy) language: the BLOCK language¹. One sentence in BLOCK consists of a *block*, and a block is a (possibly empty) list of *statements*. A statement is one of the following three things: a *declaration* of an identifier (such as `decl a`), the *use* of an identifier (such as `use a`), or a nested *block*. Statements are separated by the punctuation symbol

¹ The BLOCK language, that we introduced in [SSK97, Sar99], has become a popular example to study the static scheduling of “circular” definitions [dMPJvW99, Law01]

“;” and blocks are surrounded by square brackets. A concrete sentence in this language looks as follows:

```

sentence = [ use x ; use y ; decl x ;
             [ decl y ; use y ; use w ] ;
             decl y ; decl x
           ]

```

This language does not require that declarations of identifiers occur before their first use. Note that this is the case in the first two applied occurrences of *x* and *y*: they refer to their (latter) definitions on the outermost block. Note also that the local block defines a second identifier *y*. Consequently, the second applied occurrence of *y* (in the local block) refers to the inner definition and not to the outer definition. In a block, however, an identifier may be declared once, at the most. So, the second definition of identifier *x* in the outermost block is invalid. Furthermore, the BLOCK language requires that only defined identifiers may be used. As a result, the applied occurrence of *w* in the local block is invalid, since *w* has no binding occurrence at all.

We aim to develop a program that analyses BLOCK programs and computes a list containing the identifiers which do not obey to the rules of the language. In order to make the problem more interesting, and also to make it easier to detect which identifiers are being incorrectly used in a BLOCK program, we require that the list of invalid identifiers follows the sequential structure of the input program. Thus, the semantic meaning of processing the example sentence is [*w*,*x*].

The BLOCK language does not force a *declare-before-use* discipline. Consequently, a conventional implementation of the required analysis naturally leads to a program that traverses each block twice: once for processing the declarations of identifiers and constructing an environment and a second time to process the uses of identifiers (using the computed environment) in order to check for the use of non-declared identifiers. The uniqueness of identifiers is checked in the first traversal: for each newly encountered identifier declaration it is checked whether that identifier has already been declared at the same lexical level. In this case, the identifier has to be added to a list reporting the detected errors. The straightforward algorithm to implement the BLOCK processor looks as follows:

1st Traversal	2nd Traversal
- <i>Collect the list of local definitions</i>	- <i>Use the list of definitions as the global environment</i>
- <i>Detect duplicate definitions (using the collected definitions)</i>	- <i>Detect use of non defined names</i>
	- <i>Combine “both” errors</i>

As a consequence, semantic errors resulting from duplicated definitions are computed during the first traversal, and errors resulting from missing declarations, in the second one. Thus, a “gluing” data structure has to pass explicitly the detected errors from the first to the second traversal, in order to compute the final list of errors in the desired order.

2.2 The Attribute Grammar for the Block Language

In this section we shall describe the program `block` in the traditional attribute grammar paradigm. To define the structure of the BLOCK language, we start by introducing one context-free grammar defining the abstract structure of Block. Then, we extend this grammar with attributes and the attribution rules.

We associate an inherited attribute *dcli* of type *Env* to the non-terminal symbols *Its* and *It* that define a block. The inherited environment is threaded through the block in order to accumulate the local definitions and in this way synthesizes the total environment of the block. To distinguish between the same identifier declared at different levels, we use an attribute *lev* that distributes the block's level. We associate a synthesized attribute *dclo* to the non-terminal symbols *Its* and *It*, which defines the newly computed environment. The total environment of a block is passed downwards to its body in the attribute *env* in order to detect applied occurrences of undefined identifiers. Every block inherits the environment of its outer block. The exception is the outermost block: it inherits an empty environment. To synthesize the list of errors we associate the attribute *errs* to *Its* and *It*.

The static semantics of the BLOCK language are defined in the attribute grammar presented in Fragment 1. We use a *standard* AG notation: productions are labelled for future references. Within the attribution rules of a production, different occurrences of the same symbol are denoted by distinct subscripts. Inherited (synthesized) attributes are prefixed with the down (up) arrow \downarrow (\uparrow). Pseudo terminal symbols are syntactically referenced in the AG, *i.e.*, they are used directly as values in the attribution rules. The attribution rules are written as HASKELL-like expressions. Copy rules are included in the AG specification (although there are well-known techniques to omit copy rules, in this paper, we prefer to explicitly define them). The semantic functions *mBIn* (standing for “must be in”) and *mNBIn* (“must not be in”) define usual lookup operations².

$Its < \downarrow lev : Int, \downarrow dcli : Env, \downarrow env : Env$	$It < \downarrow lev : Int, \downarrow dcli : Env, \downarrow env : Env$
$, \uparrow dclo : Env, \uparrow errs : Err >$	$, \uparrow dclo : Env, \uparrow errs : Err >$
$Its = \text{Nil}Its$	$It = \text{Use } String$
$Its.dclo = Its.dcli$	$It.dclo = It.dcli$
$Its.errs = []$	$It.errs = mBIn (String, It.env)$
$ \text{Cons}Its\ It\ Its$	$ \text{Decl } String$
$It.dcli = Its_1.dcli$	$It.dclo = (Pair\ String\ It.lev) : It.dcli$
$Its_2.env = Its_1.env$	$It.errs = mNBIn (Pair\ String\ It.lev, It.dcli)$
$It.env = Its_1.env$	$ \text{Block } Its$
$Its_2.dcli = It.dclo$	$It.dclo = It.dcli$
$Its_1.dclo = Its_2.dclo$	$Its.dcli = It.env$
$It.lev = Its_1.lev$	$Its.lev = It.lev + 1$
$Its_2.lev = Its_1.lev$	$Its.env = Its.dclo$
$Its_1.errs = It.errs ++ Its_2.errs$	$It.errs = Its.errs$

Fragment 1: The BLOCK attribute grammar.

² These inductive functions can be defined via higher-order attributes. Indeed, in the BLOCK HAG presented in [Sar99], we have such an example.

It is common practice in attribute grammars to use additional non-terminals and productions to define new data types and constructor types, respectively. The type *Env* and the constructor function *Pair* are examples of that:

```

Tuple = Pair      String Int
Env   = ConsEnv Tuple Env
        | NilEnv
Err   = ConsErr String Err
        | NilErr

```

Note that, the type *Env* is isomorphic with non-terminal *Env*: the term constructor functions *ConsEnv* and *NilEnv* correspond to the HASKELL built-in list constructor functions *:* and *[]*, respectively. Roughly speaking, non-terminals define tree type constructors and productions define value type constructors. We will use both notations to define and to construct value types.

To make the AG more readable, we introduce a root non-terminal so that we can easily write the attribution rules specifying that the initial environment of the outermost block is empty (*i.e.*, the root is context-free) and that its lexical level is 0.

```

P <  $\uparrow$  errs : Err >
P = Root Its
    Its.dcli = []
    Its.lev  = 0
    Its.env  = Its.dclo
    P.errs  = Its.errs

```

The above fragment includes a typical equation where a inherited attribute (*env*) depends on a synthesized attribute (*dclo*) of the same non-terminal (*Its*). Although such dependencies are natural in attribute grammars they may lead to complex and counterintuitive solutions in other paradigms (functional, imperative, etc), because they induce additional traversal functions which have to be explicitly “glued” together to convey information between them.

The AG fragments presented so far formally specify the static semantics of the BLOCK language. A higher-order extension to this AG will be presented in next section, where we introduce our component-base programming techniques.

3 Gluing Grammar Components via Higher-Order Attribute Grammars

In functional programming, it is common practice to use intermediate data structures to convey information between functions. One function constructs the intermediate data structure which is destructed by another one. The intermediate data structure is the component “glue”. We will mimic this approach in the higher-order attribute grammar setting: an AG component defines (or, at attribute evaluation time, constructs) a higher-order attribute (*i.e.*, a tree-like data structure), which is used (or decorated) by the other AG component.

This gluing of AG components is defined in the HAG formalism itself as follows: consider, for example, that an AG component, say AG_{reuse} , expresses some algorithm \mathcal{A} over a grammar rooted X , and suppose that we wish to express the same algorithm when defining a new grammar, say AG_{new} . Under the higher-order formalism this is done as follows: firstly, we define an attributable attribute, say a with type X , in the productions, say P , of AG_{new} where we need to express algorithm \mathcal{A} . Secondly, we extend AG_{new} with attributes, whose types are the types (*i.e.*, non-terminals) defined in AG_{reuse} , and attribute equations, where the semantic functions are the constructors (*i.e.* productions) of AG_{reuse} . That is, we define attributes that are tree-value attributes. After that, we instantiate the higher-order attribute a with the tree-value attribute of type X constructed in the context of production P . Then, we instantiate the inherited attributes of associated type/non-terminal (*i.e.*, X). Finally, and by definition of HAGs, the generated synthesized attribute occurrences of a are defined by the attribute equations of AG_{reuse} . They are ready to be used in the attribute grammar specification, like any other first-order attribute.

Notice that by expressing the gluing of AG components within the AG formalism itself, we are able to use all the standard attribute grammar techniques, *e.g.*, the efficient scheduling of computations and the static detection of circularities. For example, the inherited/synthesized attributes of the AG components can be “connected” in any order. The HAG writer does not have to be concerned with the existence of cyclic dependencies among AG components: the AG circularity test will detect them for him. Furthermore, we can use attribute grammar techniques to derive efficient implementations for the resulting HAG. For example, we can use our deforestation techniques to eliminate the possibly large intermediate trees that glue the different components [SS99a].

Most of the powerful attribute grammar techniques are based on a global static analysis of attribute dependencies. Thus, they require that the different AG modules/components are “fused” into an equivalent monolithic HAG, before they are analysed. In [SS99b] we have presented techniques to achieve the separate analysis and compilation of AG modules than naturally extend to our component-based approach.

3.1 An Attribute Grammar Component for Visualization and Animation of Language-Based Tools

In order to be more precise about our approach, let us consider the BLOCK language example again. Because this simple toy example has a non-trivial scheduling of computations, we would like to “plug into” the AG specification an AG component that allows us to visualize and animate the BLOCK processor.

Thus, we introduce a generic component for the visualization and animation of AGs. We wish to use this AG as a *generic visual and animation AG component*. We start by defining an abstract grammar that is sufficiently generic to define all possible abstract tree structures we may want to visualize and animate. The grammar is as follows:

<i>TreeViz</i>	= CTreeViz	<i>TreeId</i>	[<i>TreeStmt</i>]
<i>TreeStmt</i>	= CStmtNode	<i>NodeStmt</i>	
	CStmtEdge	<i>EdgeStmt</i>	
	CStmtAttr	<i>AttrStmt</i>	
<i>NodeStmt</i>	= CNodeStmt	<i>NodeId</i>	[<i>Attr</i>]
<i>EdgeStmt</i>	= CEdgeStmt	<i>NodeId</i>	[<i>EdgeRHS</i>] <i>Attrs</i>
<i>EdgeRHS</i>	= CRHSExpNode	<i>EdgeOp</i>	<i>NodeId</i>
<i>Attr</i>	= CAttr	<i>AttrId</i>	<i>AttrVal</i>

The non-terminals *TreeId*, *NodeId*, *EdgeOp*, *AttrId*, *AttrVal* define sequences of characters (strings). In order to make it easier to use this component, we define a set of functions/macros that, using the productions of this AG component, define usual occurring node formats in our trees. Next, we present four functions that define the shape of a node as a record (*attrShapeRecord*), as a circle (*attrShapeCircle*), as the value of a node label (*attrLabel*), and, finally, as a node that contains a value and an arrow to a child node. These functions are presented next.

```

attrShapeRecord      = CAttr "shape" "record"
attrShapeCircle      = CAttr "shape" "circle"
attrLabel label      = CAttr "label" label
nodeRecord1 val father child =
  [CStmtNode (CNodeStmt father) [attrShapeRecord , attrLabel (val ++ "|<c>")]
  ,CStmtEdge (CEdgeStmt "c") [CRHSExpNode "->" child] ]

```

The label is a string that defines the format of the node record. The non-terminal *EdgeOp* is a string defining the direction of the arrow.

The above grammar defines the abstract structure of abstract trees only. To have a concrete graphical representation of the trees, however, we need to map such abstract tree representation into a concrete one. Rather than defining a concrete interface from scratch and implementing a tree/graph visualization system (and reinventing the wheel!), we can synthesize a concrete interface for existing high quality graph visualization systems, *e.g.*, the GraphViz system [GN99]. We omit here the attributes and attribution rules that we have associated to the visualization grammar since they are neither relevant to reuse this component nor to understand our techniques.

To reuse this component, however, we need to know the inherited and synthesized attributes of its root non-terminal, *i.e.*, the *interface* of the AG component. This grammar component is context-free (it does not have any inherited attributes) and synthesizes two attributes *graphviz* and *xml*, both of type string. These two attributes synthesize a textual representation of trees in the GraphViz input language. The first attribute displays trees in the usual graphic tree representation, while the second one uses a Xml tree-like representation (where the production names are the element tags).

TreeViz < ↑ *graphviz* : String, ↑ *xml* : String >

We are now in position to “glue” this component to the BLOCK AG. Let us start by defining the attribute and the equations that specify the construction of the GraphViz representation.

```

Its < ↑ viztree : [TreeStmt] >
Its = NilIts
  Its.viztree = nodeEmptyCircle treeRef(Its)
  | ConsIts It Its
  Its1.viztree = (nodeRecord2 "" treeRef(Its1) treeRef(It) treeRef(Its2))
                ++ It.viztree ++ Its2.viztree
It < ↑ viztree : [TreeStmt] >
It = Use String
  It.viztree = nodeRecord0 ("Use" ++ String) treeRef(It)
  | Decl String
  It.viztree = nodeRecord0 ("Decl" ++ String) treeRef(It)
  | Block Its
  It.viztree = (nodeRecord1 "Block" treeRef(It) treeRef(Its)) ++ Its.viztree

```

Fragment 1: Constructing the Visual Tree.

Where the function *treeRef* returns a unique identifier of its tree-value argument (the tree pointer).

Next, we declare a higher-order attribute, *i.e.*, attributable attribute (*ata*) named *visualTree*, in the context of the single production applied to the root non-terminal of the BLOCK AG. The type of the higher-order attribute is *TreeViz* which is the type of the root non-terminal of the reused component. After that, we have to instantiate the higher-order attribute with the attribute synthesized in the above fragment. Finally, and because *TreeViz* has no inherited attributes, we just have to access the synthesized attribute of the higher-order attribute, as usual. The HAG fragment looks has follows:

```

P < ↑ String : visualTree >
P = Root Its
  ata visualTree : TreeViz -- Declaration
  visualTree = CTreeViz "BlockTree" Its.viztree -- Instantiation
  P.visualTree = visualTree.graphviz -- Use of its syn. attrs

```

This fragment defines a higher-order extension to the BLOCK attribute grammar presented in the previous section. To process such higher-order attribute grammar, we use the LRC system: an incremental, purely functional higher-order attribute grammar based system [KS98]. Thus, we can use LRC to process the BLOCK HAG and to produce the desire BLOCK processor.

Figure 1 shows two different snapshots (displayed by GraphViz) of the tree that is obtained as the result of running the BLOCK processor with the input example sentence. As we can see the tree is collapsed into a minimal *Direct Acyclic Graphs* (DAG). This happens because we are using the incremental model of attribute evaluation of LRC³.

³ LRC achieves incremental evaluation through function memoization. Trees are arguments of the evaluators' functions. Thus, to make function memoization possible,

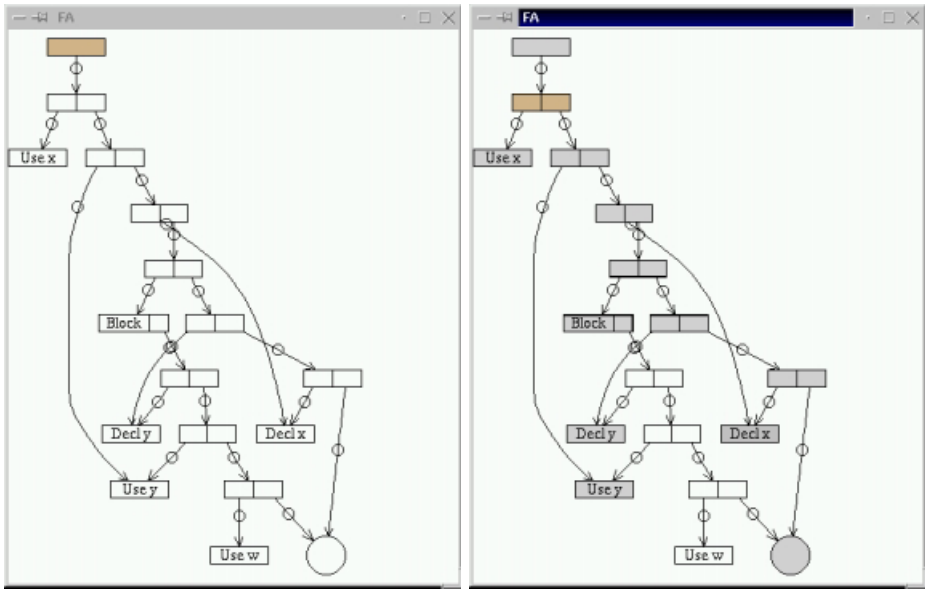


Fig. 1. The DAG representing the BLOCK example sentence at the beginning of the evaluation (left) and after completing the first traversal to the outermost block (right).

Besides computing the graphical representation of the tree, the processor generated by LRC also produces a sequence of node transitions. This is exactly the sequence of visits the evaluator performs to decorate the tree under consideration. Such sequence can be loaded in and animated in GraphViz, either in single step or in continuous mode, forwards and backwards. Furthermore, colors are used to mark the visited nodes.

The snapshot on the left shows the beginning of the evaluation: the root node is visited for the first time (the shadowed node). The snapshot on the right shows the end of the first traversal to the outermost block. Note that the nodes of the nested block were not visited (they are not shadowed). Indeed, the AG scheduler induced (as we expected) that only after collecting the complete environment of the outer block (performed on its first traversal), can the evaluator visit the inner ones. The inner blocks are traversed twice in the second traversal of the outer block.

3.2 An Attribute Grammar Component for Advanced Interactive Interfaces

As it was previously stated, types can be defined within the attribute grammar formalism. So, we may use this approach to introduce a type that defines an

they have to be efficiently compared for equality. Minimal DAG's allow for efficient equality tests between all terms because a pointer comparison suffices.

abstract representation of the interface of language-based tools. In other words, we use an abstract grammar to define an abstract interface. The productions of such a grammar represent “standard” graphical user interface objects, like menus, buttons, etc. Next, we present the so-called *abstract interface grammar*.

<i>Visuals</i> = CVisuals [<i>Toplevel</i>]	<i>Frame</i> = Label	<i>String</i>
	ListBox	<i>Entrylist</i>
	PullDownMenu	<i>String MenuList</i>
<i>Toplevel</i> = <i>Toplevel Frame String String</i>	PushButton	<i>String</i>
	Unparse	<i>Ptr</i>
	HList	[<i>Frame</i>]
	VList	[<i>Frame</i>]

The non-terminal *Visual* defines the type of the abstract interface of the tool: it is a list of *Toplevel* objects, that may be displayed in different windows. A *Toplevel* construct displays a frame in a window. It has three arguments: the frame, a name (for future references) and the window title. The productions applied to non-terminal *Frame* define concrete visual objects. For example, production *PushButton* represents a *push-button*, production *ListBox* represents a *list box*, etc.

The production *Unparse* represents a visual object that provides *structured text editing* [RT89]. It displays a pretty-printed version of its (tree) argument and allows the user to interact with it. Such beautified textual representation of the abstract syntax tree is produced according to the unparse rules specified in the grammar. It also allows the user to point to the textual representation to edit it (via the keyboard), or to transform it using user defined transformations. The productions *VList* and *HList* define combinators: they vertically and horizontally (respectively) combine visual objects into more complicated ones. These non-terminals and productions can be directly used in the attribute grammar to define the interface of the environments. Thus, the interface is specified through attribution, *i.e.*, within the AG formalism.

To define a concrete interface, we need, as we have said above, to define the mapping from the abstract interface representation into a concrete one. Instead of defining a concrete interface from scratch, we synthesize a concrete interface for a existing GUI toolkit, *e.g.*, the TCL/Tk GUI toolkit [Ous94]. Indeed, the GUI AG component synthesizes TCL/Tk code defining the interface in the attribute named *tk*.

Next, we present an attribute grammar fragment that glues the BLOCK HAG with this GUI AG component. It defines an interactive interface consisting of three visual objects that are vertically combined, namely: a push-button, the unparsing of the input under consideration and the unparsing of the list of errors. The root symbol *P* synthesizes the TCL/Tk concrete code in the attribute occurrence *concreteInterface*.

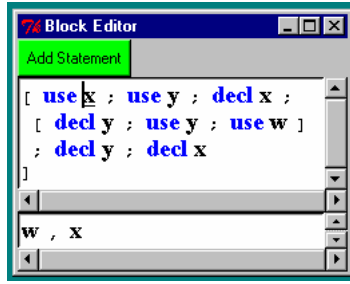


Fig. 2. The BLOCK environment's interface generated from the HAG.

```

P < ↑ concreteInterface : Tk >
P = Root Its
  ata absInterface : Visuals
    absInterface = let { button = PushButton "Add Statement"
                        editor = Unparse &P
                        errors = Unparse &P.errs
                        comb = VList [ button , editor , errors ]
                      } in [ Toplevel comb "edit" "Block Editor" ]
P.concreteInterface = absInterface.tk

```

Fragment 2: The BLOCK graphical user interface.

Figure 2 shows the concrete interface of the BLOCK processor.

The `PushButton` constructor simply displays a push-button. To assign an *action* to the displayed button we have to define such an action. Once again we use the same technique, *i.e.*, we define an abstract grammar to describe the abstract events handled by interactive interfaces. Basically, we associate an abstract event-handler to each visual object.

```

Event = ButtonPress String
      | ListBoxSelect Entrylist
      | MenuSelect String
      | TextKeyPress Char

```

The constructor `ButtonPress` is the event-handler associated with `PushButton`. Next, we show a possible action associated with this event-handler.

```

Its = NilIts
  bind on ButtonPress "Add Statement"
    : Its → ConsIts (Decl "a") NilIts;

```

The *bind* expression is used to specify how user interactions are handled by the language-based environment. In this case, it simply defines that every time the push-button "Add Statement" is pressed, the rooted subtree *Its* is transformed into `ConsIts Decl("a") NilIts`. Note that this event-handler constructor is defined in the context of a `NilIts` production. Thus, a new declaration is added at the end of the program being edited.

Other features of visualization and animation, and of the advanced graphical user interface AG components are:

- The use of abstract grammars (*i.e.*, intermediate representation languages) makes these components highly modular: new concrete visualizations/animations/interfaces can be “*plugged into*” the AG system, just by defining the corresponding mapping function.
- This approach has another important property: under an incremental attribute evaluation scheme, *the visualization/animation/interface is incrementally computed*, like any other attribute value [Sar99,SSK00].
- Because the LRC system uses an incremental computational model, we can animate incremental attribute evaluators. Indeed, in the animations produced by LRC, it is possible to visualize the reuse of a memoized function call: the animation simply changes the color of a node, without visiting its descendents.

4 Related Work

The work presented in this paper is closely related to attribute coupled grammars [GG84,LJPR93,CDPR98], composable attribute grammars [FMY92] and Kastens and Waite work on modularity and reusability of attribute grammars [KW94].

Attribute coupled grammars consist of a set of AG components each of which (conceptually) returns a tree-valued result that is the input for the next component. Grammars are coupled by defining attribute equations that build the required tree-valued attributes, very much like the values of higher-order attributes are defined in our approach (*e.g.*, Fragment 1). In attribute coupled grammars, however, the flow of data is strictly linear and unidirectional. In our approach the data can flow freely throughout the components, provided that no attribute depends directly nor indirectly on itself. Under our techniques such cyclic dependencies are statically detected.

In [GG84] *descriptive composition* is defined to eliminate the creation of the intermediate trees. That is, from the coupling attribute grammar (modules) a grammar is constructed that defines the same equations, but that eliminates the construction of the intermediate trees. The descriptive composition, however, can result in a non-absolute circular AG. Furthermore, descriptive composition does not allow the separate analysis and compilation of grammar components.

Composable attribute grammars [FMY92] use a particular grammar module for gluing AG components. Grammar modules can be analysed and compiled separately. However, the gluing of the components is expressed with a special notation outside the AG formalism.

Kastens and Waite [KW94] aim at a different form of modularity. They show that a combination of notational concepts can be used to create reusable attribute modules. They also define a set of modules to express common operation on programming languages. However, such modules are not defined within the

AG formalism, thus, making the maintenance, updating and understanding of such components much harder.

5 Conclusions

This paper presented techniques to write attribute grammars under a component-based style of programming. Such techniques rely entirely on the higher-order attribute grammar formalism: attribute grammar components are glued into a larger AG system through higher-order attributes. Standard attribute grammar techniques are used to detect circularities (*e.g.*, AG circularity test), to efficiently schedule the computations (*e.g.*, AG scheduling algorithms), and, to eliminate redundant intermediate data structures induced by higher-order attributes (*e.g.*, AG deforestation techniques).

We also have presented two generic, reusable and off-the-shelf AG components that can easily be “plugged into” any higher-order attribute grammar specification. Such components provide powerful properties to visualize, animate and interact with language-based tools. Thanks to the fact that these components are themselves defined in the HAG formalism, we inherit all of its nice properties and because of that the maintenance, updating and understanding of such components is simpler.

These components are implemented in the LRC system. However, they can be reused in any attribute grammar system, provided it processes higher-order attribute grammars.

References

- [CDPR98] Loic Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Generic Programming by Program Composition. In *Proceedings of the Workshop on Generic Programming*, pages 1–13, June 1998.
- [dMBS00] Oege de Moor, Kevin Backhouse, and Doaitse Swierstra. First-Class Attribute Grammars. In D. Parigot and M. Mernik, editors, *Third Workshop on Attribute Grammars and their Applications, WAGA'99*, pages 1–20, Ponte de Lima, Portugal, July 2000. INRIA Rocquencourt.
- [dMPJvW99] Oege de Moor, Simon Peyton-Jones, and Eric van Wyk. Aspect-Oriented Compilers. In *Proceedings of the First International Symposium on Generative and Component-Based Software Engineering (GCSE '99)*, LNCS, September 1999.
- [FMY92] Rodney Farrow, Thomas J. Marlowe, and Daniel M. Yellin. Composable Attribute Grammars: Support for Modularity in Translator Design and Implementation. In *19th ACM Symp. on Principles of Programming Languages*, pages 223–234, Albuquerque, NM, January 1992. ACM press.
- [GG84] Harald Ganzinger and Robert Giegerich. Attribute Coupled Grammars. In *ACM SIGPLAN '84 Symposium on Compiler Construction*, volume 19, pages 157–170, Montréal, June 1984.
- [GN99] Emden R. Gransner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software Practice and Experience*, 00(S1):1–29, 1999.

- [KS98] Matthijs Kuiper and João Saraiva. Lrc - A Generator for Incremental Language-Oriented Tools. In Kay Koskimies, editor, *7th International Conference on Compiler Construction, CC/ETAPS'98*, volume 1383 of *LNCS*, pages 298–301. Springer-Verlag, April 1998.
- [KW94] Uwe Kastens and William Waite. Modularity and reusability in attribute grammar. *Acta Informatica*, 31:601–627, June 1994.
- [Law01] Julia L. Lawall. Implementing Circularity Using Partial Evaluation. In *Proceedings of the Second Symposium on Programs as Data Objects PADO II*, volume 2053 of *LNCS*, May 2001.
- [LJPR93] Carole Le Bellec, Martin Jourdan, Didier Parigot, and Gilles Rous-sel. Specification and Implementation of Grammar Coupling Using Attribute Grammars. In Maurice Bruynooghe and Jaan Penjam, editors, *Programming Language Implementation and Logic Programming (PLILP '93)*, volume 714 of *LNCS*, pages 123–136, Tallinn, August 1993. Springer-Verlag.
- [Ous94] J.K. Ousterhout. *Tcl and the Tk toolkit*. Addison Wesley, 1994.
- [Pen94] Maarten Pennings. *Generating Incremental Evaluators*. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, November 1994.
`ftp://ftp.cs.uu.nl/pub/RUU/CS/phdtheses/Pennings/`.
- [RT89] T. Reps and T. Teitelbaum. *The Synthesizer Generator*. Springer, 1989.
- [Sar99] João Saraiva. *Purely Functional Implementation of Attribute Gram-mars*. PhD thesis, Department of Computer Science, Utrecht Univer-sity, The Netherlands, December 1999.
`ftp://ftp.cs.uu.nl/pub/RUU/CS/phdtheses/Saraiva/`.
- [SS99a] João Saraiva and Doaitse Swierstra. Data Structure Free Compila-tion. In Stefan Jähnichen, editor, *8th International Conference on Compiler Construction, CC/ETAPS'99*, volume 1575 of *LNCS*, pages 1–16. Springer-Verlag, March 1999.
- [SS99b] João Saraiva and Doaitse Swierstra. Generic Attribute Grammars. In D. Parigot and M. Mernik, editors, *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, pages 185–204, Amster-dam, The Netherlands, March 1999. INRIA Rocquencourt.
- [SSK97] João Saraiva, Doaitse Swierstra, and Matthijs Kuiper. Strictifica-tion of Computations on Trees. Technical report UU-CS-1997-30, Department of Computer Science, Utrecht University, August 1997.
`ftp://ftp.cs.ruu.nl/pub/RUU/CS/techreps/CS-1997/1997-30.ps.gz`.
- [SSK00] João Saraiva, Doaitse Swierstra, and Matthijs Kuiper. Functional Incre-mental Attribute Evaluation. In David Watt, editor, *9th International Conference on Compiler Construction, CC/ETAPS2000*, volume 1781 of *LNCS*, pages 279–294. Springer-Verlag, March 2000.
- [TC90] Tim Teitelbaum and Richard Chapman. Higher-order attribute gram-mars and editing environments. In *ACM SIGPLAN'90 Conference on Principles of Programming Languages*, volume 25, pages 197–208. ACM, June 1990.
- [VSK89] Harald Vogt, Doaitse Swierstra, and Matthijs Kuiper. Higher order attribute grammars. In *ACM SIGPLAN '89 Conference on Program-ming Language Design and Implementation*, volume 24, pages 131–145. ACM, July 1989.

Altering Java Semantics via Bytecode Manipulation

Éric Tanter¹, Marc Ségura-Devillechaise², Jacques Noyé², and José Piquier¹

¹ UNIVERSITY OF CHILE, COMPUTER SCIENCE DEPT.
Avenida Blanco Encalada 2120, Santiago, Chile,
{etanter,jpiquer}@dcc.uchile.cl

² ECOLE DES MINES DE NANTES, OCM GROUP
La Chantrerie, 4, rue Alfred Kastler. B.P. 20722,
F-44307 Nantes Cedex 3, France,
{msegura,noye}@emn.fr

Abstract. Altering the semantics of programs has become of major interest. This is due to the necessity of *adapting* existing software, for instance to achieve interoperability between off-the-shelf components. A system allowing such alterations should operate at the bytecode level in order to preserve portability and to be useful for pieces of software whose source code is not available. Furthermore, working at the bytecode level should be done while keeping high-level abstractions so that it can be useful to a wide audience. In this paper, we present Jinline, a tool that operates at load time through bytecode manipulation. Jinline makes it possible to inline a method body before, after, or instead of occurrences of language mechanisms within a method. It provides appropriate high-level abstractions for fine-grained alterations while offering a good expressive power and a great ease of use.

1 Introduction

Altering the semantics of programs serves many objectives in software engineering, related to software *adaptation*. A particular case of software adaptation, highlighted by Keller and Hölzle in [1], is to make several off-the-shelf components interoperable [2]. To this end, Keller and Hölzle proposed binary component adaptation (BCA), a tool for performing coarse-grained alterations on component binaries. However, coarse-grained alterations, usually limited to modifications of the interface or of the type hierarchy, may turn out to be insufficient. Another objective addressed by alteration of program semantics is that of separation of concerns [3], as emphasized by the work carried out within the reflection community [4,5,6], and more recently, by the emerging paradigm of aspect-oriented programming (AOP) [7]. In both cases, an important objective is to separate the development of the functional core of an application from the implementation of its non-functional concerns, such as persistency, distribution, or security. The complete application is then obtained by merging the different parts together. Such a merging requires to perform fine-grained alterations

within method bodies. The purpose of the work we present in this paper is to provide a tool enabling such alterations with the appropriate level of abstraction.

In Java, portable transformation mechanisms require code rewriting. This usually automated rewriting can be performed on source code or on bytecode. The Java community has already developed an impressive set of tools transforming source code: AspectJ [8] to support AOP, Sun's JavaScope project to instrument source code, a Dylan-like macro system called Java Syntactic Extender [9] and a class-based macro system, OpenJava [10]. Nevertheless, in many contexts, expecting source code availability is a mistake: off-the-shelf components usually ship in binary form, and sophisticated distributed systems, like mobile agent platforms, usually rely on dynamic class loading. Therefore, while still interesting in themselves, these tools are not generally applicable. This is why we claim that transformation tools should operate on bytecode.

Available transformation tools based on bytecode rewriting are usually inadequate for a wide and generic use. First, most of these tools offer bytecode-level abstractions. This is inadequate if the tool has to be used by a wide audience, since precise knowledge of the bytecode language is required. This point has been addressed by Javassist [11], which offers high-level abstractions. Though targeted to structural reflection, Javassist can be used to perform fine-grained alterations. However, in this domain, Javassist suffers from a limited expressive power and a lack of generality, as we will discuss in section 2.

In this perspective, we propose Jinline, a tool for altering Java semantics. Jinline operates on bytecode, keeps high-level abstractions, offers a good expressive power and generality. To summarize, Jinline makes it possible to inline a method body before, after, or instead of a language mechanism occurrence¹ within a method.

Traditionally, *inlining* means *replacing a call to a function by an instance of the function body* [12]. What Jinline actually does is inserting code or replacing code. The new code is defined by a method and therefore the inserted code is *conceptually* a method call, except that Jinline actually *inlines* this new method. Hence, although Jinline cannot be qualified as an inliner, most of its job consists of inlining pieces of code into others. In addition to this, Jinline provides two different sets of information:

1. **Static information at inlining time.** Jinline provides static information that can be used to drive the inlining process. For instance, in the case of a message send, it will provide the signature of the invoked method. This helps to decide whether inlining should occur or not, which method should be inlined and where (before, after, instead of).
2. **Dynamic information at run time.** Jinline ensures that the inlined method will receive as arguments *all the useful* dynamic information that

¹ By *language mechanisms* we refer to the standard mechanisms offered by the language, such as message sending, accessing fields, casting, etc. A *language mechanism occurrence* is a particular instance of a language mechanism in a piece of code.

can be extracted. This point is very important since it makes the tool particularly suited for implementing generic extensions, as we will exemplify in the rest of this paper. In the case of a message send, the dynamic information includes the method invoked, the method from which the invocation is done, references to the caller and the callee, in addition to the actual arguments of the invocation.

Applications of such an alteration tool are manifold. We have already mentioned the issue of off-the-shelf components integration. Two of the authors are actually working on an open implementation of a run-time MetaObject Protocol (MOP), Reflex [13]. Many transformers for the Reflex framework can be implemented with Jinline, thus increasing its expressiveness with caller-side interceptions. Jinline is also particularly adapted for implementing custom extensions and AOP systems.

The rest of this paper is organized as follows: in section 2, we will review the different Java bytecode manipulation tools and relate our work to them. In section 3 we will present Jinline, its interface to the outside world and an overview of its architecture. In section 4 we will present a simple example of applying Jinline. Section 5 will conclude the paper.

2 An Overview of Bytecode Manipulation Tools

One way of modifying a program is to alter its semantics by using reflection [14, 15]. However, the Java programming language does not provide support for altering the semantics of programs. Since the class model is closed (class `Class` and all the classes of the Reflection API are final), it is not possible to refine the semantics of language mechanisms by specializing the class model, as can be done in Smalltalk [16]. Therefore, alterations have to be implemented either at the virtual machine level, like in VM-based run-time metaobject protocols like Metaxa [17], Guaraná [18] and Iguana/J [19] thus sacrificing portability, or at the code level, through code transformation. We have already discarded the possibility of operating on source code for reasons of availability of the source code itself. This is why a number of propositions have been made to transform bytecode. These propositions differ in terms of the abstraction level of the entities a user is expected to program with, and in the expressive power or granularity of the transformations permitted.

2.1 Transformations Based on Bytecode-Level Abstractions

A number of extensions allow programmers to transform classes at load time at the expense of manipulating abstractions representing bytecode.

BIT [20] suffers from a too restricted scope: it only offers the possibility to insert before/after methods, but does not address transformation of interfaces or method bodies.

There are several general-purpose implementations of bytecode manipulation available: BCEL [21], JikesBT [22], and JOIE [23]. All of them translate the class file data structure into an intermediate representation, allow the user to perform modifications and to finally regenerate a valid class file data structure from the transformed intermediate representation. The bytecode-level API of Javassist [11] could fit into this category although bytecode instructions are not reified: the programmer is just provided with an iterator over a sequence of bytes. The main strength of these general-purpose extensions is their expressive power, since they are able to express anything that can be written in bytecode. However, their main drawback is to be low-level and therefore difficult to use.

2.2 Transformations Based on Source-Level Abstractions

Metaobject protocols (MOPs) are a natural framework for reifying high-level language entities [24]. Run-time MOPs are an approach to enable the run-time alteration of program semantics. Compared to static transformation systems – such as macro systems, inlining systems, and compile-time MOPs –, where the link between the modifier and the modified entity is merged at some point, run-time MOPs maintain this link, known as the *causal connection link* [14,15], at run time, thus enabling dynamic updates of this link at the expense of a certain overhead.

Reflex [13] and Kava [25] are run-time MOPs for Java that rely on load-time insertion of pieces of code (hooks) to transfer control to the metalevel at run time. These systems are bound to behavioral reflection, which is the ability of dynamically altering the behavior of objects. This approach is in fact complementary to static code transformation approaches in cases where dynamic adaptability or instance-specific alterations are needed (see for instance [26]).

BCA [1] is a bytecode modification tool with a high-level interface, but it only deals with external interfaces and class hierarchies, ignoring method bodies. Javassist [11] is a mature tool for load-time structural reflection in Java. Structural reflection is the ability of a program to alter the definitions of data structures such as classes and methods. With Javassist, the transformations that can be made are at the granularity of class or members. The main goal achieved by Javassist is a high-level and easy-to-use interface. To allow finer-grained transformations, Javassist has recently made public its bytecode-level API, which we mentioned in subsection 2.1. Recall that it lacks a concrete reification of bytecode instructions. To bridge the gap between its high-level and low-level APIs, Javassist offers a *code converter* to instrument method bodies through a high-level interface.

2.3 Limitations of the Code Converter of Javassist

The code converter of Javassist – the closest tool to our proposal – offers a simple high-level API to alter method bodies. This API allows inserting before/after methods, redirecting method invocations or field accesses, and replacing creations. We claim that its expressiveness is limited and that it lacks generality.

Its limited expressiveness is in fact not that much an issue since it can actually be upgraded, and also, in many cases, it is sufficient to alter such mechanisms as method invocations, field accesses and object creations. A more annoying problem is the limitation about the possible transformations: for instance, a field access can *only* be replaced by a static method call, and a method invocation can *only* be replaced by another method invocation on the *same* object with the *same* parameters.

But all in all, the major drawback of the code converter lies in the fact that it is not well-suited to designing generic solutions. Since Javassist lacks semantic information in the process of modifying bytecode (remember that Javassist does not reify bytecode instructions as such), the possible transformations are limited. The code converter does not perform any reification of *what* is actually occurring. For instance, an object creation can be replaced by a method invocation, but this method will not receive as argument the name of the class that was to be instantiated: it has to be *specific* to a type. This limitation is common to all transformations.

To illustrate this limitation, consider the following simple example: we want to set up a factory pattern [27] for instantiating any class in an existing application. That is to say, instead of calling directly `new`, we want to call a factory method. Designed with generality and extensibility in mind, the factory method would be:

```
public Object getInstance(String classname, Object[] args){...}
```

Then we want to transform all the instantiations so that they call this *unique* factory method, for instance:

```
new Point(1,2);  ⇒  Factory.getInstance("Point", [1,2]);
```

This is not feasible with the code converter. The only possible replacement is:

```
new Point(1,2);  ⇒  Factory.getPoint(1,2);
```

The following issues come to light:

- First, the name of the instantiated class is not passed as a parameter, which implies that we need a method per class (a `getPoint` method, a `getTriangle` method, etc.).
- Second, the arguments are not packed, which means we need a method per set of parameters (a method `getPoint(int, int)`, another method `getPoint(Point)`, etc.).

It is easy to see that such an approach is not applicable to real world cases. What is needed is a tool that can systematically provide runtime information in a cost-effective manner to the new inserted code. In addition to this, more flexibility with respect to what code can be inserted is highly appreciable. This is exactly what Jinline is about.

3 Jinline

Jinline is a Java tool for altering the semantics of programs via bytecode manipulation. Jinline makes it possible to inline a piece of code before, after, or instead of occurrences of language mechanisms. Jinline provides all the necessary static information to drive the inlining process and, which is really important, it wraps at runtime all the available dynamic information and passes it to the inlined code. This makes generic alterations possible, unlike the `CodeConverter` of Javassist. We will show in section 4 how simple it is to solve with Jinline the issue presented in section 2.3.

Unlike low-level bytecode translators, Jinline keeps high-level abstraction while offering a good expressive power. This makes it usable by a wide audience and applicable to cases where source code is not available (e.g., components, mobile code systems). Inlining can be triggered on language-level mechanisms (message send, constructor send, cast, ...) and the inlined code is also expressed in source code.

Jinline is integrated within the Javassist framework for load-time bytecode transformation which was designed with type safety² and correctness³ in mind. Types and methods reifications in Jinline are those of the core Javassist API.

3.1 A User View on Jinline

Inlined code. A natural formalism for representing a piece of code to inline is indeed a *method*. The programmer is responsible for writing it in Java and compiling it using a standard compiler. It will then need to feed the inliner with the bytecode definition of this method. In addition to this, choosing to inline methods provides us with a natural way to pass dynamic information at run time to the inlined piece of code: all relevant information is packed and passed as argument of the inlined method. We will describe the structure of this information later in this section.

When a method is inlined, it is not recursively processed by Jinline. This is to avoid inconsistencies: if one wants to inline a method for method invocations and another method for casts, Jinline will actually perform both transformations simultaneously. Therefore the order of the transformations does not have any impact on the result: none of the inlined methods will be transformed. Nevertheless, it is not impossible to imagine cases where one would like to process the inlined code. This can be done by completing the first process and then performing a second pass over the transformed code.

Descriptions. All the language mechanisms are represented in Jinline by *description classes*. There is a description class for message send, one for cast, etc. An instance of such a class contains all the static information that describes the occurrence of the mechanism. The methods of each class are the accessors to

² You cannot get an invalid reference to a reification of a class or method.

³ Javassist greatly limits the possibility of producing bytecode rejected by the JVM.

the static information that is provided to the user for driving the inlining process. Figure 1 shows the hierarchy of description classes in Jinline. Jinline does not cover message reception, since it can typically be implemented by coarser-grained transformations (as done in the reflection package of Javassist). Also, Jinline does not cover control structures, since they are not always represented the same way in the bytecode that in the source code: some compiler optimizations may eliminate or alter control structures existing in the source code.

One can refer to a language mechanism by using its description class object. For instance, `MessageSend.class` represents the language mechanism of sending messages, while an instance of this class represents an occurrence of this mechanism.

The process. The inlining process is based on a listener-notifier schema as illustrated in Figure 2. In Jinline, method definitions are the unit of transformations. When parsing a method to alter, Jinline notifies a listener each time it encounters an occurrence of interest. The notification embodies all the static information that can be extracted about the occurrence. The listener can then analyze this static information and decide to inline or not a piece of code before, after, or instead of the occurrence.

An inlining process is represented by an instance of the `Jinliner` class. The inlining process is specified by attaching the proper listener to each language mechanism of interest. For instance, to specify that the `Jinliner` object should send a notification upon occurrences of message send, one should write:

```
jinlineer.notifyUpon(MessageSend.class, aJinler);
```

One of the advantage of such an approach is that it is possible to specify a common super type of mechanism in the specification. For instance:

```
jinlineer.notifyUpon(Description.class, aJinler);
```

implies that `aJinler` will be notified of any occurrence of a known mechanism.

To instrument a method, this method first needs to be reified into a Javassist `CtMethod` object:

```
CtMethod myMethod = aCtClass.getDeclaredMethod("myMethod");
```

Then the process can be started:

```
inliner.process(myMethod);
```

It is also possible to process an entire class (by passing a `CtClass` object to `process`) in which case, by default, all its public methods and constructors will be processed.

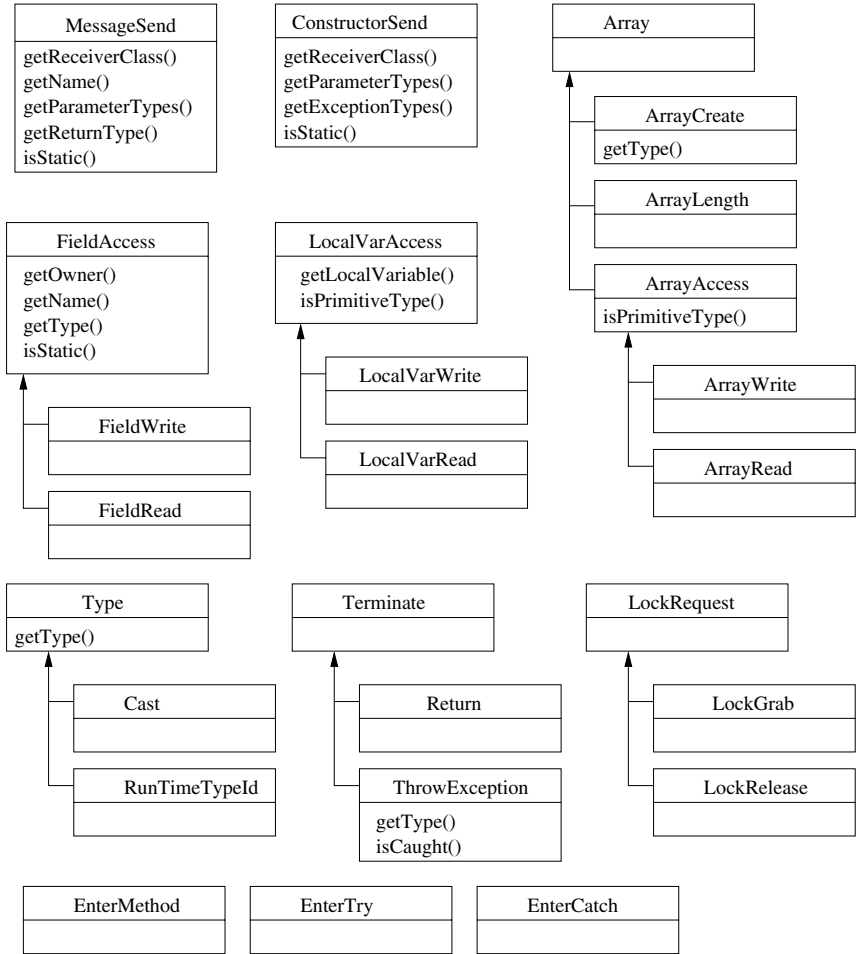


Fig. 1. The hierarchy of description classes. Note that the common superclass `Description` has been omitted for the sake of readability.

Jinlers and static information. Jinlers are objects that listen for notification from the inliner and that drive the inlining process. Such objects implement the `Jinler` interface, which declares the method:

```
public ToInline notify(Description desc, Context context);
```

This means that upon notification, the `Jinler` object receives an instance of a description class, which encapsulates all the static information about the occurrence of the mechanism. For instance, to reason about the return type of a message send, one can use:

```
if(((MessageSend)desc).getReturnType().equals(...)) ...
```

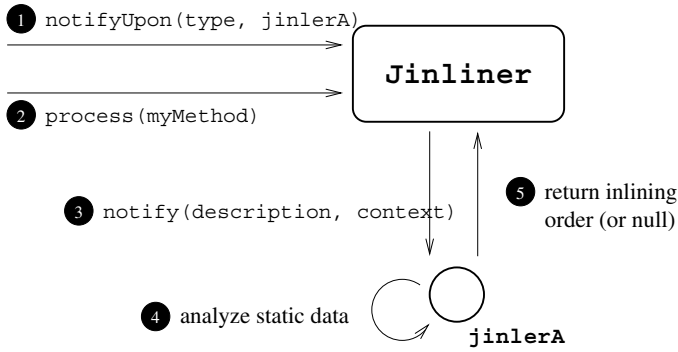


Fig. 2. The inlining process is based on a notification scheme. (1) *jinlerA* is registered for notification of occurrences of *type*. (2) The *Jinliner* is given the method to process. (3) Each time an occurrence of *type* is encountered, the *Jinliner* notifies *jinlerA*. (4) *jinlerA* analyzes the static data about the occurrence. (5) It returns an inline order or null if no inlining should take place.

The **Context** object given upon notification is simply an object that encapsulates the current class and the current method being processed.

Finally, to specify the desired inlining (that is, which method(s) to inline and where), a *Jinler* returns a **ToInline** object. Such an object is a structure of **CtMethod** objects to inline before, after, or instead of the occurrence. For instance:

```
return new ToInline(myMethod);
```

specifies that the inliner should inline `myMethod` *instead of* the current occurrence, whereas:

```
(1) return new ToInline(beforeMethod, afterMethod);
(2) return new ToInline(beforeMethod, null);
```

(1) specifies that `beforeMethod` should be inlined *before* the occurrence and that `afterMethod` should be inlined *after* it, and (2) specifies that only `beforeMethod` should be inlined *before* the occurrence.

Dynamic information and inlinable methods. At run time, dynamic information is packed and passed to the inlined method. Great attention has been paid to provide to the inlined method all the information available. The information is passed as an array of objects containing only standard objects and arrays, in order to be cost-effective. Table 1 shows how the information is ordered in the given array depending on the mechanism.

The way in which dynamic information is passed imposes a compatibility rule over inlinable methods. In *Jinline*, to be inlinable, a method has to accept either an array of objects as unique parameter or no parameter at all. If the

Table 1. Information delivered at runtime to the inlined method. The information is passed as an `Object[]`. For all mechanisms, the first two elements are the same: the method in which the occurrence is located and the instance that is involved (*this*). The table shows how specific information is stored in the array for each language mechanism. (*) Due to restrictions in the JVM specifications, only before/after insertion is possible.

Language mechanism	Dynamic information passed to the inlined method	Index: type	Expected return value
<i>Common</i>	qualified name of altered method 0: <code>String</code> <i>this</i> object 1: <code>Object</code>		
Message sent	qualified name of invoked method 2: <code>String</code> target instance 3: <code>Object</code> values of arguments 4: <code>Object[]</code>		invocation result
Constructor	qualified name of target class 2: <code>String</code> argument values 3: <code>Object[]</code>		object
Cast and RTTI	qualified name of target type 2: <code>String</code> target instance 3: <code>Object</code>		object (cast) or boolean (RTTI)
Field read	qualified name of field read 2: <code>String</code> target instance 3: <code>Object</code>		field value
Field write	qualified name of field written 2: <code>String</code> target instance 3: <code>Object</code> new field value 4: <code>Object</code>		void
Local variable read	is this 2: <code>Boolean</code> is parameter 3: <code>Boolean</code> is primitive type 4: <code>Boolean</code>		variable value
Local variable write	is this 2: <code>Boolean</code> is parameter 3: <code>Boolean</code> is primitive type 4: <code>Boolean</code> new variable value 5: <code>Object</code>		void
Array creation	qualified name of array type 2: <code>String</code>		array
Array length	array whose length is being accessed 2: <code>Object[]</code>		array length
Array read	array which is being accessed 2: <code>Object[]</code> index of the array element 3: <code>Integer</code>		object
Array write	array which is being accessed 2: <code>Object[]</code> index of the array element 3: <code>Integer</code> new array element value 4: <code>Object</code>		void
Lock grab (*)	target instance 2: <code>Object</code>		void
Lock release (*)	target instance 2: <code>Object</code>		void
Throw exception	exception instance 2: <code>Throwable</code>		exception to throw
Return	value that is to be returned 2: <code>Object</code>		object to return
Enter a method or a try/catch	<i>none</i>		void

method does not accept any parameter, no dynamic information will be passed (which is much more efficient when dynamic information is not needed). This entails that when creating a `ToInline` object, the `CtMethod` objects given to the

constructor must fulfill this compatibility requirement otherwise an exception is thrown.

As far as return values are concerned, no check is made when creating a `ToInline` object. In case of inlining before/after, the return value is simply not taken into account since it is irrelevant. In the case of a replacement, it is up to the programmer to guarantee that the value returned by the inlined method is compatible with the expected type (see table 1). `Jinline` only takes care of wrapping and unwrapping primitive types and exceptions, which is actually the only thing it can do systematically. At run time, if an inlined method returns an incompatible value, a `ClassCastException` is thrown.

3.2 Overview of Implementation

We have created our own reification of bytecode instructions and created a high-level symbolic object for bytecode modification: `BytecodeSequence`. Such objects represent sequences of bytecodes and offer services to manipulate them, in particular inserting another sequence at a given index. Since our main goal is easy insertion of sequences into others, our reification of bytecode is a straightforward one. We are not interested in optimizing the bytecode or performing analyses on it. This is why we do not change the bytecode format, as is done in a dedicated framework like Soot [28].

A `BytecodeSequence` object can be created out of Javassist `MethodInfo` objects, which are the low-level reifications of methods. During parsing, any information (indexes, jumps, local variables, etc.) is translated into symbolic data. Manipulation over bytecode sequences is all symbolic. This makes copies of method bodies into others fairly straightforward. It is only at generation time that the symbolic information is translated back to raw data. The `MethodInfo` object is then updated with this new raw data.

For the inlining part, a `MethodParser` is responsible for parsing a method body and notifying the appropriate Jinlers whenever needed. When an inlining needs to be done, it is delegated to a `MethodInliner`.

A `MethodInliner` is responsible for inlining the method in a semantically correct manner:

- if the inlined method expects dynamic information, it first inserts a prologue that does the wrapping of all the parameters. Such a prologue is specific to each supported mechanism. The array of objects that the inlined method expects as an argument is transmitted via an extra local variable;
- it adjusts the method to inline to remove its return statements and ensure stack consistency. This modification is common to all mechanisms. Note that the value returned by the inlined method is also transmitted via a local variable;
- it appends the adjusted method body after the prologue;
- it adds an epilogue that unwraps the return value and manages the exceptions that the inlined code may throw. More precisely, if the inlined method throws an exception that was expected by the original code, it is simply

thrown again, otherwise an `UndeclaredThrowableException` is thrown, as is done in the Dynamic Proxy API of Java [29]. Such an epilogue is common to all mechanisms as far as the return value is concerned, but the exception handling part is specific to each mechanism. Recall that in the case of a before/after inlining, the return value is simply ignored.

Figure 3 below illustrates the inlining operation in the case of a replacement.

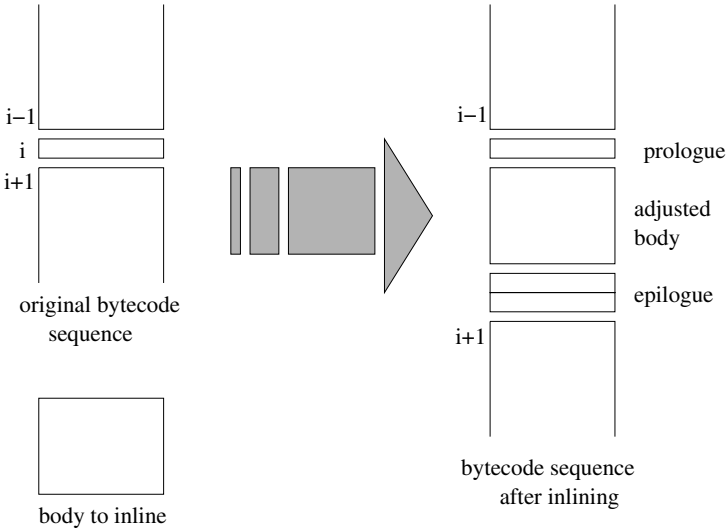


Fig. 3. The inlining operation at the bytecode level (replacement of `i`). The method body to inline is adjusted before insertion. The prologue wraps the arguments to the inlined body (if needed). The epilogue unwraps the return value (if needed) and handles exceptions.

4 Example

Let us come back to the example presented in section 2.3 when highlighting the limitations of the `CodeConverter` of Javassist. This section clearly shows how generic solutions can be implemented easily thanks to Jinline's ability to provide dynamic information. The objective is to use Jinline to replace any instantiation statement by a call to a generic factory method. We therefore assume that we have developed the factory:

```
public class Factory {
    public Object getInstance(String classname, Object[] args)
    { // factory code }
}
```

We now present the 3-step process used to solve the problem with Jinline.

1. Code to inline. The first step is to write a method that contains the code to inline. This is done by defining a sample class:

```
public class FactorySample {
    public Object newMethod(Object[] jinArgs){
(1)   String classname = (String) jinArgs[2];
(2)   Object[] args = (Object[]) jinArgs[3];
(3)   return Factory.getInstance(classname, args);
    }
}
```

Since in this example we make use of the dynamic information that Jinline provides to the inlined code, the method we will inline, `newMethod`, accepts as unique argument an array of objects. According to the way dynamic information is structured (see table 1), we know that the third argument in the array is the name of the class and that the fourth argument is the array of arguments to the constructor. Hence, we first retrieve those values (1 & 2). Then, we simply invoke the factory method with the extracted values (3).

This sample class has to be compiled so that it is possible to obtain a Javassist reification of the method `newMethod`.

2. The Jinler. Next, one needs to define the Jinler, that is to say, the entity responsible for driving the inlining process:

```
public class FactoryJinler implements Jinler {
    CtMethod newMethod;

    FactoryJinler(){
(1)   newMethod =
        ClassPool.getDefault().get("FactorySample")
                                .getDeclaredMethod("newMethod");
    }

    public ToInline notify(Description desc, Context cont){
(2)   if(desc instanceof ConstructorSend)
(3)       return new ToInline(newMethod);
(4)   return null;
    }
```

In the constructor, the Javassist reification of `newMethod` is obtained (1). In `notify`, the description object is filtered according to its type (2). If it corresponds to an instantiation, then we return an inline order, specifying that we want to inline `newMethod` instead of the current instantiation (3). Otherwise, we specify that no inlining should occur (4).

3. Connecting to the Javassist framework. The last step is to connect to the Javassist framework for load-time bytecode transformation. For the readers which are not familiar with Javassist, let us recall that it is possible to define a **Translator** object that is notified each time a class is loaded and that can perform some transformation before the class is actually loaded. Here is a translator for the Javassist framework that uses a **Jinliner** and the **Jinler** defined above:

```
public class FactoryTranslator implements javassist.Translator {
    Jinliner inliner = new Jinliner();
    Jinler jinler = new FactoryJinler();

    public void start(ClassPool pool){
(1)  inliner.notifyUpon(ConstructorSend.class, jinler);
    }

    public void onWrite(ClassPool pool, String classname){
(2)  CtClass clazz = pool.get(classname);
(3)  inliner.process(clazz);
    }
}
```

When connected to the framework, the translator is initialized by a call to its **start** method. The initialization work in this case simply consists of telling the **Jinliner** that it should notify the **Jinler** upon occurrences of constructor sends (1). Then, each time a class is about to be loaded, the translator is informed by an invocation of its **onWrite** method. Here, we simply get a reification of the class (2), and pass it to the **Jinliner** (3).

With this straightforward code, we are able, at load time, to systematically transform any occurrence of instantiation statements by the appropriate call to our generic factory.

5 Conclusion

In this paper we have presented **Jinline**, a tool for altering Java semantics via bytecode manipulation at load time. **Jinline** allows fine-grained alterations on bytecode while keeping high-level abstractions. With **Jinline**, a method body can be inlined before, after, or instead of any language mechanism occurrence within a method. Static information is given to the programmer to drive the inlining process and dynamic information can be passed to the inlined method, making generic alterations possible.

The main achievement of **Jinline** is to be a simple and powerful tool, with a wide set of possible applications, that seamlessly fits within the Javassist framework to extend the range of easily accessible bytecode manipulation in Java.

With regards to future work, we plan to perform some benchmarks and focus on optimizing the bytecode generated by **Jinline**, since code explosion might become an issue in cases where many alterations are performed within the same

method. Possible tracks are the use of subroutines to factor out common parts and basic optimizations on the generated bytecode sequences. As for applications, Jinline will be applied to build a comprehensive library of transformers for the Reflex framework, an open implementation of a run-time MOP [13].

Acknowledgments. We would like to deeply thank Shigeru Chiba for the highly valuable remarks he made on the first version of this paper. We are also grateful to the anonymous reviewers for their comments.

This work was partially funded by Millenium Nucleous Center for Web Research, Grant P01-029-F, Mideplan, Chile.

References

- [1] Keller, R., Hölzle, U.: Binary component adaptation. In: Proceedings of ECOOP'98 - 12th European Conference on Object-Oriented Programming, Brussels, Belgium, Springer-Verlag (1998) 307–29
- [2] Wegner, P.: Interoperability. *ACM Computing Surveys* **28** (1) (1996)
- [3] Tarr, P.L., Ossher, H., Harrison, W.H., Jr., S.M.S.: N degrees of separation: Multi-dimensional separation of concerns. In: International Conference on Software Engineering. (1999) 107–119
- [4] Stroud, R.J., Wu, Z.: Using Metaobject Protocols to Satisfy Non-Functional Requirements. In: Advances in Object-Oriented Metalevel Architectures and Reflection. CRC Press (1996) 31–52
- [5] Tanter, E., Piquer, J.: Managing references upon object migration: applying separation of concerns. In: Proceedings of the XXI International Conference of the Chilean Computer Science Society (SCCC 2001), Punta Arenas, Chile, IEEE Computer Society (2001) 264–272
- [6] McAffer, J.: Meta-level architecture support for distributed objects. In: International Workshop on Object-Oriented in Operating Systems (IWOOS'95). (1995)
- [7] Kiczales, G., Irwin, J., Lamping, J., Loingtier, J., Lopes, C., Maeda, C., Mendhekar, A.: Aspect Oriented Programming. In: Special Issues in Object-Oriented Programming, Max Muehlhaeuser (general editor) et al. (1996)
- [8] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: An overview of AspectJ. *Proceedings of ECOOP 2001* (2001)
- [9] Bachrach, J., Playford, K.: The Java syntactic extender. *OOPSLA 2001 conference proceedings* (2001) 31–42
- [10] Tatsubori, M.: An extension mechanism for the Java language. Master's thesis, Tsukuba, Japan (1999)
- [11] Chiba, S.: Load-time structural reflection in Java. *European Conference on Object-Oriented Programming (ECOOP'00)* (2000)
- [12] Howe, D.: FOLDLOC: Free On-Line Dictionary Of Computing. (1993) <http://foldoc.doc.ic.ac.uk>.
- [13] Tanter, E., Bouraqadi, N., Noyé, J.: Reflex – Towards an Open Reflective Extension of Java. In: Proceedings of the Third International Conference on Metalevel Architectures and Advanced Separation of Concerns (Reflection 2001). Volume 2192 of Lecture Notes in Computer Science., Kyoto, Japan, Springer-Verlag (2001) 25–43

- [14] Smith, B.: Reflection and semantics in Lisp. In: Proceedings of the 14th Annual ACM Symposium on principles of programming languages, POPL'84. (1984) 23–25
- [15] Maes, P.: Computational reflection. PhD thesis, Artificial intelligence laboratory, Vrije Universiteit, Brussels, Belgium (1987)
- [16] Goldberg, A., Robson, D.: Smalltalk-80: The Language and its Implementation. Addison-Wesley (1983)
- [17] Golm, M.: Design and implementation of a meta architecture for Java. Master's thesis, Leipzig Germany (1997)
- [18] Oliva, A., Calciolari Garcia, I., Buzato, L.: The reflexive architecture of Guaraná. Technical report, IC-98-14, Institute of Computing, State University of Campinas (1998)
- [19] Redmond, B., Cahill, V.: Supporting Unanticipated Dynamic Adaptation of Application Behavior. In: Proceedings of ECOOP 2002. Volume 2374 of Lecture Notes in Computer Science., Málaga, Spain, Springer-Verlag (2002) 205–230
- [20] Lee, H.B., Zorn, B.G.: BIT: A tool for instrumenting Java bytecodes. In: USENIX Symposium on Internet Technologies and Systems. (1997)
- [21] Dahm, M.: Byte code engineering. In Cap, C., ed.: Proceedings of JIT'99, Berlin. (1999) 267–277
- [22] AlphaWorks: JikesBT. <http://www.alphaworks.ibm.com/tech/jikesbt> (1998)
- [23] Cohen, G., Chase, J., Kaminsky, D.: Automatic program transformation with JOIE. in Proceedings of the 1998 USENIX Annual Technical Symposium (1998)
- [24] Kiczales, G., Des Rivières, J., Bobrow, D.: The Art of the Meta-Object Protocol. MIT Press (1991)
- [25] Welch, I., Stroud, R.: Kava - a reflective Java based on bytecode rewriting. In: 1st OOPSLA Workshop on Reflection and Software Engineering (OORaSE'99). Volume 1826 of Lecture Notes in Computer Science., Denver, USA, Springer-Verlag (2000) 165–167
- [26] Tanter, E., Vernailien, M., Piquer, J.: Towards Transparent Adaptation of Migration Policies. In: 8th ECOOP Workshop on Mobile Object Systems (EWMOS 2002), Málaga, Spain (2002)
- [27] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Professional Computing Series. Addison-Wesley (1994)
- [28] Vallee-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a Java optimization framework. In: Proceedings of CASCON 1999. (1999) 125–135
- [29] SUN Microsystems: Dynamic Proxy Classes. (1999)
<http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html>.

Meta-programming with Concrete Object Syntax

Eelco Visser

Institute of Information and Computing Sciences, Universiteit Utrecht, P.O. Box 80089, 3508 TB Utrecht, The Netherlands. <http://www.cs.uu.nl/~visser>, visser@acm.org

Abstract. Meta programs manipulate structured representations, i.e., abstract syntax trees, of programs. The conceptual distance between the concrete syntax meta-programmers use to reason about programs and the notation for abstract syntax manipulation provided by general purpose (meta-) programming languages is too great for many applications. In this paper it is shown how the syntax definition formalism SDF can be employed to fit *any* meta-programming language with concrete syntax notation for composing and analyzing object programs. As a case study, the addition of concrete syntax to the program transformation language Stratego is presented. The approach is then generalized to arbitrary meta-languages.

1 Introduction

Meta-programs analyze, generate, and transform object programs. In this process object programs are structured data. It is common practice to use abstract syntax trees rather than the textual representation of programs [10]. Abstract syntax trees are represented using the data structuring facilities of the meta-language: records (structs) in imperative languages (C), objects in object-oriented languages (C++, Java), algebraic data types in functional languages (ML, Haskell), and terms in term rewriting systems (Stratego).

Such representations allow the full capabilities of the meta-language to be applied in the implementation of meta-programs. In particular, when working with high-level languages that support symbolic manipulation by means of pattern matching (e.g., ML, Haskell) it is easy to compose and decompose abstract syntax trees. For meta-programs such as compilers, programming with abstract syntax is adequate; only small fragments, i.e., a few constructors per pattern, are manipulated at a time. Often, object programs are reduced to a core language that only contains the essential constructs. The abstract syntax can then be used as an intermediate language, such that multiple languages can be expressed in it, and meta-programs can be reused for several source languages.

However, there are many applications of meta-programming in which the use of abstract syntax is not satisfactory since the conceptual distance between the

concrete programs that we understand and the data structure access operations used for composition and decomposition of abstract syntax trees is too large. This is evident in the case of record manipulation in C, where the construction and deconstruction of patterns of more than a couple of constructors becomes unreadable. But even in languages that support pattern matching on algebraic data types, the construction of large code fragments in a program generator can become painful. For example, even the following tiny program pattern is easier to read in the concrete variant on the left than the abstract variant on the right.

```
let ds
  in let var x ta := (es1)
      in es2 end
end
```

```
Let(ds,
    [Let([VarDec(x,ta,Seq(es1))],
          es2)])
```

While abstract syntax is manageable for fragments of this size (and sometimes even more concise!), it becomes unpleasant to use when larger fragments need to be specified.

Besides the problems of understandability and complexity, there are other reasons why the use of abstract syntax may be undesirable. Desugaring to a core language is not always possible. For example, in the renovation of legacy code the goal is to repair the bugs in a program, but leave it intact otherwise. This entails that a much larger abstract syntax needs to be dealt with. Another occasion that calls for the use of concrete syntax is the definition of transformation or generation rules by users (programmers) rather than by compiler writers (meta-programmers). For example, [18] describes the extension of Haskell with pragmas for domain-specific optimization in the form of rewrite rules on program expressions. Other application areas that require concrete syntax are application generation and structured document (XML) processing.

Hence, it is desirable to have a meta-programming language that lets us write object-program fragments in the concrete syntax of the object language. In general, we would like to write a meta-program in meta-language M that manipulates a program in object language O , where M and O could be the same, but need not be in general. This requires the extension of the syntax of M with the syntax of O such that O expressions are interpreted as data construction and analysis patterns. This problem is traditionally approached by extending M with a quotation operator that lets the meta-programmer indicate object language fragments [1,17,23]. Antiquotation allows the use of meta-programming language constructs in these object language fragments to splice meta-computed object code into a fragment. If M equals O then the syntax extension is easy by just adding quote and antiquote operators to M . For example, MetaML [19], provides $\langle \dots \rangle$ for distinguishing a piece of object code and $\sim \dots$ to splice computed code into another piece of code. Several meta-languages, including ASF+SDF [16] and TXL [12], are designed around the idea of meta-programming with concrete object syntax, where the object language is user-definable.

Building a meta-language that supports a different object language is a difficult task with traditional parser technology. Building a meta-language that

allows the user to define the object language to use and determine the syntax for quotation and antiquotation operators is even harder. Why is this a difficult task? Traditional parser generators support only context-free grammars that are restricted to the LL or LR properties. Since these classes of grammars are not closed under composition, it is hard to extend one language with another. For example, combining two YACC grammars will usually lead to many conflicts, requiring editing the two grammars, probably extensively. The problem is compounded by the fact that lexical syntax is dealt with using a scanner, which operates separately from the parser. An embedded object language will almost certainly overlap at the lexical level with the host language (e.g., syntax of identifiers, literals, keywords). Thus, combining the two regular grammars will also require extensive editing.

The usual solution is to require a fixed syntax for quotation delimiters and parse the content of quotations in a separate pass. This requires quite some infrastructure and makes reporting syntax errors to the programmer difficult. The technology used to extend a meta-language is usually not portable to other languages.

In this paper we show how the syntax definition formalism SDF [14,20] can be employed to fit *any* existing (meta-) programming language with concrete syntax notation for object programs. *The approach does not require that either the meta-language or the object-language were designed with this application in mind.* Rather, the syntax definitions of the meta-language and object-language are combined as they are and wedded by providing appropriate injections from object language sorts into meta-language sorts. From the combined syntax definition a parser is generated that parses the entire meta-program including object code fragments, and thus reports syntactic errors immediately. An explosion algorithm that can be independent of the object language then maps embedded object code abstract syntax trees to appropriate meta-language representations. The approach is based on existing technology that is freely available and can be applied immediately.

We illustrate the approach by extending the strategic rewriting language Stratego [21,22] with concrete syntax. In Section 2 we motivate the need for meta-programming with concrete syntax by means of an example and contrast it to the use of abstract syntax. In Section 3 we show how meta-programming with concrete object syntax is implemented in and for Stratego. In Section 4 we outline a framework for extending a programming language to provide meta-programming with concrete syntax.

2 Abstract Syntax vs. Concrete Syntax

In this section we motivate the need for concrete syntax in meta-programming by contrasting the use of concrete syntax with the traditional use of abstract syntax. As an example we consider a simple meta-program for instrumenting Tiger programs with tracing statements. Tiger [2] is an imperative language with nested function definitions and statements that can be used in expressions.

We conclude the section with a discussion of the challenges posed to the meta-programming system by the use of concrete syntax.

2.1 Syntax Definition

A meta-programming system requires a syntax definition of the object language and a parser and pretty-printer that transform to and from the abstract syntax of the language used for internal representation. Figure 1 gives a condensed (i.e., incomplete) definition of the concrete and abstract syntax of Tiger. The concrete syntax is defined in the syntax definition formalism SDF [14,20]. An SDF production `sym1 ... symn -> sym` declares that an expression of sort `sym` can be constructed by the concatenation of expressions of sorts `sym1` to `symn`. SDF supports regular expression operators such as `{Exp " ; "}`, which denotes a list of Expressions *separated* by `;` semicolons. Furthermore, SDF integrates the definition of lexical and context-free syntax in one formalism. The formalism is modular so that (1) the syntax definition of a language can be divided into smaller (reusable) modules and (2) syntax definitions for separate languages can easily be combined. Since SDF definitions are declarative, rather than operational implementations of parsers, it is possible to generate other artifacts from syntax definitions such as pretty-printers [9] and signatures.

The abstract syntax is declared as a Stratego signature, which declares a term constructor for each language construct. The signature abstracts from syntactic details such as keywords and delimiters. Such a signature can be derived automatically from the syntax definition by using the `constructor` attributes as declarations of the constructor name for a language construct.

2.2 Example: Instrumenting Programs

Stratego [21,22] is a language for program transformation based on the paradigm of rewriting strategies. It supports the definition of basic transformations by means of rewrite rules. The application of such rules is controlled by programmable strategies.

Figure 2 shows the specification in Stratego of a transformation on Tiger programs that instruments each function `f` in a program such that it prints `f entry` on entry of the function and `f exit` at the exit. Functions are instrumented differently than procedures, since the body of a function is an expression statement and the return value is the value of the expression. It is not possible to just glue a print statement or function call at the end of the body. Therefore, a `let` expression is introduced, which introduces a temporary variable to which the body of the function is assigned. The rule `IntroducePrinters` generates code for the functions `enterfun` and `exitfun`, calls to which are added to functions and procedures. The transformation strategy `instrument` uses the generic traversal strategy `topdown` to apply the `TraceProcedure` and `TraceFunction` rules to all function definitions in a program, after which the printer functions are added, thus making sure that these functions are not instrumented themselves.

```

module Tiger-Condensed
exports
  sorts Exp
  context-free syntax
    Id          -> Var {cons("Var")}
    StrConst    -> Exp {cons("String")}
    Var         -> Exp
    "(" {Exp ";"* "}" -> Exp {cons("Seq")}
    Var "(" {Exp ","}* ")" -> Exp {cons("Call")}
    Exp "+" Exp -> Exp {left,cons("Plus")}
    Exp "-" Exp -> Exp {left,cons("Minus")}
    Var ":"= Exp -> Exp {cons("Assign")}
    "if" Exp "then" Exp "else" Exp -> Exp {cons("If")}
    "let" Dec* "in" {Exp ";"* "end" -> Exp {cons("Let")}
    "var" Id TypeAn ":"= Exp -> Dec {cons("VarDec")}
    FunDec+ -> Dec {cons("FunctionDec")}
    "function" Id "(" {FArg ","}* ")"
                                   TypeAn "=" Exp -> FunDec {cons("FunDec")}
    Id TypeAn -> FArg {cons("FArg")}
    ":" TypeId -> TypeAn {cons("NoTp")}
    ":" TypeId -> TypeAn {cons("Tp")}

module Tiger-Condensed
signature
  constructors
    Var      : Id -> Var
    String   : StrConst -> Exp
    Seq      : List(Exp) -> Exp
    Call     : Var * List(Exp) -> Exp
    Plus     : Exp * Exp -> Exp
    Minus    : Exp * Exp -> Exp
    Assign   : Var * Exp -> Exp
    If       : Exp * Exp * Exp -> Exp
    Let      : List(Dec) * List(Exp) -> Exp
    VarDec   : Id * TypeAn * Exp -> Dec
    FunctionDec : List(FunDec) -> Dec
    FunDec   : Id * List(FArg) * TypeAn * Exp -> FunDec
    FArg     : Id * TypeAn -> FArg
    NoTp     : TypeAn
    Tp       : TypeId -> TypeAn

```

Fig. 1. Concrete syntax definition in SDF (top) and corresponding abstract syntax signature in Stratego (bottom) of Tiger programs (condensed).

The top part of the figure shows the specification in concrete syntax while the bottom part shows the same specification using abstract syntax. For brevity, the `IntroducePrinters` rule is only shown in concrete syntax. Note that the full lexical syntax for identifiers and literals of the object language is used.

2.3 Concrete vs. Abstract

The example gives rise to several observations. The concrete syntax version can be read without knowledge of the abstract syntax. On the other hand, the abstract syntax version makes the tree structure of the expressions explicit. The abstract syntax version is much more verbose and is harder to read and write. Especially the definition of large code fragments such as in rule **IntroducePrinters** is unattractive in abstract syntax.

The abstract syntax version *implements* the concrete syntax version. The concrete syntax version has all properties of the abstract syntax version: pattern matching, term structure, can be traversed, and so on. In short, the concrete syntax is just sugar for the abstract syntax.

Extension of the Meta-language. We do not want to use Stratego only for meta-programming Tiger. Rather we would like to be able to handle arbitrary object languages. Thus, the object language or object languages that are used in a module should be a parameter to the compiler. The specification of instrumentation is based on the real syntax of Tiger, not on some combinators or infix expressions. This entails that the syntax of Stratego should be extended with the syntax of Tiger.

Meta-variables. The patterns in the transformation rules are not just fragments of Tiger programs. Rather some elements of these fragments are considered as meta-variables. For example in the term `[[function f(xs) = e]]` the identifiers `f`, `xs`, and `e` are not intended to be Tiger variables, but rather meta-variables, i.e., variables at the level of the Stratego specification.

Antiquotation. Instead of indicating meta-variables implicitly we could opt for an antiquotation mechanism that lets us splice in meta-level expressions into a concrete syntax fragment. For example, using `~` and `~*` as antiquotation operators, a variant of rule **TraceProcedure** becomes:

```
TraceProcedure :
  [[ function ~f(~* xs) = ~e ]] ->
  [[ function ~f(~* xs) =
    (print(~String(<conc-strings>(f," entry\\n"))));
    ~e;
    print(~String(<conc-strings>(f," exit\\n")))) ]]
```

With such antiquotation operators it becomes possible to directly embed meta-level computations that produce a piece of code within a syntax fragment.

3 Implementation

In the previous section we have seen how the extension of Stratego with concrete syntax for terms improves the readability of meta-programs. In this section we describe the techniques used to achieve this extension.


```

module Tiger-TraceAll
imports Tiger-Typed lib Tiger-Simplify
strategies
  instrument = topdown(try(TraceProcedure + TraceFunction));
               IntroducePrinters; simplify
rules
  TraceProcedure :
    [[ function f(xs) = e ]] ->
    [[ function f(xs) = (enterfun(s); e; exitfun(s)) ]]
    where !f => s
  TraceFunction :
    [[ function f(xs) : tid = e ]] ->
    [[ function f(xs) : tid =
       (enterfun(s);
        let var x : tid := nil in x := e; exitfun(s); x end) ]]
    where new => x ; !f => s
  IntroducePrinters :
    e -> [[ let var ind := 0
           function enterfun(name : string) = (
             ind := +(ind, 1);
             for i := 2 to ind do print(" ");
             print(name); print(" entry\\n")
           )
           function exitfun(name : string) = (
             for i := 2 to ind do print(" ");
             ind := -(ind, 1);
             print(name); print(" exit\\n")
           )
         in e end ]]

```

```

module Tiger-TraceAll
imports Tiger-Typed lib Tiger-Simplify
strategies
  instrument = topdown(try(TraceProcedure + TraceFunction));
               IntroducePrinters; simplify
rules
  TraceProcedure :
    FunDec(f, xs, NoTp, e) ->
    FunDec(f, xs, NoTp,
      Seq([Call(Var("enterfun"),[String(f)]), e,
           Call(Var("exitfun"),[String(f)])]))
  TraceFunction :
    FunDec(f, xs, Tp(tid), e) ->
    FunDec(f, xs, Tp(tid),
      Seq([Call(Var("enterfun"),[String(f)]),
           Let([VarDec(x,Tp(tid),NilExp)],
              [Assign(Var(x), e),
               Call(Var("exitfun"),[String(f)],
                 Var(x))])]))
    where new => x
  IntroducePrinters :
    e -> /* omitted for brevity */

```

Fig. 2. Instrumenting functions for tracing using concrete syntax and using abstract syntax.

3.1 Extending the Meta-language

To embed the syntax of an object language in the meta-language, the syntax definitions of the two languages should be combined and the object language sorts should be injected into the appropriate meta-language sorts. In the Stratego setting this is achieved as follows. The syntax of a Stratego module *M* is declared in the *M.syn* file, which declares the name of an SDF module. The SDF module combines the syntax of Stratego and the syntax of the object language(s)

by importing the appropriate SDF modules. The syntax definition of Stratego is provided by the compiler. The syntax definitions of the object language(s) are provided by the user. For example, Figure 3 shows a fragment of the syntax of Stratego and Figure 4 presents SDF module **StrategoTiger**, which defines the extension of Stratego with Tiger as object language. The module illustrates several remarkable aspects of the embedding of object languages in meta-languages using SDF.

A combined syntax definition is created by just importing appropriate syntax definitions. This is possible since SDF is a modular syntax definition formalism. This is a rather unique feature of SDF and essential to this kind of language extension. Since only the full class of context-free grammars, and not any of its subclasses such as LL or LR, are closed under composition, modularity of syntax definitions requires support from a generalized parsing technique. SDF2 employs scannerless generalized-LR parsing [20,8].

The syntax definitions for two languages may partially overlap, e.g., define the same sorts. SDF2 supports renaming of sorts to avoid name clashes and ambiguities resulting from them. In Figure 4 several sorts from the Stratego syntax definition (**Id**, **Var**, and **StrChar**) are renamed since the Tiger definition also defines these names.

The embedding of object language expressions in the meta-language is implemented by adding appropriate injections to the combined syntax definition. For example, the production

```
"[" Exp "]" -> Term {cons("ToTerm"),prefer}
```

declares that a Tiger expression (**Exp**) between `[` and `]` can be used everywhere where a Stratego **Term** can be used. Furthermore, abstract syntax expressions (including meta-level computations) can be spliced into concrete syntax expressions using the `~` splice operators. To distinguish a term that should be interpreted as a list from a term that should be interpreted as a list *element*, the convention is to use a `~*` operator for splicing a list.

The declaration of these injections can be automated by generating an appropriate production for each sort as a transformation on the SDF definition of the object language. It is, however, useful that the embedding can be programmed by the meta-programmer to have full control over the selection of the sorts to be injected, and the syntax used for the injections.

```
module Stratego
exports
  context-free syntax
  Int          -> Term {cons("Int")}
  String       -> Term {cons("Str")}
  Var          -> Term {cons("Var")}
  Id "(" {Term ","}* ")" -> Term {cons("Op")}
  Term "->" Term   -> Rule {cons("RuleNoCond")}
  Term "->" Term "where" Strategy -> Rule {cons("Rule")}
```

Fig. 3. Fragment of the syntax of Stratego

```

module StrategoTiger
imports
  Tiger Tiger-Sugar Tiger-Variables Tiger-Congruences
imports
  Stratego [ Id => StrategoId
            Var => StrategoVar
            StrChar => StrategoStrChar ]
exports
  context-free syntax
  "[[" Dec      "]" ]" -> Term      {cons("ToTerm"),prefer}
  "[[" FunDec   "]" ]" -> Term      {cons("ToTerm"),prefer}
  "[[" Exp      "]" ]" -> Term      {cons("ToTerm"),prefer}
  "~" Term      -> Exp            {cons("FromTerm"),prefer}
  "~*" Term     -> {Exp " ", "+"} {cons("FromTerm")}
  "~*" Term     -> {Exp ";", "+"} {cons("FromTerm")}
  "~" Term      -> Id            {cons("FromTerm")}
  "~*" Term     -> {FArg " ", "+"} {cons("FromTerm")}

```

Fig. 4. Combination of syntax definitions of Stratego and Tiger

```

module Tiger-Variables
exports
  variables
  [s] [0-9]* -> StrConst {prefer}
  [xyzfgh] [0-9]* -> Id {prefer}
  [e] [0-9]* -> Exp {prefer}
  "xs" [0-9]* -> {FArg " ", "+"} {prefer}
  "ds" [0-9]* -> Dec+ {prefer}
  "ta" [0-9]* -> TypeAn {prefer}

```

Fig. 5. Some variable schema declarations for Tiger sorts.

3.2 Meta-variables

Using the injection of meta-language **Terms** into object language **Expressions** it is possible to distinguish meta-variables from object language identifiers. Thus, in the term `[[var ~x := ~e]]`, the expressions `~x` and `~e` indicate meta-level terms, and hence `x` and `e` are meta-level variables. However, it is attractive to write object patterns with as few squiggles as possible. This can be achieved using SDF variable declarations. Figure 5 declares syntax schemata for meta-variables. According to this declaration `x`, `y`, and `g10` are meta-variables for identifiers and `e`, `e1`, and `e1023` are meta-variables of sort `Exp`. The `prefer` attribute ensures that these identifiers are preferred over normal Tiger identifiers [8].

3.3 Meta-explode

Parsing a module according to the combined syntax and mapping the parse tree to abstract syntax results in an abstract syntax tree that contains a mixture of

meta- and object language abstract syntax. Since the meta-language compiler only deals with meta-language abstract syntax, the embedded object language abstract syntax needs to be expressed in terms of meta abstract syntax. For example, parsing the following Stratego rule

```
[[ x := let ds in ~* es end ]] -> [[ let ds in x := (~* es) end ]]
```

with embedded Tiger expressions, results in the abstract syntax tree

```
Rule(ToTerm(Assign(Var(meta-var("x")),
  Let(meta-var("ds"),FromTerm(Var("es")))),
  ToTerm(Let(meta-var("ds"),
    [Assign(Var(meta-var("x")),
      Seq(FromTerm(Var("es"))))]))))
```

containing Tiger abstract syntax constructors (e.g., **Let**, **Var**, **Assign**) and meta-variables (**meta-var**). The transition from meta-language to object language is marked by the **ToTerm** constructor, while the transition from meta-language to object language is marked by the constructor **FromTerm**.

Such mixed abstract syntax trees can be normalized by ‘exploding’ all embedded abstract syntax to meta-language abstract syntax. Thus, the above tree should be exploded to the following pure Stratego abstract syntax:

```
Rule(Op("Assign",[Op("Var",[Var("x")]),
  Op("Let",[Var("ds"),Var("es")])]),
  Op("Let",[Var("ds"),
    Op("Cons",[Op("Assign",[Op("Var",[Var("x")]),
      Op("Seq",[Var("es")])]),
      Op("Nil",[])])])])
```

Observe that in this explosion all embedded constructors have been translated to the form **Op**(**C**, [**t**₁, ..., **t**_n]). For example, the Tiger ‘variable’ constructor **Var**(_) becomes **Op**("Var", [_]), while the Stratego meta-variable **Var**("es") remains untouched, and **meta-vars** become Stratego **Vars**. Also note how the list in the second argument of the second **Let** is exploded to a **Cons/Nil** list. The resulting term corresponds to the Stratego abstract syntax for the rule

```
Assign(Var(x),Let(ds,es)) -> Let(ds,[Assign(Var(x),Seq(es))])
```

written with abstract syntax notations for terms.

The explosion of embedded abstract syntax does not depend on the object language, but can be expressed generically, provided that embeddings are indicated with the **FromTerm** constructor. The *complete* implementation of the **meta-explode** transformation on **Term** abstract syntax trees is presented in Figure 6. The strategy **meta-explode** uses the generic strategy **alltd** to perform a generic traversal over the abstract syntax tree of a Stratego module. Anywhere in this tree where it finds a **ToTerm**(_), its argument is exploded using **trm-explode**. This latter strategy is composed from a number of rules that recognize special cases. The general case is handled by **TrmOp**, which decomposes a term into its constructor **op** and arguments **ts**, and constructs an abstract syntax term **Op**(**op**,**ts**'), where the **ts**' are the exploded arguments. The transformation **str-explode** is similar to **trm-explode**, but transforms embedded abstract syntax into strategy expressions.

```

module meta-explode
imports lib Stratego
strategies
  meta-explode =
    alltd(?ToTerm(<trm-explode>) + ?ToStrategy(<str-explode>))

  trm-explode =
    TrmMetaVar <+ TrmStr <+ TrmFromTerm <+ TrmFromStr <+ TrmAnno
    <+ TrmConc <+ TrmNil <+ TrmCons <+ TrmOp

  TrmOp      : op#(ts) -> Op(op, <map(trm-explode)> ts)

  TrmMetaVar : meta-var(x) -> Var(x)
  TrmStr      = is-string; !Str(<id>)
  TrmFromTerm = ?FromTerm(<meta-explode>)
  TrmFromStr  = ?FromStrategy(<meta-explode>)
  TrmAnno     = Anno(trm-explode, meta-explode)
  TrmNil      : [] -> Op("Nil", [])
  TrmCons     : [x | xs] -> Op("Cons", [<trm-explode>x, <trm-explode>xs])
  TrmConc     : Conc(ts1,ts2) ->
    Op("Conc", [<Fst>, <Snd>]), trm-explode> ts1

```

Fig. 6. Generic definition of meta-explode

4 Generalization

In the previous section we described the embedding of concrete syntax for object languages in Stratego. This approach can be generalized to other meta-languages. In this section we outline the ingredients needed to make your favorite language into a meta-language.

Given a (general-purpose) language M to be used as meta-language and a language O , which may be a data format, a programming language, or a domain-specific language, as long as it has a formal syntax, we can extend M to a meta-language for manipulating O programs. Figure 7 depicts the architecture of this extension and the components that are employed. The large box denotes the extension of the M compiler **m-compile** with concrete syntax for O . From a meta-programmer's point of view this is a black box that implements the compiler (dashed arrow) **mo-compile**, which consumes source meta-programs and produces executable meta-programs. In the rest of this section we briefly discuss the components involved.

ATerm Library.

- The communication between the various components is achieved by exchanging ATerms [6], a generic format for exchange of structured data.

SDF tools.

- **pack-sdf**: collection of all imported SDF modules;
- **sdf2table**: parser generator for SDF;
- **sgrl**: scannerless generalized-LR parser reads a parse table (**M-0.tbl**) and parses a source file according to it;
- **implode-asfix**: translation from parse trees to abstract syntax trees;
- Optionally one can use pretty-printer and signature generators.

M as meta-language.

- A syntax definition **M.sdf** of *M*
- A model for object program representation in *M* (e.g., AST represented as term)
- An API for constructing and analyzing *O* programs in *M* (e.g., pattern instantiation and matching)
- **m-explode**: An explosion algorithm for transforming *O* abstract syntax expressions into *M* expressions. If the object language program representation is generic, i.e., does not depend on a specific *O*, this can be implemented generically, as was done for Stratego using **meta-explode**. This is a transformation on *M* programs.

O as object language.

- A syntax definition **O.sdf** of *O*
- A combined syntax definition **M-0.sdf**, possibly resolving name clashes
- Meta-variable declarations for *O*
- Injection of *O* expressions into *M* expressions
 - Selection of *O* syntactic categories to manipulate (e.g., **Exp** and **Dec**)
 - Selection of *M* syntactic categories in which *O* expressions should be injected (e.g., **Term**)
 - Quotation syntax (e.g., **[[...]]**)
 - Anti quotation syntax (e.g., **~...**)

It is possible to automate this by generating syntax for variables, quotations, and antiquotations automatically from the syntax definition of *O*, provided that there is a standard convention for quotation and antiquotation.

M compiler.

- After **m-exploding** meta-programs they can be compiled by the usual compiler **m-compile** for *M*. If the compiler does not have an option to consume abstract syntax trees, but only text, it is necessary to pretty-print the program first.

O meta-programs.

- Finally, we can write a meta-program **MetaProg.mo** using concrete syntax and compile it to an executable **MetaProg.bin** that manipulates *O* programs.

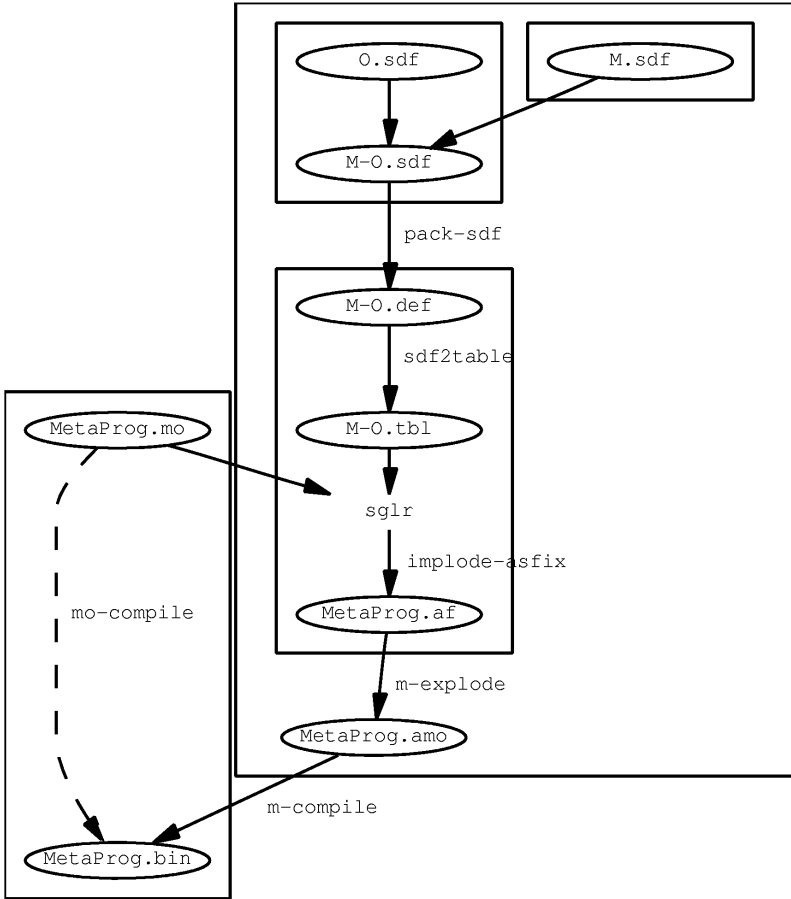


Fig. 7. Architecture for meta-programming with concrete object syntax

5 Discussion

5.1 Syntax Definition and Parsing

SDF [14] was originally designed for use as a general syntax definition formalism. However, through its implementation it was closely tied to the algebraic specification formalism ASF+SDF [4,13], which is supported by the ASF+SDF Meta-Environment [16,7]. Redesign and reimplementaion of SDF [20,8] has made SDF available for use outside the Meta-Environment. SDF is also distributed as part of the XT bundle of program transformation tools [15].

Syntax definition in SDF is limited to *context-free grammars*. This is a limitation for languages with context-sensitive syntax such as C (type identifiers) and Haskell (offside rule). However, in the setting of meta-programming with concrete object syntax, in which small fragments are used and not all context is

always available, any parsing technique will have a hard time. This type of problem made Cameron and Ito [10] suggest that language designers should consider meta-programming in designing the syntax of a programming language.

5.2 Desugaring Patterns

Some meta-programs first desugar a program before transforming it further. This reduces the number of constructs and shapes a program can have. For example, the Tiger binary operators are desugared to prefix form:

```
DefTimes : [[ e1 * e2 ]] -> [[ *(e1, e2) ]]
DefPlus  : [[ e1 + e2 ]] -> [[ +(e1, e2) ]]
```

or in abstract syntax

```
DefPlus : Plus(e1, e2) -> BinOp(PLUS, e1, e2)
```

This makes it easy to write generic transformations for binary operators. However, all subsequent transformations on binary operators should then be done on these prefix forms, instead of on the usual infix form. This is the reason why in Figure 2, the expression `-(ind,1)` is used instead of `(ind - 1)`. However, users/meta-programmers think in terms of the infix operators and would like to write rules such as

```
Simplify : [[ e + 0 ]] -> [[ e ]]
```

However, this rule will not match since the term to which it is applied has been desugared. Thus, it might be desirable to desugar embedded abstract syntax with the same rules with which programs are desugared. This phenomenon occurs in many forms ranging from removing parentheses and generalizing binary operators as above, to decorating abstract syntax trees with information resulting from static analysis such as type checking.

5.3 User-Definable Syntax

Programming languages with user-definable syntax have been a long standing goal of programming language research. To a certain extent programming languages do provide domain-specific or user-definable syntax. The use of infix syntax for arithmetic and logical operators is such a standard component of programming language syntax, that it is not considered a syntactic extension. However, they are clearly domain-specific operations, that could just as well be expressed using function call syntax. Indeed a number of languages (Prolog, Haskell, ...) allow the user to introduce new infix operators and define them just like a normal predicate or function. Other languages, especially in the domain of algebraic specification and theorem proving, have support for user-defined mix-fix operators (e.g., OBJ, ELAN, Maude). This approach is taken to its extreme in the algebraic specification formalism ASF+SDF [4,13] in which all expression constructors are defined by the user, including the *lexical syntax*. An ASF+SDF specification consists of modules defining syntax and conditional equations over terms induced by this syntax. The equations are interpreted as term rewrite rules.

The influence of ASF+SDF on the work described in this paper is profound—Stratego grew out of experience with ASF+SDF.

The architecture of JTS [3] is much like the one described in this paper, but the goal is the extension of languages with domain-specific constructs. The JTS tools are based on less powerful (i.e., lex/yacc) parsing technology.

Several experiments have been done with dynamically (parse-time) extensible syntax [23,11,5]. In these tools the program itself contains declarations of syntax extensions. This complicates the parsing process considerably. We have chosen to define the syntax in a separate file. This entails that the syntax for an entire module is fixed and cannot be extended half way. This is reasonable for meta-programming since the syntactic domain of meta-programs is usually a fixed object language or set of object languages. Changing the object language on a per module basis is fine grained enough.

5.4 Syntax Macros

The problem of concrete object syntax is different from extending a language with new constructs, for example, extending C with syntax for exception handling. This application known as syntax macros, syntax extensions, or extensible syntax [23,11,5] can be expressed using the same technology as discussed in this paper. Indeed, Stratego itself is an example of a language with syntactic extensions that are removed using transformations. For example, the following rules define several constructs in terms of the more primitive match (`?t`) and build (`!t`) constructs [22].

```
Desugar :
  [[ s => t ]] -> [[ s; ?t ]]
Desugar :
  [[ <s> t :S]] -> [[ !t; s ]]
Desugar:
  [[ f(as) : t1 -> t2 where s ]] -> [[ f(as) = ?t1; where(s); !t2 ]]
```

6 Conclusions

Contribution. In this paper we have shown how concrete syntax notation can be fitted with minimal effort onto *any* meta-language, which need not be specifically designed for it. The use of concrete syntax makes meta-programs more readable than abstract syntax specifications. Due to the scannerless generalized parsing technology no squiggles are needed for antiquotation of meta-variables leading to very readable code fragments.

The technical contributions of this paper are the implementation of concrete syntax in Stratego and a general architecture for adding concrete object syntax to any (meta-)language. The application of SDF in Stratego is more evidence for the power of the SDF/SGLR technology. The composition of the SDF components with the Stratego compiler is a good example of component reuse.

Future Work. The ability to fit concrete syntax onto a meta-programming language opens up a range of applications and research issues for exploration: transformation of various object languages such as Java and XML in Stratego; addition of concrete syntax to other meta-languages, which might involve mapping to a more distant syntax tree API than term matching and construction; object-language specific desugaring; and finally language extensions instead of meta-programming extensions.

Availability. The components used in this project, including the ATerm library, SDF, and Stratego, are freely available and ready to be applied in other meta-programming projects. The SDF components are available from <http://www.cwi.nl/projects/MetaEnv/pgen/>. Bundles of the SDF components with other components such as Stratego are available from the Online Package Base at <http://www.program-transformation.org/package-base>

Acknowledgments. Joost Visser provided comments on a previous version of this paper.

References

1. A. Aasa. *User Defined Syntax*. PhD thesis, Dept. of Computer Sciences, Chalmers University of Technology and University of Göteborg, Göteborg, Sweden, 1992.
2. A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.
3. D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: A tool suite for building genovca generators. In *5th International Conference in Software Reuse, (ICSR'98)*, Victoria, Canada, June 1998.
4. J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.
5. C. Brabrand and M. I. Schwartzbach. Growing languages with metamorphic syntax macros. In *PEPM'02*, 2002.
6. M. G. J. van den Brand, H. de Jong, P. Klint, and P. Olivier. Efficient annotated terms. *Software, Practice & Experience*, 30(3):259–291, 2000.
7. M. G. J. van den Brand, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The Asf+Sdf Meta-Environment: a component-based language laboratory. In R. Wilhelm, editor, *Compiler Construction (CC'01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–368, Genova, Italy, April 2001. Springer-Verlag.
8. M. G. J. van den Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In N. Horspool, editor, *Compiler Construction (CC'02)*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158, Grenoble, France, April 2002. Springer-Verlag.
9. M. G. J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5(1):1–41, January 1996.
10. R. D. Cameron and M. R. Ito. Grammar-based definition of metaprogramming systems. *ACM Trans. on Programming Languages and Systems*, 6(1):20–54, 1984.

11. L. Cardelli, F. Matthes, and M. Abadi. Extensible syntax with lexical scoping. SRC Research Report 121, Digital Systems Research Center, Palo Alto, California, February 1994.
12. J. R. Cordy, C. D. Halpern, and E. Promislow. TXL: a rapid prototyping system for programming language dialects. In *Proc. IEEE 1988 Int. Conf. on Computer Languages*, pages 280–285, 1988.
13. A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, Singapore, September 1996.
14. J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF – reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.
15. M. de Jonge, E. Visser, and J. Visser. XT: A bundle of program transformation tools. In M. G. J. van den Brand and D. Perigot, editors, *Workshop on Language Descriptions, Tools and Applications (LDTA'01)*, volume 44 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, April 2001.
16. P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.
17. M. Mauny and D. de Rauglaudre. A complete and realistic implementation of quotations in ML. In *Proc. 1994 Workshop on ML and its applications*, pages 70–78. Research report 2265, INRIA, 1994.
18. S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In R. Hinze, editor, *2001 Haskell Workshop*, Firenze, Italy, September 2001. ACM SIGPLAN.
19. W. Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.
20. E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.
21. E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.
22. E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *Proc. of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998.
23. D. Weise and R. F. Crew. Programmable syntax macros. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation (PLDI'93)*, Albuquerque, New Mexico, June 1993.

Managing Dynamic Changes in Multi-stage Program Generation Systems^{*}

Zhenghao Wang and Richard R. Muntz

Computer Science Department, University of California Los Angeles

Abstract. In a *multi-stage program generation* (MSPG) system, a stage- s program generates a stage- $s + 1$ program when the values of some variables are known. We call such variables *program parameters* for stage- s .

When program parameters for a stage change during runtime, all later-stage program objects that are generated directly or indirectly based on them need to be dynamically regenerated. We make two contributions. a) We explore a metaobject protocol called *reflection across stages* (RAS), which allows later-stage program objects to refer back to earlier-stage program objects they originated from. Intercessory procedures can be specified by the earlier-stage program objects to be executed at, e.g., execution of the later stage objects. b) We apply RAS to automating runtime program regeneration, so that affected later-stage programs are automatically regenerated after program parameters change.

In an initial experiment, RAS incurred an overhead of 10% when program parameters are invariant. The overhead of RAS plus regeneration is amortized to zero over 1.5 executions of the generated program object.

1 Introduction

There has been a recent interest in *multi-stage programming* [27,26,19,24,25,22,10,3], in which an earlier-stage program describes the computation for generating a later-stage program. The main use of multi-stage programming is to write generic and highly-parameterized programs that can handle a wide variety of runtime situations not known at development or compile time but avoid excessive runtime overheads potentially associated with this generality.

There are many aspects to a multi-stage programming language, but in this paper we only explore some of them. For our purposes, we define the more general notion of *multi-stage program generation* (MSPG) as follows.

^{*} This material is based upon work supported by the National Science Foundation under Grant Nos. 0086116, 0085773, and 9817773.

Definition 1 (MSPG) *An MSPG system satisfies the following: a) it has first-class program objects,^{1,2,3} b) it has facilities to build and execute program objects, and c) (for systems with more than two stages) the execution of a program object may result in either a data object or yet another program object.*

A multi-stage programming language such as MetaML [27,24] and Staged Scheme (S^2 , to be introduced in Sec. 2) can be viewed as an MSPG system with a template-based homogeneous typed language for all stages. Note that under this definition, Lisp’s *eval* mechanism (with or without quasiquotes[1]) can be viewed as a primitive MSPG system; another modern example is ‘C [9,21].

1.1 The Problem—Automatic Program Regeneration

In MSPG, a stage- s program object generates a stage- $s+1$ program object when the values of some variables are known. We call such variables *program parameters* for stage- s , which may be user-configurable preferences, measurements of physical quantities, data obtained from a remote service, etc.

One important issue is dynamic regeneration of later-stage program objects during runtime, when program parameter values may change dynamically after the later-stage programs are generated. This situation may happen when user changes her configuration, measurements of physical quantities change at the deployment site, or data from a remote service change. All later-stage program (and even data) objects that are generated directly or indirectly based on the program parameters whose values change need to be dynamically regenerated.

There are many possible strategies for regeneration. In this paper, we consider the lazy and automatic strategy in which an affected program (or even data) object is automatically regenerated just before it is used. This hints that we could use a mechanism that intercepts the using of a program/data object, traces back to earlier-stage program objects, and carries out earlier-stage computation again if necessary. This mechanism is provided by *reflection across stages* (RAS), which is briefly sketched in the next subsection.

1.2 A Solution—Reflection across Stages (RAS)

From Definition 3, the execution of a stage- s (program) object (o_s) results in a stage- $(s+1)$ (data or program) object (o_{s+1}). o_s can be viewed as a “metaobject” of o_{s+1} and the relationship between object pairs such as o_s and o_{s+1} is a “meta-relationship” upon which we can define a metaobject protocol [16].

To illustrate this, consider the following interaction with Scheme:

¹ This means that program objects are of the same status as other data objects. Program objects can be created dynamically, stored in data structures, returned as results of procedures, etc.

² Program objects have also been called code objects or code values by other authors.

³ Program objects denote abstract syntax trees. They differ from function objects in that their tree structures may be observed and/or manipulated. Such manipulation may even proceed under lambda.

```

> (define o0 '(list '+ (+ 1 2) '4)) ; o0 = "(LIST '+ (+ 1 2) '4)"
> (define o1 (eval o0))             ; o1 = "(+ 3 4)"
> (define o2 (eval o1))             ; o2 = "7"

```

where we adopted the convention of double-quoting object values in the comment of each line. We assign stage numbers 0, 1, 2 to `o0`, `o1`, `o2`, respectively, and `o0` and `o1` are “metaobjects” with respect to `o1` and `o2` respectively.

The metaobject protocol defined upon the meta-relationship between objects of different stages is called *reflection across stages* (RAS). RAS allows data objects or program objects to refer back to the program objects or (meta-)program objects they originated from. Intercessory procedures can be specified by earlier-stage program objects to be executed at certain points, e.g., execution of the later stage objects. RAS is closely related to two classical works on *computational reflection* [7] in programming languages, namely 3-Lisp [23,8] and MetaObject Protocol for CLOS [16]. We will explore the detail of RAS and apply it to address the problem of automatic program regeneration in this paper.

RAS is a general tool that can address more problems than automatic program regeneration. For example, the number of stages and which program parameters are associated with each stage are now determined at development time, based on estimates of which parameters change most frequently. However, the change frequencies of program parameters may not become known until runtime. With RAS, it is possible to even reorganize the staging design dynamically.

1.3 Plan of the Paper

The rest of the paper is organized as follows. The next two sections (Secs. 2 and 3) prepare the reader with necessary notation and a motivating as well as running example for the main content that follows. The two main topics, namely RAS and automatic program regeneration, are discussed in Secs. 4 and 5, respectively. After those, the results of an initial experiment are presented in Sec. 6, followed by a discussion of related work in Sec. 7. Sec. 8 concludes this paper.

2 Staged Scheme (S^2)

For the purpose of “proof of concept,” we have chosen to develop a prototype interpreter with RAS for Staged Scheme (S^2)—a Scheme version of the multi-stage programming language MetaML [27,24]. The enhanced language is called Staged Scheme with Reflection across Stages (S^2 RAS), which will be developed in Sec. 4. The rest of this section gives a brief introduction to S^2 .

Readers who are not familiar with MetaML are invited to read Appendix A. Basically, S^2 provides a template language for building program objects. The template notations for S^2 are listed in Table 1 together with their counterparts in MetaML to facilitate readers who are already familiar with MetaML. The “up” special form (\uparrow $_$) builds a program object, specified by the template $_$. The template is just a Scheme expression with one exception—it may contain “holes” that are “down” special forms (\downarrow e). The holes are used to generate

Table 1. Template notations in Staged Scheme (S^2).

notation	Staged Scheme (S^2)	MetaML
meta-brackets/up	(\uparrow $_$) *	<_>
escape/down	(\downarrow $_$) *	~ $_$
run	(! $_$)	run $_$
lift	(\$ $_$)	lift $_$

* In the real implementation, we use (\wedge $_$), (\vee $_$) to represent (\uparrow $_$), (\downarrow $_$) respectively.

Table 2. Correspondence of λ -notation and concrete Scheme program.

λ -notation	Scheme
$\lambda v_1 \dots v_n. e$	(lambda (v1 ... vn) e)
$\lambda(). e$	(lambda () e)
let $v_1 = e_1, \dots, v_n = e_n$ in e	(let ((v1 e1) ... (vn en)) e)
letrec $v_1 = e_1, \dots, v_n = e_n$ in e	(letrec ((v1 e1) ... (vn en)) e)
if e_1 then e_2 else e_3	(if e1 e2 e3)
$e_1 \text{ op } e_2$ *	(+ope1 e2), where <i>op</i> is an infix operator in mathematics

* Parentheses are used like in mathematics when they are clearly not Scheme ones.

dynamic content for the program object built with the up special form (\uparrow $_$), i.e., e in the holes are evaluated and the results are spliced in the program object being built. \uparrow may be nested to give rise to more than two stages. ($!$ o) is used to execute a program object o , and ($\$$ o) is used to turn a data object o into the corresponding program object.

As an example, the multi-stage computation described in the example in Sec. 1.2 can be reestablished in S^2 as:

```
> (define o0 ( $\uparrow$  ( $\uparrow$  (+ ( $\downarrow$  ($ (+ '1 '2))) 4)))) ; o0 = "( $\uparrow$  ( $\uparrow$  (+ ( $\downarrow$  ($ (+ '1 '2))) '4)))"
> (define o1 (! o0)) ; o1 = "( $\uparrow$  (+ '3 '4))"
> (define o2 (! o1)) ; o2 = "7"
```

We would like to point out that static typing, which is present in MetaML, is lost in S^2 . However, static typing is orthogonal to the main focus of this paper. Our choice of a dynamically typed language is only for the sake of simplicity, and this paper applies to statically typed languages in principle.

Note no notation. We use Table 2. to abbreviate Scheme programs as λ -notations.

3 The Lagrange Interpolation Example

A simple running example we will use in this paper is specializing Lagrange interpolation as in the following scenario: we have a set of sensors measuring a physical quantity (e.g., temperature) over a physical space and we need to set up a service that replies to potentially many clients the interpolated values of the physical quantity at client-specified positions.

```

(define code-0 (↑ (↑ (↑ λx f1 f2 f3.
  let x1 = x - x1, x2 = x - x2, x3 = x - x3 in
  (↓ (↓ let x12 = x1 - x2 in
    (↑ let x13 = x1 - x3, x23 = x2 - x3 in
      (↑ f1 * x2 * x3 / (↓ ($ x12 * x13)) -
        f2 * x1 * x3 / (↓ ($ x12 * x23)) +
        f3 * x1 * x2 / (↓ ($ x13 * x23))))))))))

```

Fig. 1. S² program for specializable Lagrange interpolation function.

For simplicity, we use one-dimensional Lagrange interpolation with three sensors as our running example. Suppose that the positions of the three sensors are x_1 , x_2 , and x_3 , and the measurements from the three sensors are f_1 , f_2 , and f_3 , respectively. Then the interpolated value at position x is:

$$f\text{-interp}(x, x_1, x_2, x_3, f_1, f_2, f_3) = \sum_{i=1}^3 f_i \prod_{j \neq i} (x - x_j) / \prod_{j \neq i} (x_i - x_j).$$

The fact that sensors may be moved infrequently suggests turning x_1 , x_2 , and x_3 into program parameters, so the interpolation function in our implementation is actually a function of only four formals: x , f_1 , f_2 , and f_3 : $f\text{-interp}(x, f_1, f_2, f_3)$.

Suppose that x_1 and x_2 change even less frequently than x_3 , then we can make x_1 and x_2 program parameters for the first stage and x_3 for the second stage. We can write an S² program as shown in Fig. 1, where `code-0` can be stage-wise specialized to obtain procedure object `f-interp` when (`x1`, `x2`) and `x3` are defined in the first and second stages:

```

> (define x1 0)    > (define x2 6)    > (define code-1 (! code-0))
> (define x3 -1)   > (define code-2 (! code-1))
> (define f-interp (! code-2))
> (f-interp 2 -1 11 4)
-5

```

The chain of metaobjects/objects is

$$\text{code-0} \xrightarrow{!(x_1, x_2)} \text{code-1} \xrightarrow{!(x_3)} \text{code-2} \xrightarrow{!} \text{f-interp}$$

Once in a while, we need to move the sensors. Then the specialized interpolation function $f\text{-interp}$ needs to be regenerated. We will study how to automatically regenerate $f\text{-interp}$ in this paper. With automatic regeneration, we just move the sensors and the service automatically updates its interpolation function $f\text{-interp}$, given that sensor locations can be obtained automatically. There is no need to stop the service and manually respecialize the service program.

One fear is that MSPG adds excessive burden to programmers. For example, programmers need to manually add \uparrow , \downarrow , etc. Though adding them should become easy after some practice, this issue can be completely solved by automating this process using, e.g., binding-time analysis partial evaluation techniques. In other words, programmers only need to specify program parameters for each stage, and staging annotations such as \uparrow and \downarrow are automatically added (or even not

added if the automatic process decides that it is not beneficial to do so). This is possible in practical systems[17,18]. Another major issue is how to assign stage numbers to program parameters. Again, this issue can be solved by automatically profiling change frequencies and program generation overheads at runtime. As mentioned at the end of Sec. 1.2, RAS is a general enough mechanism to support this approach. These interesting topics are beyond the main focus of this paper.

4 Reflection across Stages (RAS)

4.1 Meta-levels of Stages

In Smith's paper on procedural reflection [23,8], reflection happens across meta-levels of structural fields of meta-circular interpreters. In a typical metaobject protocol designed for an object-oriented programming language [16], reflection happens across meta-levels of objects and metaobjects that implement the method dispatch semantics. Following this analogy, reflection happens across meta-levels of objects of different stages in our case. A later-stage object is related to the earlier-stage program object that it "originates from," and reflection allows access to the earlier-stage program object. In addition, intercessory constructs allow an earlier-stage program object to intercept the using of a later-stage object, so as to take some control of the behavior of the later-stage object.

Definition 2 (originates-from) *If $(! \langle exp \rangle)$ evaluates to an object o_{s+1} , while during the same evaluation process, $\langle exp \rangle$ evaluates to o_s (which must be a program object), then we say that stage- $(s+1)$ object o_{s+1} originates from stage- s program object o_s . We call o_s the originator of o_{s+1} , and o_{s+1} the product of o_s .*

A note is that the result object o_{s+1} could already have its originator determined before the $!$ special form is evaluated, as shown below:

```
> (define codea-0 (↑ (↑ ⟨exp⟩)))
> (define codea-1 (! codea-0))
> (define codeb-0 (↑ codea-1))
> (define codeb-1 (! codeb-0))
```

Due to cross-stage persistence⁴, **codeb-1** is the same object as **codea-1**, whose originator is **codea-0**. In such case, we do not alter the existing originates-from relationship i.e., the originator of **codeb-1** is **codea-0** instead of **codeb-0**.

4.2 Accessors

Accessors provide a direct way to access the originator and stage number:

```
(originator-of ⟨object⟩)      (stage-of ⟨object⟩)
```

As a simple example, the following procedure returns the chain of originators:

```
(define (originator-of* x)
  (if (zero? (stage-of x)) '(),x)
      (let ((o-x (originator-of x))) (cons x (originator-of* o-x)))))
```

⁴ Cross-stage persistence is explained in Appendix A.

4.3 Introspection of Program Objects

We omitted a discussion as this is beyond the main theme of this paper.

4.4 Intercessory Constructs

Intercessory constructs are often very useful reflective constructs [23,8,16]. There are many points during the execution of a multi-stage program that are potentially interceptable. In general, we may intercept an S^2 object, call it $\langle object \rangle$, and reflect to its originator when entering the evaluation or rebuilding of a special form with $\langle object \rangle$ as a subcomponent.⁵ We identify some of these points:

AppOp(0) An operator procedure object upon being called: $(\langle object \rangle \dots)$;
AppRand(0) An operand object upon being bound to formals:⁶
 $(\dots \langle object \rangle \dots)$;
Run(0) A program object upon being run: $(! \langle object \rangle)$;
Down($n + 1$) A program object upon being spliced: $(\downarrow \langle object \rangle)$ at rebuilding level $n + 1 > 0$;⁷
Lift(0) An object upon being lifted: $(\$ \langle object \rangle)$;

To allow interception at these points, we augment the originator object with a set of interception procedures, each of which has the following form:

Event: $(\text{lambda } (o \ k \ \dots \ \text{param } \dots) \text{"body..."}),$

where o is the intercepted object; $k \ \dots$ are escape procedures (continuations) that the body of the interception procedure may call; **param** \dots are additional parameters if there are any. The object returned by the interception procedure is either the same or different from o . It is used by the runtime to continue evaluation. In addition, the interception procedure may call one of the escape procedures to alter the default control flow. The interception procedures that correspond to the above five interception points are:

AppOp: $(\text{lambda } (o \ k \ \text{args}) \text{"body..."})$
AppRand: $(\text{lambda } (o) \text{"body..."})$
Run: $(\text{lambda } (o \ k) \text{"body..."})$
Down: $(\text{lambda } (o \ k \ n+1) \text{"body..."})$
Lift: $(\text{lambda } (o \ k) \text{"body..."}),$

where each escape procedure k corresponds to the continuation of the whole application, run $(!)$, down (\downarrow) , or lift $(\$)$ special form. **args** in **AppOp** is the original list of arguments supplied to the procedure object o .

We extend the \uparrow special form to associate interception procedures with the program object that it evaluates to:

$(\uparrow \langle exp \rangle \text{Event1: proc1-definition Event2: proc2-definition } \dots),$

where **Event1**, **Event2**, ... can be one of **AppOp**, **AppRand**, **Run**, **Down**, **Lift**, etc.

⁵ It is also theoretically interesting to consider other events, e.g. leaving the evaluation.

⁶ This event also occurs when **let** binds objects to formals.

⁷ See Appendix A for rebuilding level.

Intercession up the levels. If o_s is intercepted by its originator o_{s-1} , and if o_{s-1} is used in the interceptor procedure, then we can effectively reflect to o_{s-1} 's originator o_{s-2} . Moreover, it is sometimes convenient to associate the following interception procedure to o_{s-2} , so that an event of calling an interception procedure of o_{s-1} is more explicitly intercepted by o_{s-2} :

```
Int: (lambda (os-1 os) "body..."),
```

The object returned from this interception procedure, whether the same as o_{s-1} or not, is going to be responsible for handling the original interception from o_s .

Splicing with interception procedures. In the original S^2 sans interception procedures, the only component of a program object (e.g., of value “(\uparrow $\langle exp \rangle$)”) is an expression (i.e., “ $\langle exp \rangle$ ”). \downarrow operating on that program object splices the component expression into the surrounding program object being rebuilt.

With the association of interception procedures, an expression may not be the only component of a program object. Consider the following example:

```
> (define code-a ( $\uparrow$   $\langle exp \rangle$  Event: proc))
> (define code-b ( $\uparrow$  ( $\downarrow$  code-a)))
```

Is the value of `code-b` “(\uparrow $\langle exp \rangle$)” or “(\uparrow $\langle exp \rangle$ Event: #<prococedure>)”? While sometimes we would like the latter, we have picked the former, i.e., we drop all associated interception procedures when splicing. This choice requires only limited reconstruction of the semantic model of S^2 . Under this choice, there still exist solutions which keep associated interception procedures of a program object during \downarrow ; for a solution based on RAS, see the discussion in section 9.

4.5 Examples

A little variation on lift (\$). In general, we cannot lift (\$) a procedure or program object [27]. However, if that object has an originator, we can simply return that originator as the result. The following is our interception procedure:⁸

```
(define (lift-interceptor o k) (if (closed? o) (k (originator-of o)) o))
```

We can write a general procedure that takes a program object and returns a liftable version⁹ for it:

```
> (define (make-product-liftable code)                                     ( $\uparrow$  ( $\downarrow$  code)
  Lift: lift-interceptor))
> (define gen-liftable (make-product-liftable ( $\uparrow$   $\langle exp \rangle$ )))
> (define liftable (! gen-liftable))
> ($ liftable)
( $\uparrow$   $\langle exp \rangle$  Lift: #<procedure>)
```

⁸ This scheme fails to work if the object being lifted is not closed (i.e., contains free variables), so we need to check its closedness using the `closed?` internal procedure.

⁹ More precisely, the product object of the program object is liftable.

```

(define code-0
  (↑ letrec code-1 =
    (↑ letrec code-2 =
      (↑ λ x1 f2 f3.
        let x1 = x - x1, x2 = x - x2, x3 = x - x3 in
        (↓ (↓ let x12 = x1 - x2 in
          (↑ let x13 = x1 - x3, x23 = x2 - x3 in
            (↑ f1 * x2 * x3 / (↓ ($ x12 * x13)) -
              f2 * x1 * x3 / (↓ ($ x12 * x23)) +
              f3 * x1 * x2 / (↓ ($ x13 * x23))))))
          AppOp: (code-2-AppOp))
        in code-2
        Int: (code-1-Int))
    in code-1
    Int: (code-0-Int)))

<code-2-AppOp> :
let gen-f-interp = λ().(! code-2), f-interp' = #f in
  λok_.if f-interp'
    then f-interp'
    else (begin (set! f-interp' (gen-f-interp)) f-interp')
<code-1-Int> :
let gen-code-2 = λ().(! code-1), code-2' = #f, x3' = #f in
  λo_.if (equal? x3' x3) then code-2'
    else (begin (set! code-2' (gen-code-2))
      (set! x3' x3) code-2')
<code-0-Int> :
let gen-code-1 = λ().(! code-0), code-1' = #f,
  x1' = #f, x2' = #f in
  λo_.if (equal? x1' x1) and (equal? x2' x2)
    then code-1'
    else (begin (set! code-1' (gen-code-1))
      (set! x1' x1) (set! x2' x2) code-1')

```

Fig. 2. code-0 with automatic respecialization.

Splicing extra code during down. Suppose that `code-a` is bound to a program object “(↑ ⟨*exp*⟩ Event: #<procedure>)”. We can replace (↓ `code-a`) by (↓ `code-a1`), where `code-a1` is the result of evaluating (↑ (! `code-a`)). With this replacement, we kept the program object “(↑ ⟨*exp*⟩ Event: #<procedure>)” intact, including its stage number and its interception procedures, in the context of the down (↓) special form, thus solving the problems in section 5.

```
(define (down-interceptor o k _) (↑ (! o)))
```

A sample usage of `down-interceptor` is as follows:

```

> (define code-a (! (↑ (↑ ⟨exp⟩ Event: proc)
Down: down-interceptor)))
> (define code-b (↑ (↓ code-a)))
> code-b
(↑ (! %o)) ; where %o is bound to “(↑ ⟨exp⟩ Event: #<procedure>).”

```

5 Dynamic Program Regeneration

5.1 Dynamically Respecializing Lagrange Interpolation Function

As motivated in section 1, when the program parameters `x1`, `x2`, `x3` change dynamically, we need to carry out the specialization process again automatically. One more requirement is that we would like to carry out the respecialization process with recomputation of the least number of earlier stages. For example, if `x3` is modified, we should reflect back to stage 1 and recompute `code-2` and `f-interp` successively. Only if `x1` or `x2` has changed, do we need to reflect back to stage 0 and recompute `code-1`, `code-2`, `f-interp` successively.

Fig. 2 shows what modification is needed to `code-0` to enable automatic respecialization using RAS. The modification amounts to adding three pieces of code ((`code-2-AppOp`), (`code-1-Int`), (`code-0-Int`)). With the new `code-0`:

```

> (define x1 0) > (define x2 6) > (define x3 -1)
> (define f-interp (! (! (! code-0))))
> (f-interp 2 -1 11 4)

```

<pre> (define (U code . PcodeL) letrec codes = (↑ (↓ code) Int: ⟨U-Int⟩ Down: down-interceptor) in codes) ⟨U-Int⟩ : let gen-code_{s+1} = λ().(! codes), code'_{s+1} = #f, PL' = #f, λo_.let PL = (map ! PcodeL) in if (equal? PL' PL) then code'_{s+1} else (begin (set! code'_{s+1} (gen-code_{s+1})) (set! PL' PL) code'_{s+1}). (a) </pre>	<pre> (define (W code . PcodeL) letrec codes = (↑ (↓ code) AppOp: ⟨W-AppOp⟩) in codes) ⟨W-AppOp⟩ : let gen-f_{s+1} = λ().(! codes), f'_{s+1} = #f, PL' = #f, λok_.let PL = (map ! PcodeL) in if (equal? PL' PL) then f'_{s+1} else (begin (set! f'_{s+1} (gen-f_{s+1})) (set! PL' PL) f'_{s+1}). (b) </pre>
---	---

Fig. 3. The U and W procedures.

```

-5
> (define x1 1)
> (f-interp 2 -1 11 4) ; code-1, code-2, f-interp are regenerated
-7/5

```

5.2 Automatic Generation of RAS Code

Hand-writing code-0 in Fig. 2 is hard work. Fortunately, we can use the program generation capability of S^2 to automatically generate the RAS code. Here we will develop two code-generation procedures (U and W) that take as arguments a program object and information about which program parameters are in this stage and transform the input program object to a new program object with associated interception procedures that take care of dynamic respecialization. For example, we would like to tell the procedure U that code-0 will specialize with respect to program parameters $x1$, $x2$, and the procedure U returns a new version of code-0 that is enabled to do dynamic respecialization.

The “ U ” procedure. The first procedure (called “ U ”) is for transforming program objects that produce program objects when operated on by “!”, Examples are code-0, code-1, but not code-2 (which produces a procedure object). The formals of U are the program object (code) and a list of program objects (PcodeL) that provide the program parameters. The major task of U (Fig. 3(a)) is to create **Int** interception procedures much like those in Fig. 2. In addition, we also need the **Down** interception procedure as described in section 9.

The “ W ” procedure. The second procedure “ W ” (Fig. 3(b)) is for transforming program objects that produce procedure objects, e.g., code-2 in section 5.1. W is quite similar to U , except that it creates **AppOp** interception procedures.

```

(define codef-0
  (↑ λx1x2.(↑ λx3.(↑ λxf1f2f3.let x1 = x - x1, x2 = x - x2, x3 = x - x3 in
    (↓ (↓ let x12 = x1 - x2 in
      (↑ let x13 = x1 - x3, x23 = x2 - x3 in
        (↑ f1 * x2 * x3/(↓ ($x12 * x13)) -
          f2 * x1 * x3/(↓ ($x12 * x23)) +
          f3 * x1 * x2/(↓ ($x13 * x23))))))))))

```

Fig. 4. Staged generator function program “codef-0”.

Dynamic respecializable Lagrange interpolation in “U” and “W”.

```

(define (U0 code) (U code (↑ x1) (↑ x2)))
(define (U1 code) (U code (↑ x3)))
(define code-0 (U0 (↑ U1 (↑ W (↑ λxf1f2f3.
  let x1 = x - x1, x2 = x - x2, x3 = x - x3 in
  (↓ (↓ let x12 = x1 - x2 in
    (↑ let x13 = x1 - x3, x23 = x2 - x3 in
      (↑ f1 * x2 * x3/(↓ ($x12 * x13)) -
        f2 * x1 * x3/(↓ ($x12 * x23)) +
        f3 * x1 * x2/(↓ ($x13 * x23)))))))))))

```

Compared with the previous program for `code-0` given in Fig. 1, the dynamic respecializable version simply inserts a proper *U* or *W* procedure at each level.

5.3 The “U_f” Procedure

There is an issue with “*U*” and “*W*”. When an object is regenerated, the values of program parameters are computed twice: once for comparison with the memoized old program parameters values, and the second time for regenerating the product object. If the parameter values change between the two computations, it is possible that the product object and the memoized program parameters become inconsistent. Further still, if we compute program parameters only once per interception procedure call, we can also gain some performance benefit.

To solve this problem it is necessary to adopt generator functions that take in parameter values as arguments, and return specialized program objects, which “wrap” yet another more specialized generator function or the target function. The generator functions and program objects form a chain like the following:

$$\text{codef-0} \xrightarrow{!} \text{gen-gen-f-interp} \xrightarrow{(x_1, x_2)} \text{codef-1} \xrightarrow{!} \text{gen-f-interp} \xrightarrow{(x_3)} \text{codef-2} \xrightarrow{!} \text{f-interp}.$$

In the above chain, `codef-0`, `codef-1`, and `codef-2` are program objects, while `gen-gen-f-interp`, `gen-f-interp`, and `f-interp` are functions.

For the Lagrange interpolation example, we can define `codef-0` as in Fig. 4.

In Fig. 5, we show the definition of a procedure, called “*U_f*”, that takes in the program object for the generator function at the head of the chain (i.e., `codef-0`), plus a list of lists of program objects for accessing program parameters at each stage of the chain (i.e., $((px_1, px_2), (px_3))$, where px_i is code for accessing x_i), and creates a stage-0 object `code-0`, which behaves like the one obtained with *U* and *W* in section 5.2:

```

(define (Uf codefs PcodeLLs)
  if (null? PcodeLLs)
    then let fs+1 = (! codefs) in
      (↑ (↓ codefs)
        AppOp : ⟨ Uf-AppOp ⟩)
    else let* PcodeLs = (car PcodeLLs),
      PcodeLLs+1 = (cdr PcodeLLs),
      PL's = (map ! PcodeLs),
      codef's+1 = (apply (! codefs) PL's),
      code's+1 = (Uf codef's+1 PcodeLLs+1) in
      (↑ code's+1
        Int : ⟨ Uf-Int ⟩
        Down : down-interceptor)),

⟨ Uf-AppOp ⟩ :
  λok. fs+1

⟨ Uf-Int ⟩ :
  λo. let PLs = (map ! PcodeLs) in
    if (equal? PL's PLs)
      then code's+1
    else (begin (set! PL's PLs)
      (set! codef's+1 (apply (! codefs) PL's))
      (set! code's+1 (Uf codef's+1 PcodeLLs+1))
      code's+1)).

```

Fig. 5. The “ U_f ” procedure.

```

> (define x1 0)    > (define x2 6)    > (define x3 -1)
> (define code-0 (Uf codef-0 '(, (, (^ x1) , (^ x2)) (, (^ x3)))))
> (define f-interp (! (! (! code-0)))
> (f-interp 2 -1 11 4)
-5
> (define x1 1)
> (f-interp 2 -1 11 4)
-7/5

```

Pay attention to the 2nd and 3rd lines above.

6 Experimental Results

In this section we report on an experiment using our prototype S²RAS interpreter and the performance results obtained.

The application we consider is a staged version of a program that uses a Bayesian Network (BN) to infer the location of an IEEE 802.11 wireless client (e.g., a laptop or an embedded device) from signal quality measures [4]. The input to the locate function (f_{locate}) is a vector of measurements of signal-noise ratios—SNRs (s_1, \dots, s_{N_S}) between the mobile client and N_S access points in the environment, and the function outputs the probability distribution over a discrete set of N_L locations (l_1, \dots, l_{N_L}). Mathematically, the location function computes the following conditional probability distribution:

$$f_{locate}(s_1, \dots, s_N) = \{\Pr(L = l | S_1 = s_1, \dots, S_{N_S} = s_{N_S}), \text{ where } l = l_1, \dots, l_{N_L}\}.$$

The underlying BN is shown in Fig. 6. It (together with its conditional probability tables) is treated as a program parameter, which can (but only infrequently) change over time. There are three cases when the program parameter can change: a) more access points are installed (this affects the BN topology); b) more locations are added to be inferred; c) conditional probability tables change (when a location is resampled). We will consider the last two cases.

Both the stageless and the staged program use an algorithm constructed via the procedure described in [15] according to Fig. 6.¹⁰ The complexity of the algorithm is $O(N_L N_S^2 N_{SNR}^{N_S})$, where N_{SNR} is the number of SNR levels. The results are shown in Fig. 7, where $1 \leq N_L \leq 10$, $N_S = 4$ and $N_{SNR} = 2$.

¹⁰ In other words, the stageless version is already specialized to the two-layer format.

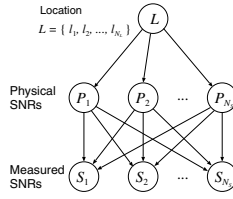


Fig. 6. Bayesian Network for inferring location.

The first three graphs Fig. 7(a-c) are based on measurements of three versions of f_{locate} : stageless version, staged version running with RAS disabled, and staged version running with RAS enabled. Note that no specialization cost is included for both staged versions, i.e., we first obtain a specialized f_{locate} and then measure how much time is needed to call it once.¹¹

- Fig. 7(a) shows the time complexity curves for three cases. The time complexity is linear in N_L as expected.
- Fig. 7(b) shows the *speed-up* ratio of the staged (specialized) function to the stageless function. The staged version runs about 11 ~ 12 or 8 ~ 11 times faster (than the stageless version) with RAS disabled or RAS enabled.
- Fig. 7(c) shows the *overhead* of enabling RAS in the interpreter:

$$overhead = (time_{staged_with_RAS} - time_{staged_w/o_RAS}) / (time_{staged_w/o_RAS}).$$

The curve shows that the overhead for $N_L = 1$ is about 30%, while the asymptotic limit for the overhead as $N_L \rightarrow \infty$ is below 10%.

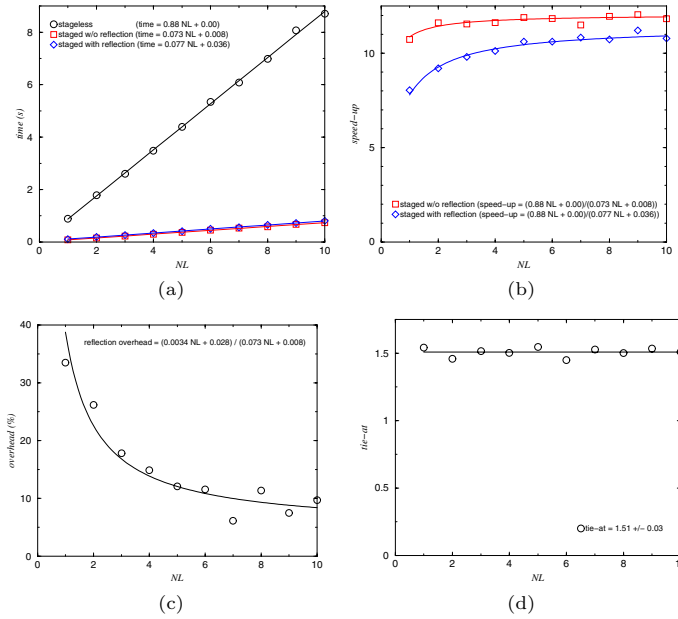
Fig. 7(d) shows the “tie” frequency (at which program parameter changes) when the staged program with RAS-based dynamic respecialization have the same performance with the stageless program. This frequency is measured using the quantity “*tie-at*”, which is the number of times f_{locate} is called before the program parameter changes. The experiment resulted in a tie-at of ~ 1.5 .

7 Related Work

The goal of our approach is mainly making software adaptable while minimizing runtime overhead caused by computations needed for adaptation. Software components are always running as specialized programs adapted to the current program parameter values, and dynamic respecialization is carried out by need if program parameter values change.

[14] describes a dynamic adaptation scheme in which software components may be recompiled using a dynamic compiler. The recompilation will generate (re-)optimized target code according to profiling data and/or the new configuration. This work differs from ours mainly in that the software component is only optimistically compiled but not specialized to the new configuration. In addition,

¹¹ The data points shown are averaged over calling f_{locate} 100 - 1000 times.



Note: The three lines in (a) are obtained using linear regression on the experiment data points. The equations and curves in (b) and (c) are based on linear coefficients from linear regression in (a), plotted together with data points computed directly from experiment data.

Fig. 7. Performance graphs.

no detail is given about how the recompilation is triggered, and how to continue the computation of the software component that is recompiled.

Tempo supports runtime specialization of C functions [6], and a multi-stage extension has been published [17]. The emphasis of their work is on producing efficient runtime specializers automatically using program analysis. These runtime specializers are invoked by programmers when runtime invariant program parameters become known, i.e., automatic respecialization is not addressed. DyC[12, 13] also advocates runtime specialization, where the programmer annotates regions of the programs that should be dynamically specialized and variables for which to specialize the region. To avoid respecialization of such regions every time they are executed, DyC provides automatic caching, reuse, and reclamation of dynamically generated code. This bears similarity to our approach, however our approach is more general not only in that program objects are first class and there are arbitrarily many stages, but also in that RAS is a tool that the programmer can use to implement her own caching mechanism.

We have only implemented an interpreted S²RAS, but the idea of RAS can be applied to dynamic compilation systems. Dynamic compilation can be viewed as a special program generation stage, which generates a stage-(s + 1) program object in machine code. To enable RAS, we need to insert some code that activates the RAS mechanism at the beginning of the generated machine code.

8 Conclusion and Discussion

We have studied a metaobject protocol—reflection across stages (RAS)—in multi-stage program generation (MSPG) systems, especially intercessory constructs for reflecting to earlier-stage program objects. The reflection constructs provide a nice solution to the problem of regeneration of MSPG software with respect to dynamic change of program parameters. The solution is encapsulated in two sets of procedures ($\{U, W\}$ and $\{U_f\}$), which are very easy to use.

It is not surprising that computational reflection techniques worked well in solving the dynamic regeneration problem. However, the full potential of RAS is yet to be explored. This includes exploring the design space of reflection constructs as well as other application areas of RAS. For example, we think that RAS techniques may apply well to more dynamic and flexible staging design (see Sec. 1.2) as well as debugging and profiling multi-stage programs.

On the other hand, although the properties of program objects (the enclosed expression, the interception procedures, its stage number and originator) are immutable in this work, we have taken semi-object-oriented views on them. The originator program object is like the class of its product objects as it controls their behavior. Likewise, a program object with interception procedures is like a subclass of a program object without interception procedures. This motivates us to develop a truly object-oriented MSPG language, where program is a special kind of (meta)object, like code blocks in Smalltalk. In such a language, there exists a third meta-relationship—originates-from—between objects besides instance-of and inheritance relationships. We may aspire to align functionality more aesthetically along the three sets of meta-relationships.

On the practical side, programmers may not need to be aware of the basic concept of RAS in order to benefit from it. For example, she might only need to know U_f . This is much like the case with other basic concepts in programming language theory. For example, programmers may use exceptions and coroutines without understanding the more basic concept of continuations.

Acknowledgments. We are grateful to Dr. Walid Taha and Dr. Stott D. Parker for suggestions of improvement on an earlier draft.

References

1. A. Bawden. Quasiquotation in lisp. In *Proc. of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 4–12, Jan. 1999.
2. A. Bawden and J. Rees. Syntactic closures. In *Proc. of the 1988 ACM Symposium on Lisp and Functional Programming*, pages 86–95, 1988.
3. C. Calcagno, W. Taha, L. Huang, and X. Leroy. A bytecode-compiled, type-safe, multi-stage language. draft, Nov. 2001.
4. P. Castro, P. Chiu, T. Kremenek, and R. Muntz. A probabilistic location service for wireless network environments. In *Proc. of the 2001 Ubiquitous Computing Conference*, pages 18–34, Sept. 2001.

5. W. Clinger and J. Rees. Macros that work. In *Proc. of the 1991 ACM Conference on Principles of Programming Languages*, pages 155–162, 1991.
6. C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Proc. of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, 1996.
7. F.-N. Demers and J. Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *Proc. of the IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pages 29–38, Aug. 1995.
8. J. des Rivières and B. C. Smith. The implementation of procedurally reflective languages. In *Proc. of the 1984 ACM Symposium on LISP and functional programming*, pages 331–347, 1984.
9. D. R. Engler, W. C. Hsieh, and M. F. Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Proc. of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 131–144, Jan. 1996.
10. S. Ganz, A. Sabry, and W. Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. In *Proc. of the 25th ACM Symposium on Principles of Programming Languages*, Sept. 2001.
11. R. Glück and J. Jørgensen. Efficient multi-level generating extensions for program specialization. In *Programming Languages: Implementations, Logics and programs (PLILP'95)*, volume 982 of *Lecture Notes in Computer Science*, pages 259–278. Springer, 1995.
12. B. Grant, M. Mock, M. Philipose, C. Chambers, and S. Eggers. Annotation-directed run-time specialization in C. In *Proc. of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 163–178, June 1997.
13. B. Grant, M. Mock, M. Philipose, C. Chambers, and S. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1-2):147–199, Oct. 2000.
14. B. Harrison and T. Sheard. Dynamically adaptable software with metacomputations in a staged language. In W. Taha, editor, *Proc. of the Second International Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG 2001)*, volume 2196 of *Lecture Notes in Computer Science*, pages 163–182. Springer, Sept. 2001.
15. C. Huang and A. Darwiche. Inference in belief networks: A procedural guide. *International J. of Approximate Reasoning*, 15(3):225–263, Oct. 1996.
16. G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, Cambridge, Massachusetts, 1991.
17. R. Marlet, C. Consel, and P. Boinot. Efficient incremental run-time specialization for free. In *Proc. of the ACM SIGPLAN Conference on Programming language Design and Implementation*, pages 281–292, 1999.
18. M. Mock, C. Chambers, and S. Eggers. Calpa: A tool for automating selective dynamic compilation. In *Proc. of the 33rd ACM SIGMICRO Symposium on Microarchitecture*, pages 291–202, 2000.
19. E. Moggi, W. Taha, Z. E.-A. Banaissa, and T. Sheard. An idealized MetaML: Simpler, and more expressive. In *Proc. of the 1999 European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 193–207. Springer, Mar. 1999.
20. F. Nielson and H. R. Nielson. *Two-Level Functional Languages*. Cambridge Tracts in Theoretical Computer Science **vol. 34**. Cambridge University Press, 1992.

21. M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek. 'C and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21(2):324–369, March 1999.
22. M. Shields, T. Sheard, and S. P. Jones. Dynamic typing as staged type inference. In *Proc. of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 289–302, Jan. 1998.
23. B. C. Smith. Reflection and semantics in Lisp. In *Proc. of the 11th annual ACM symposium on Principles of programming languages*, pages 23–35, 1984.
24. W. Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, Nov. 1999.
25. W. Taha. A sound reduction semantics for untyped CBN multi-stage computation. In *Proc. of the 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 34–43, Jan. 2000.
26. W. Taha, Z.-E.-A. Benaïssa, and T. Sheard. Multi-stage programming: Axiomatization and type safety. In *Proc. of the 25th International Colloquium on Automata, Languages, and Programming*, pages 918–929, July 1998.
27. W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Proc. of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 203–217, June 1997.

A Overview of Staged Scheme (S²)

The multi-stage programming language MetaML originated from research on two-level and multi-level languages in the partial evaluation community [20,11]. However, a reader who is familiar with quasiquotations in Lisp dialects [1] will find that there is a close relationship between the staging annotations introduced in MetaML and the metaprogramming facilities (quasiquotations and *eval*) in Lisp dialects. Since we have transplanted the staging annotations from MetaML into S², we must discuss the difference and relative benefits between MetaML-like annotations and Lisp-like quasiquotations. This appendix is a feature-oriented discussion of the above issue.

A.1 Typing Program Objects

One of the strengths of MetaML is that it provides a static type system for multi-stage programs. The static type system ensures that a correctly typed multi-stage program can never produce erroneous program objects in any later stage. When transplanted to S², the static aspect of the typing system is lost, but program objects are dynamic typed.

For example, consider the following two interactions with Scheme and S²:

```
> (define code-scheme '(+ ,(+ 1 2) 4)) ; code-scheme = "(+ 3 4)"
> (eval code-scheme)
```

7

```
> (define code-s2 (↑ (+ (↓ ($ (+ 1 2))))) ; code-s2 = "(↑ (+ '3 '4))"
> (! code-s2)
```

7

Despite the similarity, the biggest difference between the `code-scheme` and `code-s2` is that the former is an object of type “list”, while the latter is an object of type “ \uparrow ”, which signifies a program object.¹² Therefore, the following are incorrect programs in S^2 :

```
(! code-scheme)
(car code-s2)
```

Similarly, because of S^2 ’s typing system, the result “3” of evaluating `(+ 1 2)` needs to be lifted (\$) to “(\uparrow 3)” before it can be spliced (\downarrow) into a program object of type “ \uparrow ”.

The major benefit of having dedicated types for program objects is like adding types to an untyped language, i.e., some program errors can be captured by either static or dynamic type checking. Another less obvious benefit is that program types can be used as guidelines when writing multi-stage programs. For example, it will be much more difficult to come up with the correct U , W , and U_f procedures in Sec. 5 without knowing their types first.

The disadvantage of having program types is again like any typed language, i.e., the language becomes more restrictive. For example, one cannot analyze the internal structure of a program object in S^2 because no such operation is defined on objects of program type, while program objects are just data objects, which can be manipulated more freely, in Scheme.

A.2 Lexical Scoping in Multi-stage Programs

Scoping has been a non-trivial problem with metaprogramming systems, e.g., macros [2,5]. MetaML and S^2 have a pleasant lexical scoping scheme that makes variables bound in an earlier-stage computation available to later-stage computation as long as they are used within the same static lexical scope where they are defined. This scoping scheme is called cross-stage persistence.

For example, consider the following two interactions with Scheme and S^2 respectively:

```
> (define o-scheme (let ((x 'a)) 'x)) ; o-scheme = "x"
> (eval o-scheme)
Error: variable x is not bound.

> (define o-s2 (let ((x 'a)) ( $\uparrow$  x))) ; o-s2 = "( $\uparrow$  %x->a)"
> (! o-s2)
a
```

The two x ’s in defining `o-scheme` refer to two different variables at two different stages, although they appear in the same lexical scope. The first x is bound to an atom `a` but has nothing to do with the second x , which is not bound in the later stage unless x happens to be bound in the toplevel environment. The S^2 version treats the two x ’s in defining `o-s2` as essentially the same variable—a variable which is bound in the first stage (when the `let` special form is computed) but is still available to the second stage (when `o-s2` is evaluated). We say that x

¹² The type of `code-s2` would be “ \uparrow of `int`” if S^2 were statically typed.

is a cross-stage persistent variable which is bound to an atom **a**. Cross-stage persistent variables are often denoted like **%x** according to MetaML tradition.

While there may be cases when cross-stage persistence is not desirable, it often makes writing multi-stage programs a lot easier for the same reason that lexical scoping makes writing correct programs easier. This is especially true when the number of stages are greater than two. In contrast, nested quasiquotations are often a difficult art to learn in Lisp dialects.

Author Index

- Aldawud, Omar 189
Attardi, Giuseppe 50
- Back, Godmar 66
Bader, Atef 189
Balat, Vincent 78
Bapty, Ted 236
Barbeau, Michel 93
Bednasch, Thomas 156
Bordeleau, Francis 93
Brichau, Johan 110
Butler, Greg 128
- Chin, Wei-Ngan 140
Cisternino, Antonio 50
Czarnecki, Krzysztof 156
- Danvy, Olivier 78
Douence, Rémi 173
- Eisenecker, Ulrich 156
Elrad, Tzilla 189
- Fradet, Pascal 173
- Glenstrup, Arne J. 1
Gokhale, Aniruddha 236
Gray, Jeff 236
- Hu, Zhenjiang 140
- Jones, Neil D. 1
- Karsai, Gabor 32
Kiselyov, Oleg 202
Krüper, Florian 252
- Lee, Chin Soon 218
- Mens, Kim 110
Muntz, Richard R. 316
- Neema, Sandeep 236
Noga, Markus 252
Noyé, Jacques 283
- Piquer, José 283
- Saraiva, João 268
Ségura-Devillechaise, Marc 283
Südholt, Mario 173
Sztipanovits, Janos 32
- Tanter, Éric 283
- Unger, Peter 156
- Visser, Eelco 299
Volder, Kris De 110
- Wang, Zhenghao 316