Lecture Notes in Computer Science 2177

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

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

Greg Butler Stan Jarzabek (Eds.)

Generative and Component-Based Software Engineering

Second International Symposium, GCSE 2000
Erfurt, Germany, October 9-12, 2000
Revised Papers

Springer

Series Editors

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

Volume Editors

Greg Butler
Concordia University, Department of Computer Science
1455 de Maisonneuve Blvd West, Montreal, Quebec H3G 1M8, Canada
E-mail: gregb@cs.concordia.ca

Stan Jarzabek
National University of Singapore
School of Computing, Dept. of Computer Science
Singapore 117543
E-mail: stan@comp.nus.edu.sg

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Generative and component based software engineering : second international symposium ; revised papers / GCSE 2000, Erfurt, Germany, September 9 - 12, 2000. Greg Butler ; Stan Jarzabek (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Tokyo : Springer, 2001
(Lecture notes in computer science ; Vol. 2177)
ISBN 3-540-42578-0

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

ISSN 0302-9743
ISBN 3-540-42578-0 Springer-Verlag Berlin Heidelberg New York

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

http://www.springer.de

Printed in Germany

Typesetting: Camera-ready by author
Printed on acid-free paper SPIN: 10840460 06/3142 5 4 3 2 1 0

Preface

These are the proceedings of the second symposium on Generative and Component-Based Software Engineering that was held in Erfurt, Germany, on October 9–12, 2000, as part of the Net.Object Days conference. The GCSE symposium was born in 1999 at the Smalltalk and Java in Industry and Education Conference (STJA), the precursor to the Net.Object Days conference. The GCSE symposium grew out of a special track on generative programming that was organized by the working group "Generative and Component-Based Software Engineering" of the "Gesellschaft für Informatik" FG 2.1.9 at STJA in the two years 1997 and 1998. The GCSE symposium covers a wide range of related topics from domain analysis, software system family engineering, and software product lines, to extendible compilers and active libraries.

The second GCSE symposium attracted 29 submissions from all over the world. This impressive number demonstrates the international interest in generative programming and related fields. After a careful review by the program committee, 12 papers were selected for presentation. We are very grateful to the members of the program committee, all of them renowned experts, for their dedication in preparing thorough reviews of the submissions.

Special thanks go to Elke Pulvermüller, Andreas Speck, Kai Böllert, Detlef Streitferdt, and Dirk Heuzeroth, who continued the tradition from GCSE'99 and organized a special conference event, the Young Researchers Workshop (YRW). This workshop provided a unique opportunity for young scientists and Ph.D. students to present their ideas and visions of generative programming and related topics and to receive thorough critique and feedback from senior experts in the field.

We are also indebted to the keynote speakers and tutorial presenters, Paul Bassett, Doug Smith, and Michel Tilman, for their contribution to GCSE 2000. Fortunately, we have papers from two of the three keynote speakers for these proceedings. Unfortunately, there is no paper for the keynote talk on *"Software Development by Refinement"* given by Doug Smith of the Kestrel Institute. Finally, we wish to thank all who put in their efforts and helped to make this symposium happen, especially the authors and the Net.Object Days organizers.

We hope you will enjoy reading the GCSE 2000 contributions, and invite you to contribute to future symposiums.

October 2000

Greg Butler
Stan Jarzabek

Organization

GCSE 2000 was co-hosted with Net.Object Days 2000 and organized by the Working Group "Generative and Component-Based Software Engineering" of the German "Gesellschaft für Informatik".

Program Chairs

Greg Butler (Concordia University, Canada)
Stan Jarzabek (National University of Singapore, Singapore)

Program Committee

Mehmet Aksit (University of Twente, The Netherlands)
Paul Bassett (Netron Inc., Canada)
Don Batory (University of Texas, USA)
Ira Baxter (Semantic Designs, Inc., USA)
Manfred Broy (Technical University of Munich, Germany)
Jim Coplien (Bell Labs, USA)
Krzysztof Czarnecki (DaimlerChrysler AG, Germany)
Jin Song Dong (National University of Singapore, Singapore)
Ulrich Eisenecker (University of Applied Sciences, Kaiserslautern, Germany)
Harald Gall (Technical University of Vienna, Austria)
Kyo Kang (Pohang University of Science and Technology, Korea)
Rudolf Keller (Université de Montréal, Canada)
Peter Knauber (Fraunhofer Institute for Experimental Software Eng., Germany)
Kung-Kiu Lau (University of Manchester, UK)
Hafedi Mili (University of Quebec, Canada)
Gail Murphy (University of British Columbia, Canada)
Pavol Navrat (Slovak University of Technology, Slovakia)
John Potter (University of New South Wales, Australia)
Dieter Rombach (University of Kaiserslautern, Germany)
Clemens Szyperski (Microsoft Research, USA)
Todd Veldhuizen (Indiana University, USA)

Organization Committee

C. Cap (University of Rostock)
K. Czarnecki (DaimlerChrysler AG)
T. Dittmar (Daedalos Consulting GmbH)
B. Franczyk (University of Essen)
U. Frank (University of Koblenz-Landau)

M. Goedicke (University of Essen)
M. Jeckle (DaimlerChrysler AG)
H. Krause (Transit)
F. Langhammer (Living Pages Research)
M. Lehmann (Sun Microsystems)
B. Lenz (Transit)
C. Müller-Schloer (University of Hannover)
I. Philippow (Technical University of Ilmenau)
H.-W. Six (Fern University of Hagen)
R. Unland (University of Essen)
G. Vossen (University of Münster)
M. Weber (University of Ulm)
H.G. Weißenbach (HW Consulting)

Table of Contents

Mixin-Based Composition and Metaprogramming

The Theory and Practice of Adaptive Components

Paul G. Bassett

Netron, Inc., Canada
pbassett@netron.com

Abstract. It is well known that modifying software by hand, especially someone else's, is not only slow and tedious but so notoriously error-prone that we strive for components that never need changing — black-box building blocks. Unfortunately, we lack the omniscience required to engineer such parts, especially when our problem domains are ill-defined. A promising alternative is to make each component *adaptive* — a simple mark-up language (e.g., frame technology) converts each component's properties (i.e., granularities that are meaningful within a given domain) into a default value that can be overridden or extended by other components in a rapid yet reliable manner. Adaptive components constitute a gray-box strategy — black when their properties fit the context as is, otherwise white. The approach works by automating the tedious and error-prone aspects of construction and modification, while localizing all the unique properties of an object or program in its root component. Each root exercises complete control over the structure and content of the code that its hierarchy of components emits. Such custom roots constitute only 5–15% of a typical system. In other words, 85–95% reuse is the norm. Along with independently gathered statistical evidence of efficacy in constructing and evolving business systems of up to 10 million lines of code, the further implications of adaptive components for software engineering are presented.

1 Introduction

Good morning ladies and gentlemen. It gives me great pleasure to be able to talk to you about my favorite software engineering topic: adaptive components. In fact, I had trouble writing this address, as there are so many ways to approach them. I decided to introduce them by tracing their origins and current status. Having set the stage, we will see adaptive components as they have been realized in *frame technology*, discussing how they change the way we think about domain modeling and componentization. By then you will be wondering how well formal adaptivity works. I'll present independently gathered statistics that show savings in time and effort that many find too good to be true! So what's the catch? I'll close with my thoughts on what it takes for adaptive components to find their way into general practice.

G. Butler and S. Jarzabek (Eds.): GCSE 2000, LNCS 2177, pp. 1-14, 2001.

"Call/Return" mechanisms have been built into computer hardware for more than 40 years. Both macros and subroutines have been standard features of assembly languages for almost as long. Software engineers have been generating code and building systems from components almost since the inception of programming.

In principle, generators, subroutines, and macros all have the potential to simplify program understanding and reduce coding effort. In practice, generators have had a checkered history, the reasons for which I shall address later. Let me now compare and contrast macros with subroutines.

Curiously, macros fell into disuse as higher level languages emerged. Only a handful of 3GLs e.g., C, contained them, and even fewer 4GLs. Subroutines, on the other hand, flourished in more and more forms — function calls, overloaded operators, coercion, recursion, reentrant, call-by-name, call-by-value, and object methods, to name a few. (For the sake of simplicity, I shall use the term subroutine to refer to all of the above.)

2 Duality

I've pondered long and hard about why subroutines succeeded while macros languished.

One technical reason is that macros typically suffer from the so-called "global variable" problem. That is, macros can overwrite each other's variables, making macros unpredictable and error-prone. Global variables also afflicted programs and subroutines, inspiring the evolution of block-structured programming languages. Macro languages could easily scope their variables to nested blocks too, but for some reason most don't.

Another technical objection, at least in the early days, was that memories were small and expensive, causing people to avoid in-line macro expansions. Today that reason is totally obsolete.

A more insidious reason is the myth that macros and subroutines are competitive ways to do much the same thing, and after all, don't programmers already have too many ways hang themselves? In fact they are complementary, not competing mechanisms for implementing components.

While it is true that both are parameter-driven, macros operate only at construction time to emit program structures, and subroutines operate only at run time to emit data. Macros operate at the component meta-level: components that construct components, including, of course, subroutines.

It's hard to overstate the significance of this fact. To give you one facet of its power, consider that one macro can give rise to an infinity of polymorphs, depending on its parameter inputs. So, rather than drowning in a sea of look-alike functions, systems can be architected, built, and evolved with small set of normalized basis *functionals*.

The myth is not only false but it blinds us to a new universe of opportunity. What is that universe? In our business, when the word "architecture" is

heard, we automatically think of the structure and modularization of our run-time systems. The issues driving the architecture have to do with modularity, functionality, efficiency, and ease-of-use in distributed environments. But at the meta-level, there is a *dual* set of architectural questions having to do with minimizing complexity while optimizing generality and adaptability. Given that a construction architecture must implement its run-time counterpart, the former is even more important to a well-engineered system than the latter. Yet hardly anyone pays attention to it! Why has the build/run duality, a source of so much insight and simplifying power, gone largely unrecognized?

3 Softness

Before proceeding, further into construction-time, we need to pause and ask ourselves a very basic question: Why do we solve problems using software at all? The case could be made that, especially with the advent of chip foundries, all could and should be burned into chips. Of course, a moment's thought supplies the answer. The *raison-d'être* of keeping our solutions in software is *ease of modification*. Our first version of a system often merely tells us that we have solved the wrong problem. Without software most of the systems that power modern society would be beyond our grasp simply because it's totally infeasible to build special purpose machines for every version that such systems must evolve through.

Given the fundamental importance of modifiability it is ironic that we rarely take the time to explicitly engineer our systems so that they conserve this property. As a result, systems once soft become brittle to the point that decrees are issued that no one is to modify them on pain of dismissal. Why have we ignored a meta-problem with such expensive consequences?

Another meta-problem is the highly non-linear nature of the beast. Systems are said to behave linearly if the effect of a change is proportional to its size. But with software, change a single symbol and the result could range anywhere from totally benign to the crash of 200-million dollar Martian spacecrafts, or worse.

Last but not least, billions of years of evolution have conditioned us to expect everything to be made from concrete, stable, parts. We continue to design software systems as if they were cars and TV sets. Indeed, the OO paradigm perpetuates the myth that the world can be built from immutable components — Lego blocks.

Before I raise too many hackles, let me explain why I rail against this attitude. Problem domains come in many flavors. When the domain is both well understood and stable, it seems natural to create cost-effective, efficient solutions in terms of fixed building blocks. Such domains are constrained by the laws of physics, laws that are well understood, true for all space and time, and unbreakable. Statistics routines, embedded systems, and power-plant control systems are examples. In these domains, ease of modification is a non-issue and the software is often burned into chips.

But most interesting domains are not so well behaved. Typically, the requirements are conflicting, poorly understood by the stakeholders who also do not share a common understanding of them, and they change with time, sometimes radically. These issues pertain to the much larger category of *ill-defined domains*. Business applications are certainly not constrained by the laws of physics; most of the software on the web, and a myriad of other domains too numerous to mention all exhibit engineering degrees of freedom not available in the physical world.

As we explore and experiment with what works and what doesn't, ill-defined domains evolve into better-defined ones. Commodity packages supercede once-custom software for those domains. We take them for granted and stand on their shoulders, so to speak, so that we can tackle new the ill-defined domains inspired by our wealth-creation fantasies. And so the cycle repeats.

But packaged systems never completely stabilize. As one version succeeds another, and as product lines are created for different market niches, they need to be engineered for cost-effective change.

None of this is new to anyone, so why have we done such a poor job of engineering our systems for perpetual ease-of change? We continue to believe software should be built from immutable building blocks because, I contend, evolution has conditioned us to see the world as being built that way. The multi-dimensional universe of software structures possesses inherently new engineering degrees of freedom, which we are naturally slower to grasp.

Reinforcing this collective blind spot is the deep psychological need to believe that only we are capable of making the fussy changes that software needs. On the other hand, we are so poorly equipped to make them reliably that we've institutionalized the idea that because we can build anything from fixed parts, and therefore we should do so.

I have a simple reductio ad absurdum rebuttal. Just two fixed parts are sufficient to build all software — 0 and 1. This is about as sensible as manufacturing cars from atoms. With atomic force microscopes we can attempt it, but should we?

The engineering issue concerns how to model the "natural" graininess of any domain with corresponding parts. Here "natural" means allowing us to highlight the important properties and ignore the irrelevant ones. This strategy leads to a complexity-minimized vocabulary for thinking about and designing system architectures. When a domain is well defined, its stability permits (but does not require) fixed parts. To the extent that it is ill defined, the parts should be adaptable to unforeseen requirements, reliably and without undue increases in complexity. The good news is that *adaptive components* not only make this is possible, but they benefit well-defined domains too!

4 Context Sensitivity

To prepare for the meat of adaptive components, we need to discuss context freedom and context sensitivity. These are relative terms, spanning a spectrum

of possibilities. At run time, the smaller and simpler the assumptions that a component needs to make about its interface, the more contexts it will satisfy and hence fit into. Hence the more context-free and reusable it is. For context sensitive parts, the converse is true. To a first approximation, context freedom is a nice property to have.

The Lego block metaphor implies that all building blocks should be context free. But there is a problem. When a collection of such parts come together in a specific context they typically clash because they are literally ignorant of each other. Especially when components originate from different contexts and authors, the probability quickly jumps to near certainty that they will make inconsistent assumptions about mundane things like data formats and units of measure.

Is this failure to fit into a new context the fault of the context-free component? Not usually. Rather, it is the mutual interaction of the components in a given context that is the source of the difficulties, not the individual components. This is why subclassing in OO is so rampant and why libraries of thousands of polymorphic classes arise, drowning us in seas of lookalikes.

A variant of this strategy is to write additional "work around" code to undo, redo, and/or extend the existing functionality, but without modifying the original component or creating lookalikes. The disadvantages are several including: (a) more space (code), (b) more time (execution cycles), (c) more complexity (both in tortuous extra logic and in the fact that it is dislocated from the logic of the components with which it naturally belongs).

On the other hand, we seldom modify the original component to suit the new context because we are likely to give ourselves enormous retrofit headaches.

So our wish to believe that the world can be made out of Lego blocks puts us on the horns of a dilemma.

There is an analogy that illuminates the real issue. Imagine you are in an auto plant watching cars being assembled. Suppose engineers came up with parts such as a fender that could be adapted to fit any make or model of car as each came down the line. Clearly this would cause a wonderful collapsing of the parts inventory and of the complexity of managing them all. But alas, it is a silly supposition; physical parts simply do not have the needed flexibility. But in software engineering, adaptive engineering degrees of freedom exist because the parts are non-physical. Such adaptive components are a very natural thing to have in software engineering, and the attendant benefits are large, as we shall see.

Unfortunately, our current practices are still rooted in the physical paradigm, leading to the above dilemma. We make actual parts rather than meta-parts. By managing our much-reduced inventory of meta-parts we avoid the headaches and shrink software projects dramatically. Strong claims. I now explain how they work, and conclude with evidence of how well they work.

5 Adaptive Components Defined

For any information domain, an adaptive component *is a formal embodiment of an* archetype *together with* rules *for combining (sub)component-archetypes into context-specific structures and/or functions.* — Webster's Dictionary defines an *archetype* as "the original pattern or model from which all other things of the same kind are made."

I have characterized adaptive components in this way to emphasize that they have applications to any domain for which a symbolic language can be used to describe its contents. For example, manuals have been assembled from adaptive components that explain how to use a software product that works a little differently, depending on which computer platform it is installed. The core functionality of the product is the same, but the user interface must adopt the look and feel of the appropriate operating system. So platform-specific components contain the information needed to adapt the core documentation components and each manual looks like it was written especially for its platform. While the statements to follow apply to this more general context, my focus will be the construction and evolution of software systems, because that is what I know best.

The archetypes to which the definition alludes are the models of the domain's entities and operations. The more generic archetypes can be thought of as generic roots of nouns and verbs in a domain-specific vocabulary. The more specific ones contain the modifiers — adjectives, adverbs, sentence agreement rules, prepositions, etc. — that combine with the root concepts to form words, sentences, paragraphs, sections, chapters (e.g., executables), and documents (e.g. systems).

Creating an archetype is not as hard as it may sound. Given a specific problem to solve, your solution can serve as your first cut at the archetype for all similar problems. You strive for a solution that typifies the others, then parameterize it to be adaptable and to adapt. Alternatives may turn out to be needed for a given class of problems, depending on the context. This is clearly an imperfect exercise. The good news, as we shall see, is that the facilities of adaptive components are forgiving of many errors of omission and commission.

Technically, adaptive components are a kind of macros that "mark up" underlying texts. We are familiar with such markup languages as HTML and XML. XML in particular has the virtue of defining a standard meta-syntax independently of the actual syntax and semantics of the underlying text. It does this by embedding the text with a set of nested, labeled "brackets". An XML-compliant sample might appear as:

```
<patient>
   <name>Joe Blow</name> <drug-allergy>penicillin</drug-allergy>
</patient>
```

By parsing the embedded markups "browsers" can variously process bills, analyze patient medical records, update databases, or do whatever. The meta-notation is readable — it uses angle brackets, ampersands and semi-colons to

separate markup text from regular text. It structures the contents of any document as a tree, rendering it block structured, straightforward to parse, and to process. Many different processors can operate on the same text, say to display it in a graphical form (hiding the markup notation), or to compile it.

I use *frame technology* to illustrate one of many ways to implement adaptive components. For more information, see my book [1]. It consists of a frame (macro) processor (FP), operating on *frame* syntax, which was invented before XML but easily translated into it. As you would expect, each frame is one adaptive component — a macro of marked up text. The text typically is a 3GL (e.g. COBOL, C) but could be a language to capture requirements, or a high level modeling language, or, as I've said, just about any text.

The mark ups form nested tree structures, both within and among frames. — A frame library actually forms what in mathematics is called a semi-lattice, under the "adapt" partial ordering. FP processes each semi-lattice as if it were a tree of subassemblies — a component that has multiple parents is, in effect, cloned, one per parent. So with that understanding, I can refer to a frame hierarchy or semi-lattice as a tree. — FP begins with a root frame, called a *specification frame*, and traverses its frame-tree in natural (depth first, left-to-right) order. By carrying out the mark-up rules, FP assembles one module of text per specification frame. The emitted text should conform to the syntax of the marked up language, e.g., if the output is source-code, it should be compileable without errors. While the frames are strictly read only inputs to the FP, it is natural to pretend the frames are adapting each other, and in what follows that fiction is a useful shorthand.

Frame trees stratify contexts, the specification frame being completely context specific (sensitive), and the leaf components being the most context-free. Each frame, being the root of a subtree, localizes the context information needed to achieve a seamless integration of the elements of that subtree. What follows outlines how frame technology implements this recursive idea.

6 Handling Unpredictable and Unique Variability

The main problem that adaptive components seek to solve is coping with arbitrary and unknowable change. Much research in domain analysis focuses on identifying the commonalities and the variabilities inherent in our domains prior to designing our systems. Adaptive component engineers take the position that much of this analysis is a waste of time. Why? Adaptivity makes it much easier and faster to assume that everything is potentially variable — it's just that our ill defined domain has yet to reveal the specifics. Actually building a good first approximation is the fastest way to zero in on non-obvious variabilities.

In spite of not being omniscient we can still complete our projects on time and within budget because adaptive components are engineered to tolerate our ignorance. Our building blocks can incorporate our improving understanding, usually without invalidating much of what went before. Sound like magic? Okay it's time to show how such components work. (I apologize for having to introduce

some technical vocabulary. Hopefully, your reward for persevering will be a better insight into interesting engineering issues.)

A frame's overall structure is simple:

```
<frame> name <body> frame-text </body> </frame>.
```

Frame-text is a mixture of plain and marked-up text. Plain text encodes anything that is currently not known or anticipated to be variable, and FP emits it as-is. Let's turn our attention to how the various mark-ups deal with variability.

There are two kinds of parameter assignment. The first looks like this:

```
<replace> name <by> expression </by> </replace>
```

name is the name of a parameter but may itself be an **expression**. An **expression** can be a combination of literals, `<replace>` parameter references, and arithmetic expressions that together evaluate to a literal text string. If, after evaluating its **name**, a given `<replace>` is allowed to alter the `<replace>` symbol table — which records the current assignments of parameters (symbols) and manages their scopes (see next paragraph) — then its **expression** is evaluated, and the result is entered into that table.

`<replace>` is scoped by the following "global overrides local" rule: Values assigned to parameters inside frame F cannot be changed by any of F's descendent frames. E.g., if F has set P to value V then V will replace all references to P within the subtree rooted in F. When F exits (i.e., FP finishes processing it), any parameters that F was able to `<replace>` become undefined. In effect, for F to set P, it must be undefined when F is invoked, and all other frames' attempts to `<replace>` P are ignored until F exits.

This scoping rule is not found in most languages and has several noteworthy implications for adaptive components.

- Whenever context-free (local) default values fit the contexts of its more context-sensitive (global) ancestor frames, those parameters need not be overridden. In effect, each frame `<replace>`s the least number of parameters sufficient to define its own context — i.e., ensure that itself and its component frames result in a seamless integrated whole.
- Conversely, this is the "same as, except" principle in action. When you want to reuse a component subassembly, highlight the exceptions in the embedding context; the rest stays the same.
- A frame is blissfully unaware that the contexts into which it is being embedded may override some of its parameters. This allows a context-free component to be evolved, rather than having to be created in mature form. How? Start with plain text that suits a single context — recall it gets emitted as-is; when a new context requires variability to elements of the plain text, simply replace those elements with parameter references whose default assignments are the original elements. Now the frame still works in all prior contexts, and new contexts can override as necessary.

– The recursive nature of nested frame trees enables each ancestor frame to seamlessly integrate ever larger subassemblies, a software-engineering basis for the truism, "the higher in the hierarchy, the more clout you have".

The way to utilize <replace> parameters is to embed references to them in **frame-text** using the syntax <;> name </;>, where **name** is an expression that evaluates to a symbol table entry. The mark-up is then replaced by the table's corresponding value. When the underlying programming language is marked-up with <;> parameters, they cause data structures, interfaces, and logic to vary according to the embedding context. Just as importantly, as we shall see, such references also control the processing of frames.

There is one other kind of assignment; it occurs when a frame adapts (invokes) a frame subassembly:

```
<adapt [samelevel]> frame-name
   [<insert [before, after]> break-name
       <content> frame-text  </content>
   </insert> ... ]
</adapt>
```

frame-name is an expression that evaluates to the unique name of an existing frame. Control transfers to the invoked frame and returns to this point after it and its subtree has been processed. Optional <insert>s form a list of parameters assignments *that govern the subtree* rooted in **frame-name**, not just the root frame. This ability of actual arguments to bind to formal arguments located anywhere in the subtree, together with "global overrides local" scoping, allows us to think of the entire subassembly as one seamless whole — which it will be, when the frame processor is finished. Being able to easily flip between integrated and discrete views of the structure simplifies the way frame engineers converge to the appropriate graininess of their domain models, the subject of several chapters in my book.

<insert> causes **frame-text** to be inserted before, after, or instead of the matching <break>, which, as I've said, can be anywhere in the subtree. It looks like this:

```
<break> name <default> frame-text </default> </break>
```

Whereas <replace> evaluates its assigned expression during the assignment (in a possibly more global context), <insert> simply points to its unevaluated **frame-text**. Evaluation takes place upon reference, that is, after being "inserted" into the <break> (of some less context-sensitive frame). In other words, <insert>s get evaluated in the local context while <replace>s may be evaluated in more global contexts. Note that when you want a <replace> to be evaluated in the local context, simply <insert> it. <insert>s work this way because the adapter frame plays the role of an editor, rewriting the adaptee. The <insert>ed frame-text often contains references to <replace> parameters whose values may have been set within the adaptee frame.

The basic assumption is that "the world is arbitrarily hairy". If components are to be sufficiently adaptable, then there must be a way to cope with unpredictable and unique deviations from our archetypes. To do this, a <break> surrounds each structure within a frame, allowing for any element from the smallest paragraph to the entire frame to be "edited". Clearly, such a powerful mechanism can be misused, but something like this is needed to handle arbitrary variability. Programmers give themselves this power, so why not harness it in a formal way?

As with <replace>, frames <insert> only the exceptions needed to fit the context. And again, <break>s can be added later "for free", forgiving our lack of omniscience. In ill-defined domains, you must get on with things well before you understand the best ways of engineering the components.

Consider the reuse of components whose job is to define standard contexts. For example, suppose you have a library of adaptable components that work in both Sun and Mac operating environments (contexts). The natural way to isolate and reuse all the <replace> parameters needed to characterize the desired operating environment is to create one frame for each environment. Frame "S" contains all and only Sun's parameter assignments; frame "M" does the same for Mac. But if I simply <adapt> S or M, the settings vanish (become undefined) as soon as the frame is processed. So, "<adapt samelevel> M" treats M as if it were inside the adapter frame, usually a specification frame. In such a case, the parameter assignments would govern the entire hierarchy.

7 Handling Known Variants

Whereas <insert> is designed for unique customizations, when a list of variations is known, there should be a way to place them inside the frame where they would otherwise have to be <insert>ed. Then any adapter frame merely sets a <replace> parameter to select the appropriate variation. Thus, rather than having to <insert> the desired structure from some more context-sensitive level, the <select> permits the component to carry all the alternatives within itself.

```
<select> <;>name</;>
   <when> relational-expression <content> frame-text </content> ...
  [<otherwise> frame-text </otherwise>]
</select>
```

<select> is a simple case statement, operating at construction time. The <;> parameter's value is tested against each relational ($=, \neq, <, >, \leq, \geq$) expression (including undefined), and the true ones cause the associated frame-texts to be processed. An <otherwise> clause may be included to handle the case where no other selection is made.

While simple, <select> has three important uses. We have already seen one — to eliminate redundancy where a list of variants defines the variability at a single point in the tree. The Mac versus Sun example hints at the converse issue — needing to have *a single point for controlling variability at many places.*

A global or non-functional property that must be implemented in a myriad of localized instantiations is one class of examples — say designing frames to handle multiple languages. `<select>` allows us to set a global parameter that causes a cascade of different effects to take place throughout the hierarchies. Each place where a variation is needed contains a `<select>` with the appropriate variations stored within, depending on the value of the global variables.

The third important application of `<select>` solves the "retrofit" problem described earlier. Suppose a component has been successfully embedded in many contexts, and then an upgrade to it is required. There is a simple way to alter the component so that existing embeddings cannot see it, yet all subsequent ones do. Each place where alterations are required, instead of replacing what was there, place it in a `<otherwise>` clause and make the new variation a choice in the same `<select>`, parameterized with a variable whose value is time-dependent. Existing embeddings do not contain the variable and so will pick up the original frame-text, while newer ones will select for the alternative. This technique extends to successive alterations by adding further `<select>` options, and if the frame becomes cluttered, simply invoke the `<select>` from a sub-frame.

So, no proliferation of look-alike polymorphs occurs, and no retrofits are forced for the sake of compatibility. Nor are work arounds needed. The single adaptive component exhibits its evolutionary audit trail for anyone to understand. The dilemma I described has been "de-horned".

The last construct I would like to show you iterates a pattern such that each iteration expresses a distinct but predefined variation on a theme. For example, in designing a GUI there is an archetypal way to take any data field and position it within the window, display it, provide help, accept an entry, validate it, provide error messages, and so on. The number of data fields and their specific properties can be parameterized so that each iteration over the archetype instantiates it with a set of properties for each field. The syntax for iterating a pattern looks like this

```
<while> <;>name</;>  <content> frame-text </content></while>
```

frame-text not only contains the pattern to be iterated but also increments a parameter that is part of the **name** expression such that when the resulting evaluation of **name** is undefined (not found in the symbol table), the `<while>` terminates.

8 Generators are Unnecessary

This section's heading is deliberately provocative. As I said in the introduction, generators have had a checkered history. They have been a source of programmer frustration because they presume to know how to solve a class of problems in advance of knowing all the contexts for which the generator is applicable. Programmers have a knack for finding situations where the generated code needs to be modified, but either they cannot get at the code or they can but now reusing

the generator destroys their changes. I know this first hand, a main reason why I was motivated to develop frame technology.

The fundamental problem is that the generator "hard wires" the domain's semantics inside itself. Adaptive components, on the other hand, provide a means of storing semantics externally to the processor that assembles them into source code (or whatever). In principle, any detail within any component is adaptable, and all adaptations for a given executable can be segregated into one separate component (e.g., a specification frame). Instead of building a generator, one builds a bi directional translator between the desired high-level abstractions (e.g., WYSIWYG tools), and semantically equivalent parameters in a frame. From there the assembly process completes the code construction, as would a generator, but with the vital difference that the result is arbitrarily fine-tunable by a programmer without compromising users of the high-level abstractions.

In this manner all the power of generators can be made available without their drawbacks.

9 Adaptive Components in Practice

I have tried to explain how, in theory, adaptive components offer:

- Meta-models for domains from well- to ill-defined.
- Automated software modification with increased reliability.
- The ability to collapse the information needed to span a given domain.
- Modularity that matches the domain.
- Flexible software architectures (product lines).
- Adaptability across time.
- Localization of context-sensitive domain knowledge.
- More forgiving approaches to analysis and design.
- The means to transcend code generators.

Good science requires the objective testing of theories. While my firm, Netron Inc., has plenty of strong but anecdotal evidence, such claims are routinely dismissed as "vendor hype". To counter this, Netron took the unusual step of inviting QSM Associates, a highly respected software metrics firm, to do a study and let the chips fall where they may. The study was funded by its participating organizations: Ameritech, Chemical Bank (now Chase), Hudson's Bay Company, Noma-Cabletech, Revenue Canada, Teleglobe Insurance Systems, Union Gas, and two who wished to remain anonymous. They submitted 15 projects done in, believe it or not, COBOL, and ranging in size up to 9.3 million ESLOC (effective source-lines of code = new and modified lines without blanks and comment) — the average being 841K ESLOC. First published in 1995, the study was updated with 14 more projects in 1998. (All figures are courtesy of QSM Associates, Inc.)

Figure 1 compares the study data against QSM's database of thousands of projects done with all manner of tools, languages, and management techniques. The projects are spread along the x-axis, which is QSM's index of productivity

or PI.

$$PI = \log_{1.272}\left[\frac{ESLOC}{time^{4/3} \times (effort/B)^{1/3}}\right] - 26.6 \quad (1)$$

The PI combines the project duration (*time* from detailed design to user acceptance inclusive) with staff-months consumed (*effort*) and the size (*ESLOC*) and complexity (a skills factor B between 0.16 and 0.39) of the resulting system. It is logarithmic such that an increase of 3 PIs implies a doubling of productivity — given the same staff and time, the higher-PI project would have turned out a system twice as large. The open bars show the distribution of projects from QSM's database; the closed bars distribute the projects in the study. The updated study shows an average jump of 10 PIs, compared to the industry norm of 16.9. QSM found that average project time shrank by 70% and staff costs were down by 84%.

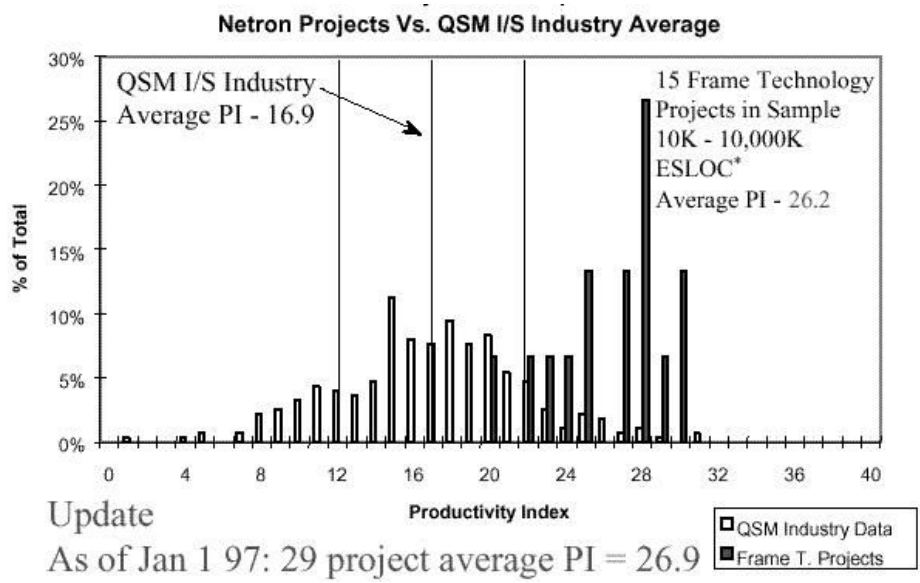


Fig. 1. Productivity of Frames

These are statistics that many industry people find hard to believe. Some object that ESLOC is a bad measure of size, but all admit that multi-million line systems are non-trivial. Others wonder if QSM's database is outdated. In fact, QSM regularly purges old projects. To drive home this point, QSM's Figure 2 compares the study data to projects done with OO tools. Again you can see a major displacement between them and frame technology.

All this nit picking leads me back to a question I raised near the beginning — why, in spite of hard evidence of efficacy, are adaptive components not the standard way that industry does things? I wonder if frames are an idea whose

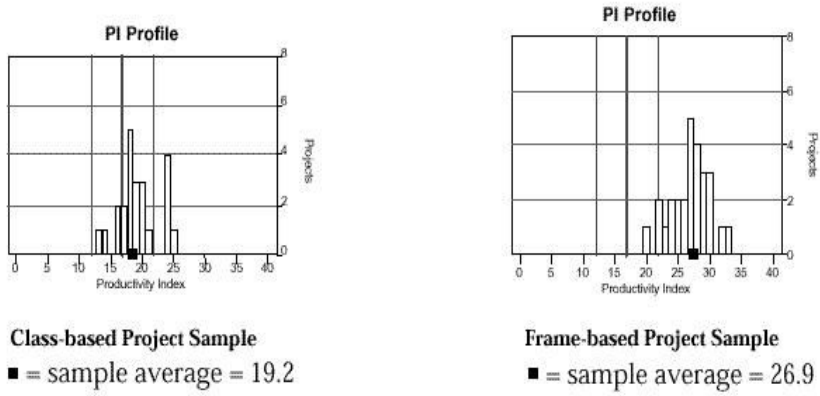


Fig. 2. Productivity of Frames vs Object Technology

time is yet to come. When people can shake off their mental bias for building things from immutable parts, when they realize that well tested adaptive components, like well tested subroutines, can be trusted to output correct results — modified software — much faster and more reliably than humans can, when they become responsive to the importance of avoiding brittle software and the attendant maintenance headaches, just maybe then they can shift to a rigorous development paradigm based on adaptive components.

If you agree with where I am pointing, then help me make it happen! Thank you for listening to my rant.

References

1. BASSETT, P. G. *Framing Software Reuse: Lessons from the Real World.* Prentice Hall, New York, 1997.

Designing for Change, a Dynamic Perspective

Michel Tilman

System Architect, Unisys Belgium
mtilman@acm.org

Abstract. Despite advances in software engineering and process methodologies over the past decades, not many IT projects seem particularly well adapted to today's fast-paced world. Software developers must start to acknowledge change and even uncertainty as a given, rather than the exception that should be studiously avoided, and they must adapt their techniques accordingly.
Some business domains have seen attempts to address this situation. Several workflow vendors, for instance, have been marketing change and end-user programmability as major assets of their products. But, in general, they have been strangely ignorant of (good) modern software engineering practices, and the results have not really lived up to the claims. But we may expect a revival on a grander scale: the ability to (re) define the business logic on the fly is becoming a crucial asset when businesses re-align their core processes around the Internet.
The Internet is transforming the way we envision and design applications. While we could build yesterday's simple Web applications with, let's face it, primitive techniques, this is simply no longer true. High-volume databases, long-term transactions, interoperability, distributed objects, re-use, these are some of the technical issues that must be dealt with. But the real challenge will be to leverage all this technology: we must empower the user to set up, maintain and change his applications more easily.
We need dynamic systems, where applications can be changed at run-time in a high-level way, preferably by end-users. Above all, we need appropriate architectural techniques. In this paper we explore the use of dynamic object models. It turns out that the basic concepts are fairly simple. As for the difficulties, we can borrow solutions from many disciplines in computer science. If we do it right, we can even make the system work for itself.

Introduction

When we analyze a typical business application, say, a relational database application, we find out that knowledge of the business model is usually hard to locate. Looking at the database structures does not tell us the whole story of object structures and relationships. Neither is it easy to spot higher-level constraints and business rules. Thus we have to delve into the application code in order to discover the business model. This makes it harder to understand the business model, and it makes it even harder to change.

G. Butler and S. Jarzabek (Eds.): GCSE 2000, LNCS 2177, pp. 15-25, 2001.

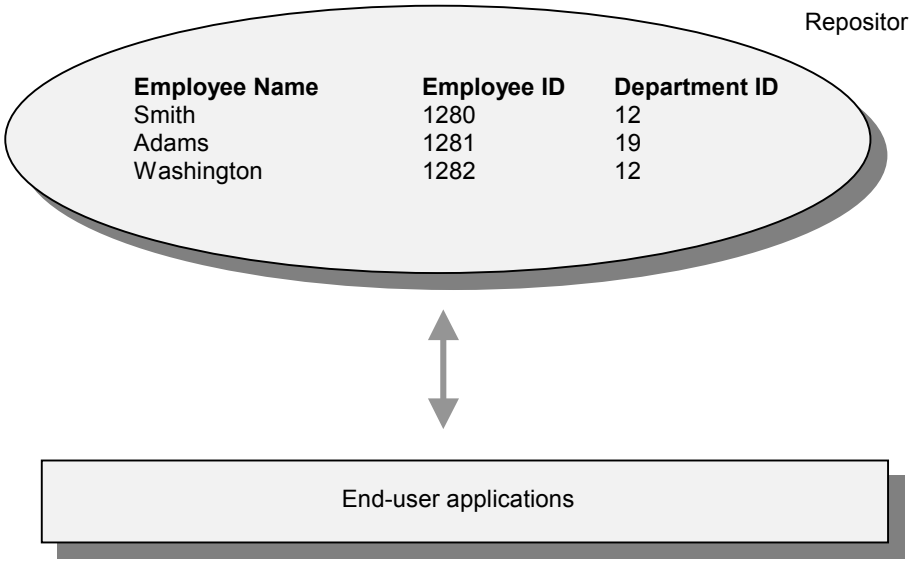


Fig. 1. Business Logic Hard-Wired in End-User Applications (Typical Database Application).

In a way, this is all the more surprising since object technology (and later component technology) was heralded as the solution to re-use and enhanced maintainability. One of the problems is, of course, that object technology itself can never make up for lack of design and algorithmic skills, and for inappropriate process models and team structures. And we need more than class libraries or re-usable components.

Object-oriented frameworks aim to capture the commonality and variability in particular application domains. Frameworks take time and investment to mature, however, and are (initially) harder to understand than straightforward code. Hotspots, points in the framework where we extend or specialize the generic behavior, are of crucial importance. Even with good documentation it usually takes knowledgeable developers to properly use or build frameworks, as the hotspots are (once again) often buried in code. Another point that is often overlooked relates to the framework's ability to work 'out of the box'. Many so-called frameworks are really just skeletons that do not perform any real function of their own.

Model-Driven Systems

In the past decade, several procedure-oriented workflow systems have been replaced by dynamic systems, driven by explicit models of the workflow processes. These

allow for easy run-time change of process definitions without having to shutdown and restart the system. Typical examples are flowchart-like models of processes with graphical editors to build and simulate the processes (whether flowcharts are appropriate representations of business processes, is beyond the scope of this paper).

Another example of a model-driven system is the typical CASE tool. But instead of being interpreted at run-time, most CASE tools generate source code that is subsequently edited by developers. The challenge here is to make design and development co-evolve consistently, the biggest obstacle being the usual discrepancy between a domain-specific design model and a general-purpose programming language.

Dynamic Object Models

A system with a dynamic object model (a.k.a. active or adaptive object model) [Riehle+2000,Tilman2000] has an explicit object model that it interprets at run-time. If you change the object model, the system changes its behavior (essentially) immediately. What goes into a dynamic object model is domain-specific. Typically the model defines objects and their relationships, their states and the conditions under which an object changes its states, as well as the various business rules that specify how a company does their business. Commercial and research applications of systems driven by dynamic object models have been developed for insurance, banking, administrative and workflow applications, amongst others. Several of these add their own specific extensions a core dynamic object model theme.

For dynamic object models to be effective, they need to be sufficiently 'high-level' and complete with regards to their application domains. The UDP insurance system [Johnson+1998], for instance, obviates the need for a general programming language, since the computations that the users are interested in consist essentially of arithmetic expressions, akin to spreadsheet formulas. End-users express their business rules in terms of this high-level 'scripting language', and in terms of objects and object properties. The insurance system empowers end-users to specify the solutions to their problems completely in terms of the UDP dynamic object model.

Generic CASE tools on the other hand provide a high-level way to express objects with their classifications and relationships, but necessarily leave out many details that must be specified in an external language, usually a general-purpose programming language. Thus CASE tool models, even if dynamically interpreted, are not necessarily effective dynamic object models.

Dynamic object models bring 'programmability' closer to the end-user, who is the domain expert. Still, a bit of caution is in order here, as there are many shades of gray in domain expertise and in 'end-user programmability'. Modeling several hundred object types and their relationships, for instance, requires good skills in object-oriented design, whereas creating queries by means of a visual builder may be easier to learn if the user is not exposed to actual SQL code.

Model-Driven Frameworks

Dynamic object models are high-level, executable specifications of the business model. Hence dynamic object models require execution engines. The engine typically provides components and tools to manage the dynamic object model in a repository, to let users interact with the objects through forms, overview lists and search facilities, and to execute the business rules. Thus it makes sense to view the engine as a domain-specific virtual machine for programs expressed as dynamic object models.

When we build several end-user applications with dynamic object models, after a while we may discover duplicate pieces of high-level 'code'. This usually indicates the need for a new kind of generic functionality. Then we change the meta-model that describes what goes into a dynamic object model, and modify the execution engine to support the new functionality. From now on, end-user applications can be developed that exploit this functionality directly, without having to 'code' it over and over again.

While dynamic models seem an overly complex solution at first sight, they arise rather naturally. Applications tend to evolve over the course of the years. Some aspects, like the organizational structure and the business rules, should be easier and faster to change than others, such as the generic functionality of form tools. Object-oriented frameworks provide generic functionality and support for evolution, but, in general, their designs do not pay sufficient attention to the nature of genericity and evolution from the user's point of view. Starting out with a regular framework, we make hotspots that should be easily configurable by 'end-users' explicit, by means of domain-specific dynamic object models. We leave the interpretation of the models to the slower-evolving generic components and to the tools of the framework. Such model-driven frameworks thus directly support software evolution at different levels.

Obviously, we can apply this technique recursively. If, say, at the framework level we regularly require new interface tools, it may worthwhile to consider if a dynamic object model for assembling user interface components is preferable to coding them each time from scratch.

Example: The Argo Framework

Argo, who are responsible for managing public schools within the Flemish community in Belgium, initiated a project to support their administration through IT technology. Not only did they commission a set of finished end-user applications, but they also required a framework to develop and maintain these applications [Tilman+1999]. Furthermore, end-users were to participate in the development process. All this was partly motivated by the fact that Argo, being a political organization, was very susceptible to changes in policies. In the course of the project, two major reorganizations occurred. As a result, project goals and requirements changed frequently.

From the onset we opted for an object-oriented framework driven by a dynamic object model, although both framework and meta-model as such have changed considerably over the past six years. The system (which is completely developed in VisualWorks\Smalltalk) supports the development of administrative-style

applications. These applications typically require a mix of database, document management and workflow functionality. In a later phase we extended the client / server tools with an application server that gives a new interpretation to the dynamic object model and allows users to access the existing applications through the Internet. While the actual end-user applications are specific to the school system, the generic functionality of the framework itself suits many administrative environments.

To enable users to participate in the development process, end-user applications can be developed to a large degree without coding, through modeling and configuring. Scripting is confined to implement more sophisticated business rules. Even the scripting elements are partly modeled and stored in a (meta-) repository alongside the rest of the dynamic object model, separate from the framework code.

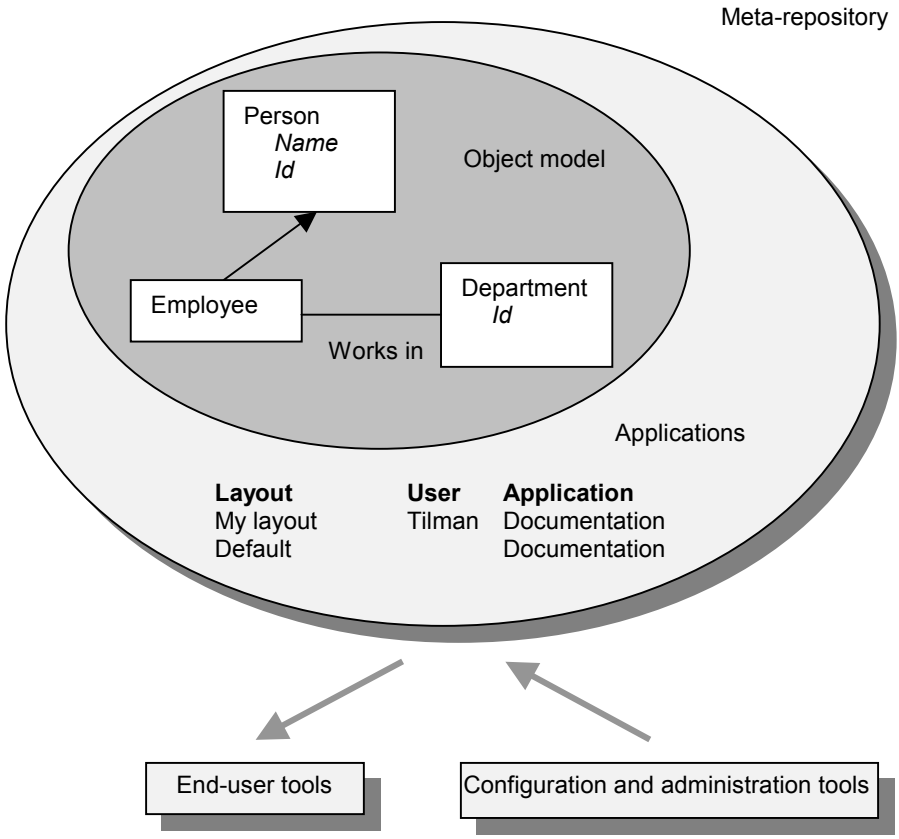


Fig. 2. Dynamically Consulted Meta-information Stored in Repository (Argo Framework).

The dynamic object model at Argo not only describes the business model, such as the organizational model, the structure of the objects, the authorization rules, the constraints and other business rules, but it also describes how the user interacts with all this information. This includes amongst others specifications of forms, overview

lists, query screens and default behavior of applications. In contrast to, say, free-form painter tools to design user interfaces, we provide users with configuration tools that are driven by the model.

Building an application with the Argo framework does not follow a specific sequence of steps. There are some obvious dependencies, however. For instance, it is impossible to define constraints on object types that do not yet exist. Other than these dependencies, developers are free to perform the actions in any order they want.

Building an Application with the Argo Framework

A new application typically starts by extending the existing object model with the required object types, attributes, relationships and basic constraints. Then we specify which part of the object model we want to use in our new application. We capture this information in application environments. Application environments provide a view on the database, but they also serve another important function. For each object type, attribute and relationship within the environment, we define in a course-grained way how we are going to use that particular element: for instance, can we use it in our query editor, is it read-only and can we instantiate a given type? We store both object model and application environment in the database.

Then, without any code or code generation, we get a fully functional default application, where the generic end-user tools (such as the query editor, the overview list and the form tool) adapt themselves automatically to the meta-information in the dynamic object model. The query editor, for instance, dynamically retrieves the list of searchable object types and properties from the application environment, and presents this list to the user. The form tools allow to create and update objects, and to import and view documents. Other tools are operational, such as the thesaurus browser and inbaskets containing to-do lists. This (and other) functionality becomes immediately available to the user, suitably filtered by the specifications of the dynamic object model and the application environment. Thus the framework only needs a (partial) model to work right out of the box.

Later on, the user can customize the default applications interactively, for instance to build (even) sophisticated queries and store them for later re-use, and to configure more appropriate layouts for overview lists and forms. More knowledgeable users configure authorization rules, advanced (passive) constraints and active event-condition-action rules. The latter perform specific context-dependent actions whenever particular events happen. Behavior may be added to the model in order to support the scripting used in some of the business rules.

Advantages

The advantages of this approach are manifold. For one, we do not need to follow a fixed sequence of steps. Instead, we build increasingly complete specifications that are available for immediate use. This effectively blurs the boundaries between developing and prototyping. Although (due to the generic nature of the framework)

we still need a certain amount of scripting, we build end-user applications to a large degree by modeling and configuring only.

Another benefit is the dynamic, interactive nature of the tools, which results in immediate feedback. And since the tools adapt themselves automatically to the run-time meta-information in the central repository, maintenance of client software becomes easier.

Internet-Enabling the Argo Framework

The goal for the Internet version was to re-use the existing dynamic object model and the generic framework functionality. Since application specifications provide a high-level view on how to interact with the information in the database, we had considerable freedom to adapt the interpretation of these specifications to the Web metaphor. To fine-tune this process even further, we added a new configuration tool to the system. But this new tool is really just a configured application, like regular end-user applications. This has important ramifications: once the generic end-user tools have become available through the Internet, any existing application becomes accessible through the Web, including this configuration tool itself. This allows us to bootstrap the development environment to the Internet.

Bootstrapping the Argo Framework

In order for this bootstrapping process to take place, we must not only port the generic end-user tools, but also other configuration tools, such as the object model editor, the business rules editors and the document management tools. Things get a lot easier when the development tools are not dedicated tools, but when they can be configured in the system itself, like the new Internet configuration tool. To this end, we model the meta-model and system objects like electronic documents, thesaurus keywords, layouts, queries and constraints in the system itself. Our generic end-user tools also need to be sufficiently expressive. It is not a trivial task to design such a system up-front, however, so we opted for a bootstrapping process, starting with hard-wired development tools, and replacing them gradually as the generic end-user tools got more powerful.

Another benefit is that end-user application developers, who are not experienced framework developers can now enhance or personalize their own tools. And whenever we enhance the functionality of the generic end-user tools, the development tools go along too.

The ‘downside’ of all this is that we need a more reflective system, including a self-descriptive meta-model and rules acting on rules.

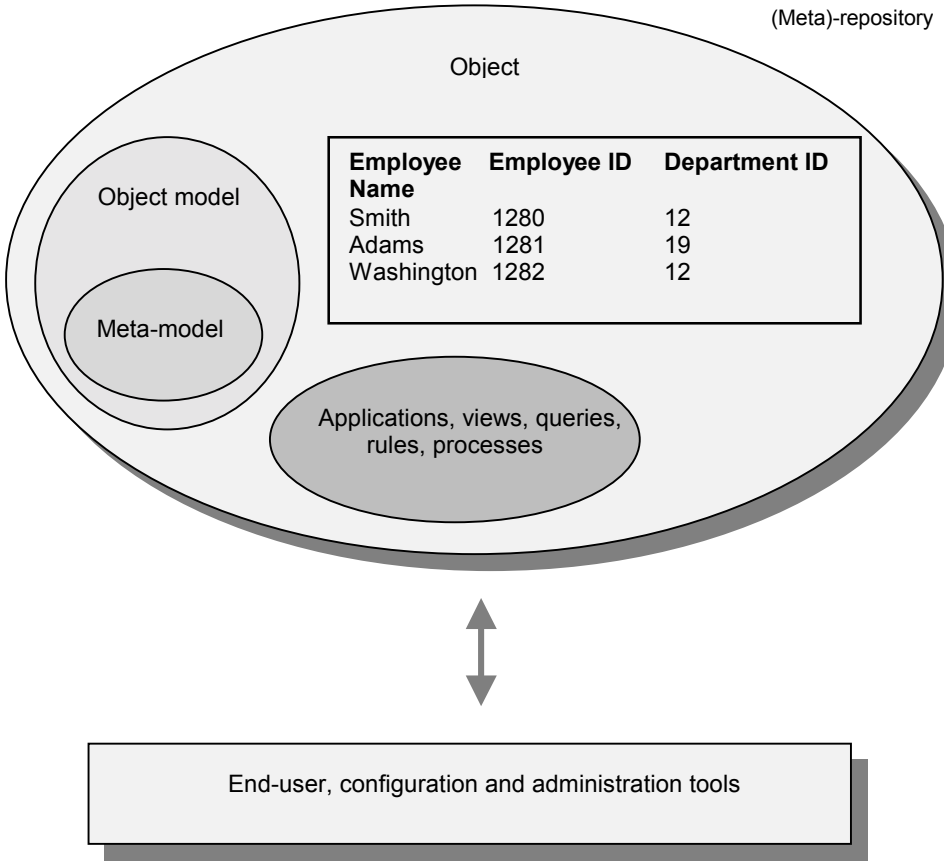


Fig. 3. Self-Descriptive Dynamic Object Model (Argo Framework).

Our experience tells us that building such as system is not necessarily that hard, as long as we start small and stay focused. In the end, we get many things for free.

Pros and Cons of Dynamic Object Models

Dynamic object models allow us to build applications that are easier to change, even by end-users. Dynamic object models offer additional benefits, some of which are not apparent at first sight, but they also impose their own challenges. We refer the reader to the "Dynamic Object Model" pattern paper [Riehle+2000] for additional discussions.

Advantages of Dynamic Object Models

Dynamic object models contain run-time meta-information of the end-user applications. Since this data is explicitly available, rather than implicitly hard-coded, we can put this information to many uses. For one, it becomes possible to automatically generate useful and consistent documentation about the end-user applications. We can even personalize this documentation for each individual user.

Compatibility with existing client code is always a difficult problem when interfaces need to be changed. Dynamic object models make this much less of a chore. Say we must change the functionality of form layouts in the Argo framework, then we simply write a conversion script that enumerates the layouts in the repository, and converts them from one format into another. No 'traditional' modification of the client code in the end-user applications is required.

When end-user applications are completely described by (high-level) dynamic object models, we can easily migrate to other environments, since we only need to port the framework code. The end-user applications remain operational. The bootstrapping approach used in the Argo framework further reduces the amount of work that needs to be done.

Dynamic object models help us set up test suites easily. For instance, by enumerating application environments, stored queries and list and form layouts in the Argo framework, we easily created a parameterizable framework to benchmark database servers or to perform regression testing.

Since the business model is explicitly represented, it is much easier to understand the model and to estimate the impact of changing user requirements. Building end-user applications becomes a more declarative process, where developers concentrate on the 'what', rather than on the 'how'.

Increased Design and Run-Time Complexity

Frameworks driven by dynamic object models are initially harder to understand. Some of this complexity is common to most frameworks. A further failure to grasp readily what is going on is due to the design of dynamic object models. Many systems use the following (or a similar) core design:

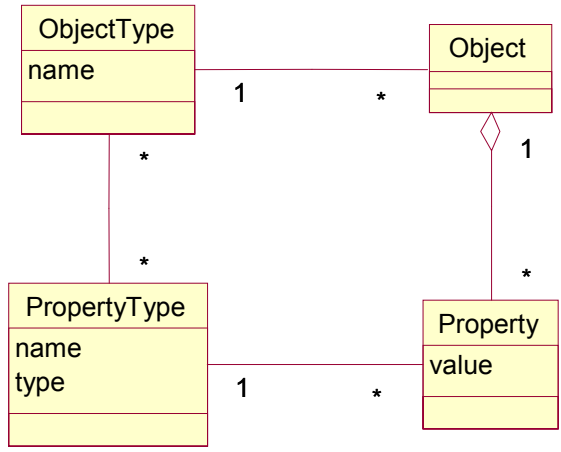


Fig. 4. Dynamic Object Model (Core Design).

Here, objects are typically represented as dynamically typed instances of a generic class. The object types are full-blown objects in their own right. Each object type maintains a list of property types that defines the 'structure' of its 'instances'. Each instance maintains its properties in a dynamic list. This design enables us to create new object types at run-time, and to instantiate these types without having to create new classes. More elaborate or domain-specific designs build upon this kernel structure, adding, for instance, type-subtype relationships, arbitrary relationships and composite structures.

Hence what used to be an instance of a specific class in a regular programming language, is now represented as a collection of objects with various relationships. This makes it harder to understand the design and the run-time structure of dynamic object models. The main problem lies with the tools. Dynamic object models effectively introduce new 'languages', but we are still using the old development tools (such as class browsers and object inspectors) to look at the structure. Once we have created new design and development tools adapted to the new language, these problems go away.

Some of the regular development tools are no longer available when building end-user applications. The Argo framework, for instance, uses a multi-user team development tool to store the framework code. Since we needed versioning facilities for business rules and object behavior, we added a simple versioning scheme to our dynamic object model. A bit of modeling and a few business rules enabled us to configure this in the system itself in just a few hours.

Performance

Many people fear that dynamic systems are inherently slow. Yet many dynamic systems in diverse domains offer the flexibility we really need, while exhibiting good performance. Smalltalk virtual machines, for instance, have come a long way since the early seventies, and they are still improving; processor speeds have increased in part because of clever, dynamic scheduling strategies, and database vendors are improving statistics-based query optimizers.

Whether performance of a specific dynamic object model is acceptable, ultimately depends on its particular application domain and on the end-user requirements, but we can also borrow many ideas from other software engineering domains.

For one, there are many ways to improve upon a naive implementation of dynamic object models [Tilman2000]. If the language permits, we can even generate concrete classes for each object type dynamically, using a just-in-time scheme.

The Argo framework contains many caching strategies, some explicit, some implicit. These are not confined to the persistence component only.

Another solution aims to optimize for typical usage patterns. The Argo persistence component, for instance, uses generic hints and statistical information to drive the query generation process. Discovering usage patterns and designing appropriate heuristics usually takes time, however, since we need to analyze the run-time behavior of a relevant number of end-user applications.

We can also apply the very dynamic nature of these systems to overcome their weaknesses. Some of the authorization rules in the Argo system, for instance, require a lot of data, depending on the contents of an object's properties, and are rather time-consuming. By waiting till the last possible moment, we analyze the definitions of these rules at run-time, and decide which ones we may safely skip. Partial evaluation techniques offer additional interesting opportunities to execute static 'computations' at development time.

References

[Johnson+1998] Ralph E. Johnson and Jeff Oakes. "The User-Defined Product Framework." Unpublished manuscript, available from http://st-www.cs.uiuc.edu/users/johnson/papers/udp

[Riehle+2000] Dirk Riehle, Michel Tilman and Ralph E. Johnson, Dynamic Object Model, PloP 2000

[Tilman+1999] Michel Tilman and Martine Devos, A Reflective and Repository-Based Framework, pp.29-64, Implementing Application Frameworks (M.E. Fayad, D. C. Schmidt, R. E; Johnson ed.), Wiley Computer Publishing

[Tilman2000] Michel Tilman, Designing for Change, a Reflective Approach, Tutorial at the GCSE 2000 Conference

On to Aspect Persistence

Awais Rashid

Computing Department, Lancaster University, Lancaster LA1 4YR, UK
marash@comp.lancs.ac.uk

Abstract. Over the recent years aspect-oriented programming (AOP) has found increasing interest among researchers in software engineering. *Aspects* are abstractions which capture and localise cross-cutting concerns. Although persistence has been considered as an aspect of a system, persistence of aspects has been largely ignored. This paper identifies several scenarios where aspect persistence is an essential requirement. A model for aspect persistence and an initial prototype based on AspectJ (0.6beta2) are presented. Various open issues are also pointed out.

1 Introduction

Over the recent years aspect-oriented programming (AOP) [9] has found increasing interest among researchers in software engineering. It aims at easing software development by providing further support for modularisation. *Aspects* are abstractions which serve to localise any cross-cutting concerns e.g. code which cannot be encapsulated within one class but is tangled over many classes. A few examples of aspects are memory management, failure handling, communication, real-time constraints, resource sharing, performance optimisation, debugging and synchronisation. In AOP classes are designed and coded separately from aspects encapsulating the cross-cutting code. The links between classes and aspects are expressed by explicit or implicit *join points*. An *aspect weaver* is used to merge the classes and the aspects with respect to the join points. This can be done statically as a phase at compile-time or dynamically at run-time [7, 9].

AOP research has identified persistence as an aspect of a system [12, 21]. However, the need for aspect persistence has not been considered. Existing work on lifetime of aspects [7, 9, 11] has argued that at least some of the aspects should live for the lifetime of the program execution and not die at compile-time. In certain cases the need for aspects to outlive the program execution can arise. [17], for example, proposes the use of an aspect repository for managing and reusing aspects in a distributed environment. Aspects in the repository live beyond the execution of the programs using them. Some aspects can be persistent by nature. [18, 19] identify several aspects cross-cutting the schema and the data in object-oriented databases. These include instance adaptation, versioning, clustering, access rights and data representation, etc. Due to the close integration between object-oriented programming languages and object-oriented databases these aspects are seamlessly used by application programs but are persistent by nature and reside in the database.

G. Butler and S. Jarzabek (Eds.): GCSE 2000, LNCS 2177, pp. 26-36, 2001.

This paper proposes an approach for aspect persistence. The next section makes a case for aspect persistence by discussing several scenarios involving persistent aspects. A description of the proposed aspect persistence model is presented in section 3. The model is independent of a particular AOP approach and takes into account the evolving nature of aspect languages and representations. An initial prototype implementation of the model using AspectJ (0.6beta2) [2] and the Jasmine (1.21) object database management system is discussed in section 4 while the various open issues are outlined in section 5. Section 6 concludes the paper and identifies directions for future work.

2 Why Persistent Aspects?

This section describes some scenarios involving persistent aspects. The discussion demonstrates that aspect persistence is an essential requirement in many cases, hence highlighting the need to provide support for the purpose.

2.1 An Aspect Repository for Managing and Reusing Aspects

[17] proposes the use of an aspect repository to manage and reuse aspects in a distributed environment. The approach aims at reducing the additional complexity introduced due to the need to find an aspect in a network of computing units. Aspects in the repository are persistent and applications at different locations can query the repository to retrieve the required aspects at run-time. This is shown in Fig. 1 [17].

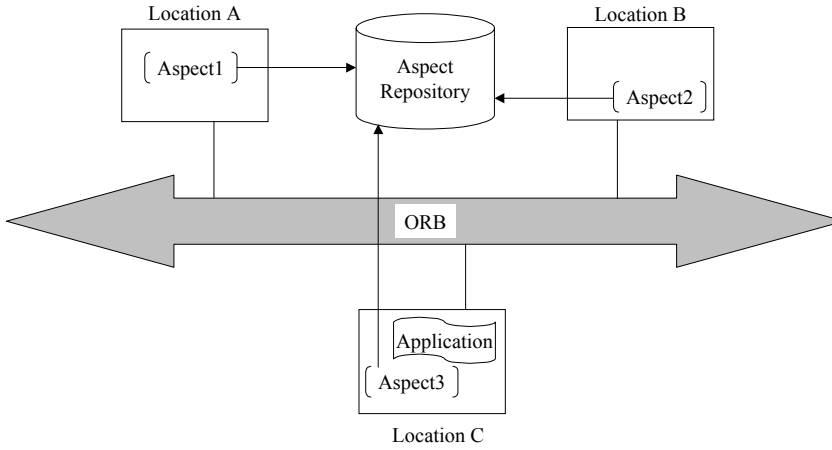


Fig. 1. Managing and Reusing Distributed Aspects through an Aspect Repository [17].

Another scenario is an automated software development environment where both components and aspects reside in a database. The appropriate components and aspects are retrieved by the assembling process which carries out the weaving.

2.2 Instance Adaptation During Class Versioning

The second example is based on instance adaptation during class versioning [3, 14, 20] in object-oriented databases. Class versioning allows several versions of one type to be created during evolution. An instance is bound to a specific version of the type and when accessed using another type version (or a common type interface) is either converted or made to exhibit a compatible interface. This is termed as instance adaptation and is essential to ensure structural consistency. A detailed description of class versioning is beyond the scope of this paper. Interested readers are referred to [3, 14, 20]. The following discussion demonstrates that the instance adaptation code cross-cuts the class versions and can be separated using aspects. It should be noted that the instance adaptation aspects cross-cut persistent entities (the class versions) and are persistent by nature.

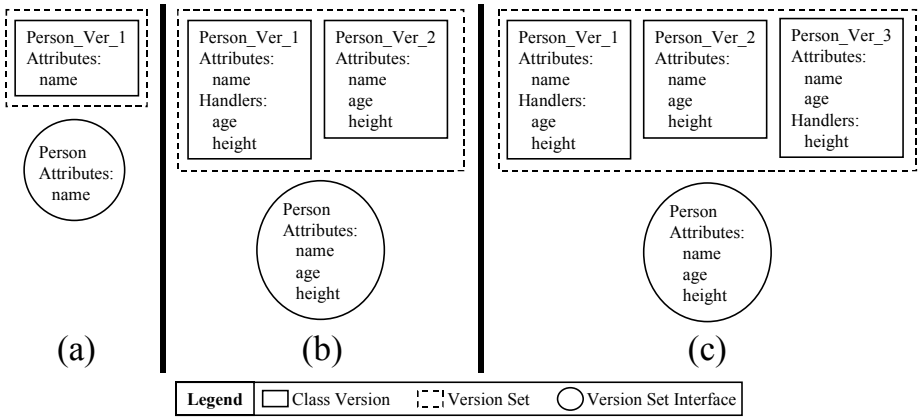


Fig. 2. Class Versioning in ENCORE.

We first consider the instance adaptation strategy of ENCORE [20]. A similar approach is employed by AVANCE [3]. As shown in figure 2, applications access instances of a class through a *version set interface* which is the union of the properties and methods defined by all versions of the class. Error handlers are employed to trap incompatibilities between the version set interface and the interface of a particular class version. These handlers also ensure that objects associated with the class version exhibit the version set interface. As shown in Fig. 2(b) if a new class version modifies the version set interface (e.g. if it introduces new properties and methods) handlers for the new properties and methods are introduced into all the former versions of the type. On the other hand, if creation of a new class version does not modify the version set interface (e.g. if the version is introduced because properties and methods have been removed), handlers for the removed properties and methods are added to the newly created version (cf. Fig. 2(c)).

The introduction of error handlers in former class versions is a significant overhead especially when, over the lifetime of the database, a substantial number of class versions can exist prior to the creation of a new one. If the behaviour of some handlers needs to be changed maintenance has to be performed on all the class

versions in which the handlers were introduced. To demonstrate the use of persistent aspects we have chosen the scenario in Fig. 2(b). Similar solutions can be employed for other cases. As shown in Fig. 3(a) instead of introducing the handlers into the former class versions they are encapsulated in an aspect. Fig. 3(b) depicts the case when an application attempts to access the *age* and *height* attributes in an object associated with version 1 of *class Person*. The aspect containing the handlers is woven into the particular class version. The handlers then simulate (to the application) the presence of the missing attributes in the associated object.

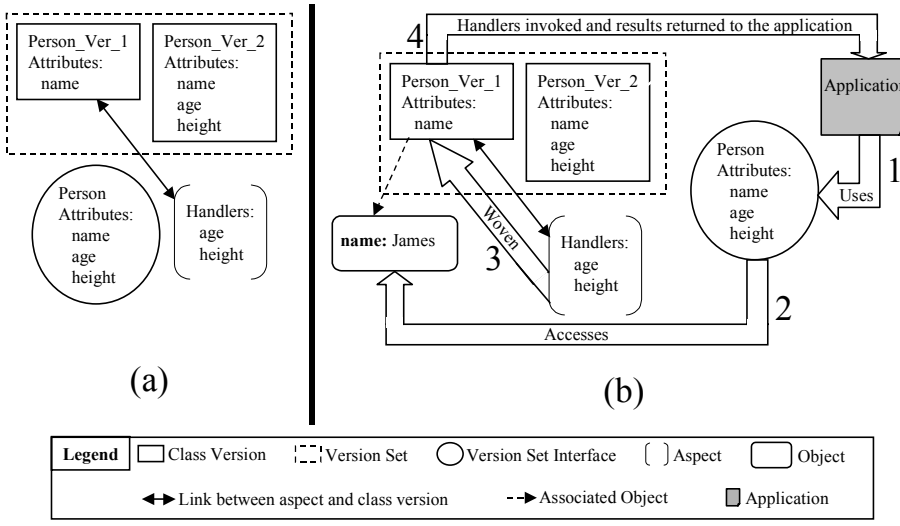


Fig. 3. The Aspect-Oriented Instance Adaptation Approach.

Encapsulating handlers in an aspect offers an advantage in terms of maintenance as only one aspect is defined for a set of handlers for a number of older class versions. Behaviour of the handlers can be modified within the aspect instead of modifying them within each class version. Aspects also help separate the instance adaptation strategy from the class versions. For example, let us suppose one wants to employ a different instance adaptation approach[1], the use of update/backdate methods for dynamic instance conversion between class versions (as opposed to simulating a conversion) [14]. In this case only the aspects need to be modified without having the problem of updating the various class versions. These are automatically updated to use the new strategy when the aspect is woven. The aspect-oriented approach has a run-time overhead as aspects need to be woven and unwoven (if adaptation strategies are expected to change). However, this overhead is smaller than that of updating and maintaining a number of class versions. The overhead can be reduced by leaving an aspect woven and only reweaving if the aspect has been modified. Details of the aspect-oriented instance adaptation approach have been reported in [18].

[1] Such a need can arise due to application/scenario specific adaptation requirements or the availability of a more efficient strategy.

2.3 Clustering

[19] identifies clustering as a persistent aspect which cross-cuts the objects residing in an object-oriented database. Traditionally it is the task of the database application programmer to ensure that related objects are clustered together. However, the task is easy only for a small number of objects. As the number of objects that need to be clustered increases (it should be noted that the clustering reasons could be different for different groups of objects) the programmer's task becomes more and more complicated. The situation worsens if the same object needs to be clustered together with more than one group. Considering clustering as an aspect of data residing in a database allows managing these complex scenarios transparently of the programmer. The programmer can specify clustering as an aspect of a group of objects regardless of whether some objects in the group also form part of another group. The clustering aspects can then be used by the system to work out an efficient storage strategy. Furthermore, if the clustering requirements for an object change the programmer can re-configure the clustering aspect to indicate which group of objects should be clustered with this object. This helps to manage the physical reorganisation of the various clustered objects transparently of the programmer. It should also be noted that clustering is not necessarily an aspect of all the objects residing in the database. Introducing clustering as an aspect allows only those objects having this aspect to be clustered.

2.4 Other Persistent Aspects

In addition to the examples presented in section 2.1-2.3 several other persistent aspects can be identified. Versioning is an aspect which cross-cuts both objects and classes in an object-oriented database (assuming the system supports both object and class versioning). Constraints can be considered as an aspect of the object database. Traditionally constraints are specified at the application level or through a DBMS service. Considering constraints an aspect of database entities and providing a concrete abstraction simplifies their specification and management. Access rights, security and data representation can also be regarded as aspects. All these aspects cross-cut persistent entities residing in a database and are persistent by nature.

3 A Model for Aspect Persistence

Section 2 presented several scenarios where aspect persistence is an essential requirement. This section proposes a model for aspect persistence. The model is based on the following observations:

1. Object database management systems often require that classes whose instances are to be stored in the database extend a system provided *Persistent Root Class*. Examples of such systems include the Object Data Management Group (ODMG) standard [4], O2 [15] and Jasmine [6].

2. Due to proprietary restrictions it is not possible to modify the system classes implementing the persistence model of the object database management system being used.
3. Most object database management systems employ the *persistence by reachability* principle. When a transaction commits all objects reachable from a persistent object are made persistent. Examples of such systems include the ODMG standard [4], O2 [15] Jasmine [6] and Object Store [16].
4. Persistence is a cross-cutting concern in a system [12, 21].
5. There are various AOP approaches available e.g. AspectJ [2], Composition Filters [1], HyperJ [5], Adaptive Programming [10, 13], etc. All these approaches aim at achieving a better separation of concerns. However, the aspect structures employed for the purpose vary considerably.
6. Aspect languages and aspect structures are continuously evolving as AOP technologies mature.

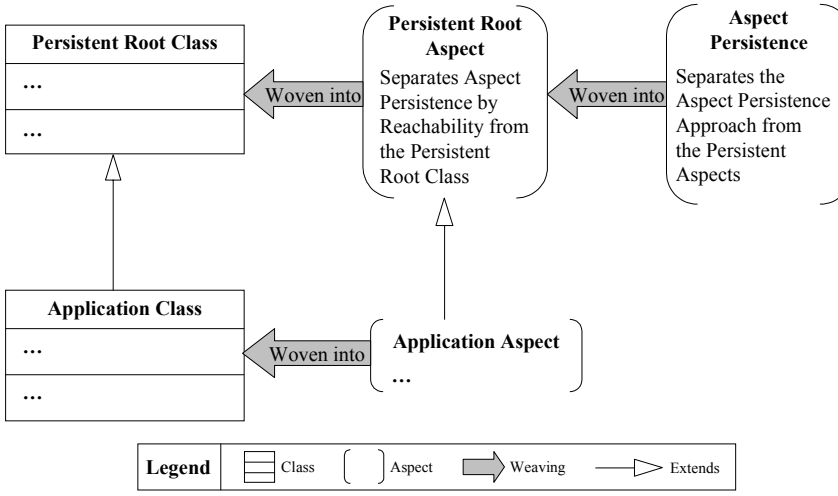


Fig. 4. The Aspect Persistence Model.

The persistence model is shown in Fig. 4. All application classes extend the *Persistent Root Class* offered by the particular object database management system. A similar mechanism is employed for making application aspects persistent. All application aspects extend a *Persistent Root Aspect*. This is a natural extension of the persistence model employed by several object database management systems (observation 1). Since it is not possible to modify the system classes implementing the persistence model of the object database management system (observation 2), all links between aspects and classes have been kept strictly *class directional* i.e. the aspects know about the classes but not vice versa [8]. This also facilitates the modification and reuse of the aspect code [8] (as discussed later) and avoids introduction of additional evolution complexity.

When a transaction commits all aspects reachable from a persistent object are made persistent (observation 3). As shown in Fig. 5 *Aspect 1* is an instance of an aspect extending the *Persistent Root Aspect* and reachable from a persistent object *Object 1*. It is, therefore, made persistent. Although *Aspect 2* is also an instance of an aspect extending the *Persistent Root Aspect* it will not be made persistent as it is not reachable from any persistent objects in the scope of the transaction. *Aspect 3* poses an interesting scenario. It is reachable from a persistent object but is not an instance of an aspect extending the *Persistent Root Aspect*. It will be coerced into persistence in order to preserve persistence by reachability. The coercion strategy can be to force the aspect to extend the *Persistent Root Aspect* or to annotate the particular aspect instance (*Aspect 3*) with persistence-related code. The choice of coercion strategy has been left to the implementation of the model. It should be noted that aspect persistence by reachability is transitive in nature i.e. not only aspects reachable from persistent objects are made persistent but also aspects reachable from persistent aspects are transitively made persistent.

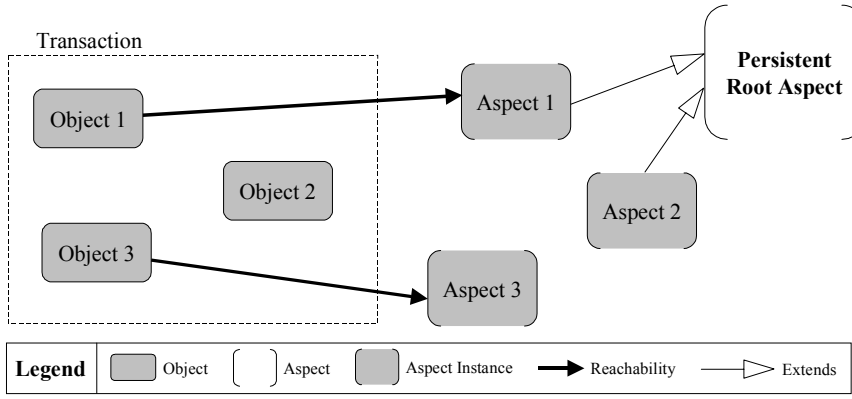


Fig. 5. Aspect Persistence by Reachability.

As shown in Fig. 4 the *Persistent Root Aspect* encapsulates the persistence by reachability code. This code defines the behaviour of an aspect transitively reachable from a persistent object upon transaction commit. Since all application aspects extend the *Persistent Root Aspect* the behaviour is propagated down the hierarchy. Reflecting on observations 4, 5 and 6 persistence has been regarded as an aspect of the persistent aspects (cf. Fig. 4). The persistence approach is separated from the persistent aspects through the *Aspect Persistence* aspect. This makes the model independent of a particular AOP approach because the persistent aspects do not encapsulate the knowledge about their storage structure (which largely depends on the particular AOP approach being employed). It also localises the changes resulting from the evolution of the aspect language or the aspect structure making maintenance and modifications to the persistence model inexpensive. Such changes are further aided by the class-directional nature of links between aspects and classes.

4 PersAJ: An AspectJ Prototype

This section presents PersAJ (Persistent AspectJ), an implementation of the persistence model using AspectJ (0.6beta2) from Xerox PARC and the Jasmine (1.21) object database management system from Computer Associates. The implementation is shown in Fig. 6. Application classes and aspects from the instance adaptation scenario in section 2.2 have been used as an example. It should be noted that the figure only depicts the code relating to the description in section 3. Other implementation code has been omitted.

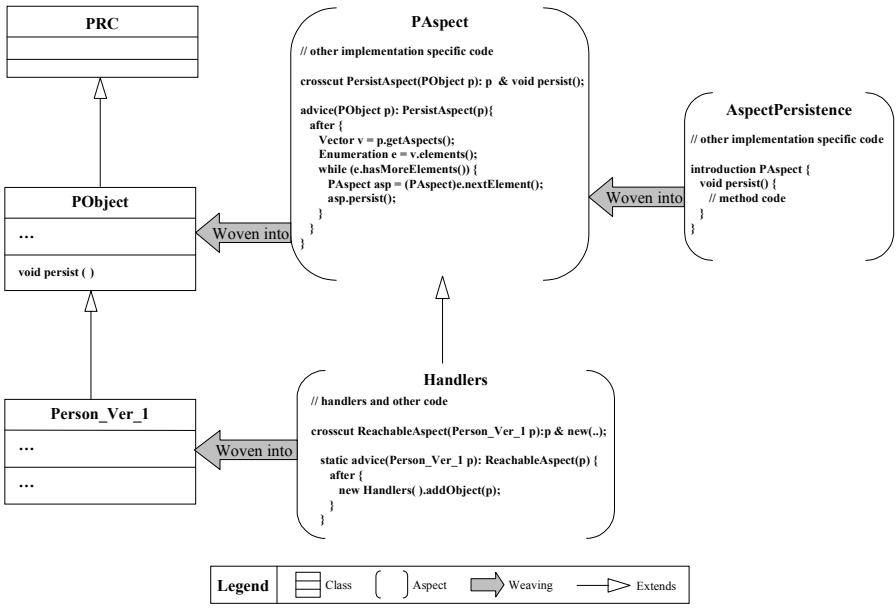


Fig. 6. PersAJ: An Implementation of the Persistence Model Using AspectJ (0.6beta2).

PRC is the *Persistent Root Class* in the Jasmine Persistent Java binding (pJ). This class is extended by all Java classes whose instances are to be stored in the Jasmine database. Since no information is available about the PRC class, we have introduced the class *PObject* which acts as the *Persistent Root Class* for all application classes in PersAJ (in this case *Person_Ver_1*). *PObject* has a special instance-level method called *persist( )* which is invoked for all persistent objects (identified through persistence by reachability) just before a transaction commits. This is achieved by providing wrappers around the Jasmine transaction commit operation. *PAspect* is the *Persistent Root Aspect* in PersAJ and is extended by all persistent application aspects (in this case *Handlers*). The application aspects implement a special static advice making the aspect instance reachable from the associated class object being instantiated. The advice shown in *PAspect* (cf. Fig. 6) determines all reachable aspects from a persistent object after the *persist( )* method has been invoked (upon transaction commit). All reachable aspects are made persistent through a call to the *persist( )*

method for the aspect instance. The *persist( )* method is introduced into *PAspect* by the *AspectPersistence* aspect which separates the persistence approach and storage structure from the aspects being made persistent.
The Jasmine Persistent Java binding (pJ) offers a Java Persistence Processor (JPP): a code generator which adds persistence capability to a Java class by adding code to its definition. It also generates a corresponding schema definition for the underlying Jasmine database system. PersAJ employs a simple script which runs the AspectJ compiler in the preprocess mode. The code generated by AspectJ is passed to a custom built parser which parses the generated class definitions and replaces any $ signs with two underscores. This is essential as AspectJ uses $ signs to differentiate generated code from the Java code supplied by the programmer while Jasmine regards $ as a reserved character. The class definitions produced by the parser are passed to the Jasmine Persistence Processor which adds the persistence related code and generates the database schema. Instances of the processed classes and aspects can now be stored in the database.

5 Open Issues

This paper introduces the idea of persistent aspects. One of the open research issues is the persistent representation of an aspect. Due to the different aspect representations e.g. AspectJ, Composition Filters, etc. used in application programs, persistent representation of aspects needs careful exploration.
Persistent aspects and dynamic weaving introduce additional overhead at run-time and can be feasible only with efficient weaving mechanisms. The development of efficient weavers is therefore an important research issue.
One of the issues identified during development of PersAJ is the need for parameterised aspects. This is not supported in the AspectJ implementation (0.6beta2) due to the lack of parametric polymorphism in Java. If aspect parameterisation is available the implementation can be more generic and maintainable. An aspect can be parameterised by classes in which it is to be woven, hence, making the join points and crosscuts generic. Special *weave parameters* can be used to provide a generic reconfiguration mechanism during dynamic weaving. It can also provide a solution for the different aspect representations problem identified above. The representation to be used can be determined by the aspect passed as a parameter making the persistence model more transparent to the programmer.

6 Conclusions and Future Work

This paper has proposed an approach for aspect persistence. Aspect persistence is a natural extension of existing work on lifetime of aspects. Although existing AOP research has considered persistence as an aspect of a system, the need for aspect persistence has been largely ignored. The novelty of this work is in identifying concrete scenarios where aspect persistence is an essential requirement and providing support for the purpose. The proposed aspect persistence model is independent of the

aspect language and the aspect representation employed by it. Persistence is considered an aspect of the persistent aspects hence providing support for inexpensive changes to the persistent representation of aspects due to changes in the aspect language or the aspect structure. Class-directional links allow implementation of the model without modifying the persistence model of the object database management system being used. A prototype implementation of the model based on AspectJ (0.6beta2) and Jasmine (1.21) object database management system validates the concepts proposed in the paper.

At present AspectJ 0.7beta4 has been released. Our work in the immediate future will involve porting the PersAJ implementation to the new release. We are also interested in providing implementations of the model for other AOP approaches. At present the persistent representation of aspects is handled by the Jasmine Java Persistence Processor. We are interested in investigating various persistent representations of aspects. We are particularly interested in developing an infrastructure mapping the various aspect structures on to a common persistent representation and vice versa. Such an infrastructure will serve as middleware between implementations of the persistence model for different aspect languages and the common persistent representation of aspects.

Note: The work reported in this paper is part of the Aspect-Oriented Databases initiative at Lancaster which aims at bringing the notion of separation of concerns to databases and providing aspect persistence mechanisms. Further information can be found at: http://www.comp.lancs.ac.uk/computing/aod/

References

[1] Aksit, M. and Tekinerdogan, B., "Aspect-Oriented Programming using Composition Filters", *Proceedings of the AOP Workshop at ECOOP'98, 1998*

[2] AspectJ Home Page, http://aspectj.org/ , *Xerox PARC, USA*

[3] Bjornerstedt, A. and Hulten, C., "Version Control in an Object-Oriented Architecture", *In Object-Oriented Concepts, Databases, and Applications (eds: Kim, W., Lochovsky,F. H.), pp. 451-485, Addison-Wesley 1989*

[4] Cattell, R. G. G., *et al*., "The Object Database Standard: ODMG 2.0", *Morgan Kaufmann, c1997*

[5] "Multi-dimension Separation of Concerns using Hyperspaces", http://www.research.ibm.com/hyperspace/

[6] "Jasmine 1.21 Documentation", *Computer Associates International, Inc., Fujitsu Limited, c1996-98*

[7] Kenens, P., *et al.,* "An AOP Case with Static and Dynamic Aspects", *Proceedings of the AOP Workshop at ECOOP '98, 1998*

[8] Kersten, M. A. and Murphy, G. C., "Atlas: A Case Study in Building a Web-based Learning Environment using Aspect-oriented Programming", *Proc. of OOPSLA 1999, ACM SIGPLAN Notices, Vol. 34, No. 10, Oct. 1999, pp. 340-352*

[9] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., and Irwin, J., "Aspect-Oriented Programming", *Proceedings of ECOOP'97, LNCS 1241, pp. 220-242*

[10] Lieberherr, K. J., "Demeter", http://www.ccs.neu.edu/research/demeter/index.html

[11] Matthijs, F., et al., "Aspects should not Die", Proceedings of the AOP Workshop at ECOOP'97, 1997

[12] Mens, K., Lopes, C., Tekinerdogan, B., and Kiczales, G., "Aspect-Oriented Programming Workshop Report", *ECOOP'97 Workshop Reader, LNCS 1357, pp. 483-496*

[13] Mezini, M. and Lieberherr, K. J., "Adaptive Plug-and-Play Components for Evolutionary Software Development", *Proceedings of OOPSLA 1998, ACM SIGPLAN Notices, Vol. 33, No. 10, Oct. 1998, pp. 97-116*

[14] Monk, S. and Sommerville, I., "Schema Evolution in OODBs Using Class Versioning", *SIGMOD Record, Vol. 22, No. 3, Sept. 1993, pp. 16-22*

[15] "The O2 System - Release 5.0 Documentation", *Ardent Software, c1998*

[16] "Object Store C++ Release 4.02 Documentation", *Object Design Inc., c1996*

[17] Pulvermueller, E., Klaeren, H., and Speck, A., "Aspects in Distributed Environments", *Proceedings of GCSE 1999, Erfurt, Germany*

[18] Rashid, A., Sawyer, P., and Pulvermueller, E., "A Flexible Approach for Instance Adaptation during Class Versioning", *Proceedings of ECOOP 2000 OODB Symposium (in publication as an LNCS volume by Springer-Verlag)*

[19] Rashid, A. and Pulvermueller, E., "From Object-Oriented to Aspect-Oriented Databases", *Proceedings of DEXA 2000, Lecture Notes in Computer Science 1873, pp. 125-134*

[20] Skarra, A. H. and Zdonik, S. B., "The Management of Changing Types in an Object-Oriented Database", *Proceedings of the 1st OOPSLA Conference, Sept. 1986, pp. 483-495*

[21] Suzuki, J. and Yamamoto, Y., "Extending UML with Aspects: Aspect Support in the Design Phase", *Proceedings of the 3rd AOP Workshop held in conjunction with ECOOP'99*

Symmetry Breaking in Software Patterns

James O. Coplien[1] and Liping Zhao[2]

[1] Bell Laboratories, ILIE00 2Z307, 263 Shuman Blvd, Naperville, IL 60566 USA
cope@research.bell-labs.com
[2] Department of Computation, UMIST, P. O. Box 88, Manchester M60 1QD, UK
liping@co.umist.ac.uk

Abstract. Patterns have a longstanding identity in the scientific community as results of a phenomenon called symmetry breaking. This article proposes a formalism for software patterns through connections from software patterns to symmetry and symmetry breaking. Specifically, we show (1) the ties from Alexander's work to symmetry and symmetry-breaking foundations; (2) many programming languages provide constructs that support symmetry; (3) software patterns are the results of symmetry breaking, compensating for design shortfalls in programming languages. The proposed pattern formalism may be useful as a foundation for pattern taxonomies, and to differentiate patterns as a design discipline from heuristics, rules, and arbitrary micro-architectures.

1 Introduction

Most contemporary attempts to formalize patterns (e.g., [14], [15]) ignore both the prior art and the most relevant foundations for potential formalization. This paper shows that symmetry lies at the very foundation of the pattern formalism. The paper also shows that the symmetry formalism has historically been broad enough not only for the natural sciences and Alexander's work, but that it can serve software as well.

Pattern and symmetry are closely related. The formation of pattern can be characterized by symmetry operations, which, in the sense of classic symmetry, are rigid motions of geometric object [33]. We believe that the geometric nature of classic patterns has been the main inspiration of Alexander's work ([1], [2], [3]). An early realization of our research is that most of these structural properties relate to symmetry. Leading software pattern practitioners have also recognized that the best software patterns are also geometric in appearance and organization for reasons of composability, which is in concert with Alexander's theories of patterns and centers.

The next section introduces symmetry concepts and group theory, which is the prerequisite for the paper. Section 3 gives an overview to Alexander's theory and its connection to symmetry. Section 4 presents symmetry in O-O software. Sections 5 and 6 introduce symmetry breaking as a foundation for patterns. Section 7 illustrates symmetry and symmetry breaking in programs and designs. Section 8 proposes a formalism for software patterns. We conclude the paper in Section 9.

G. Butler and S. Jarzabek (Eds.): GCSE 2000, LNCS 2177, pp. 37-54, 2001.

2 Group Theory and Symmetry Concepts

Group theory offers constructs to formally characterize symmetry through *symmetry groups*. This section gives a brief introduction to group theory and symmetry ([28], pp 9-10):

> A group is a nonempty set G together with a law of composition $(a, b) \mapsto ab : G \times G \rightarrow G$ satisfying the following four axioms:
>
> 1. *Closure*. For all $a, b \in G$, the set G is closed under composition: $ab, ba \in G$.
> 2. *Associativity*. For all $a, b, c \in G$, the composition is associative: $(ab)c = a(bc)$.
> 3. *Existence of An Identity Element*. For all $a \in G$, there exists an element $e \in G$ such that $ae = a = ea$.
> 4. *Existence of Inverses*. For each a $\in$ G, there exists an $a^{-1} \in$ G such that $aa^{-1} = e = a^{-1}a$.

A symmetry of an object is a *transformation* that leaves the object *apparently* unchanged ([30], p.28). Classic symmetry transformations are rigid motions, such as reflection, rotation, translation, and their combinations. For example, the appearance of the human body is invariant under reflection such that the distances from any origin point to its mirror image is preserved with respect to the reflection center.

Symmetry is more fundamentally about *invariant change*, i.e., change, yet the same. In geometric sense, it means that objects don't stretch or deform as they are translated or rotated. This basic idea extends the possibilities of symmetry beyond geometric objects. For example, symmetry principles in physics, such as *gauge symmetries*, obey symmetry properties but are not strictly geometric.

Rosen ([28], p.2) defines: "Symmetry is immunity to a possible change" in the context of the broad field of natural sciences. Immunity to the change means that some transformation brings the object into *coincidence with* itself. *Symmetry means invariant change or transformation invariant.* Rosen defines a symmetry transformation as a *bijective* (one-to-one and onto) mapping of a state to an image state *equivalent* to the object state ([28], p.80). State equivalence is defined by an equivalence relation for a state space of a system, such that any two states satisfy the conditions of reflexivity, symmetry, and transitivity. A symmetry transformation S can be denoted as:

$$u \xrightarrow{S} v \equiv u \qquad \text{or} \qquad S(u) = v \equiv u$$

for all states u ([28], p.80).

In the above definition, $v \equiv u$ denotes the equivalence relation between the states v and u. A subspace comprises a subset of states of a state space. An equivalence subspace is a subspace within which all the states are equivalent to each other. This leads to a general definition of symmetry group:

> The set of all invertible symmetry transformations of a state space of a system for an equivalence relation forms a group, a subgroup of the transformation

> group, called the *symmetry group* of the system for the equivalence relation ([28], p. 80).

A symmetry group doesn't comprise objects (buildings, software objects) of a system, but rather a set of symmetry transformations on these objects. The symmetries we investigate in this paper are those that are expressed by programming language constructs. The symmetry groups will be the compositions of symmetry transformations embodied in programming language features with respect to a certain set of invariants.

3 Alexander's Theories

Work on software patterns was largely inspired by Christopher Alexander's pattern research and application in the fields of architecture and urban design. We summarize Alexander's work here.

3.1 A Geometric Basis

As an architect, Alexander deals with geometry. Alexander views geometry to be the essence of what he was doing and this view is in line with Klein's:

> … in Klein's words, 'geometrical properties are characterised by their invariance under a group of transformations.' Each type of geometry has its own group … Group theory provides the common ground that links different geometries ([30], p. 44).

3.2 Pattern

Alexander's publications on architecture in the late 1970s focused on design elements called patterns [3]. Those patterns formed a pattern language, which can be partitioned into numerous smaller pattern languages [1]. The goal of pattern languages was to contribute to "the quality without a name," a deep feeling of architectural excellence suitable to a given culture

A pattern is a description of an architectural relationship between design parts. It is also an architectural transformation that integrates parts into a larger whole. Alexander's patterns are geometric in nature. A pattern language allows a builder to build a house by applying one pattern at a time, in sequence.

3.3 Theory of Geometric Centers

While the people using pattern languages had produced good community houses, Alexander soon realized that pattern languages alone were inadequate to achieve the beauty he sought. There were two problems: the economic processes of building didn't support piecemeal growth, and the people using the pattern language had the

skills necessary only for gross scales of beauty, not the very fine artisanship necessary for wholeness at all scales. These concerns prompted him to develop a new theory consistent with, but fundamentally different than, the theory of patterns. This theory is based on geometric *centers* and a piecemeal growth process. Informally, a center is something that draws the eye, a geometric region that we notice. Centers combine to form geometrically attractive configurations. Patterns usually are stereotypical centers that carry an aesthetic—likely culturally attuned—in addition to any visceral beauty they may have. Centers drive more at the visceral beauty of pure geometry.

The process for building with centers is a simple process *of structure-preserving transformations*. One finds the weakest center in a whole and strengthens it by adding new centers that make it more whole. If the overall result is more whole, then the process iterates to the next center. Each of these transformations increases wholeness while preserving the structure of the whole, though there are adjustments in the details. A given transformation can clean out a center that has become too messy, but the overall structure is preserved.

In mathematical terms, a structure-preserving transformation is called a *homomorphism* ([28], p.23). Briefly, a homomorphism is a many-to-one mapping from one group to another. If a homomorphism is bijective, i.e., one-to-one and onto, it is called an isomorphism. So an isomorphism is a bijective homomorphism. Although homomorphism preserves structure, homomorphic groups do not have the same structure unless they are isomorphic.

Each transformation strives to strengthen a center by reinforcing one or non-orthogonal structural properties in the whole, including 15 structural properties like Levels of Scale, Alternating Repetition, Local Symmetries, and Deep Interlock and Ambiguity [2]. We note that most of the structural properties are directly linked to symmetry. For example, *Deep Interlock and Ambiguity* and *Alternating Repetition* exhibit bilateral symmetry, and *Echoes* exhibit translational symmetry. Many of these structural properties show through in Alexander's patterns as *City-Country Fingers* [1] which is an example of *Deep Interlock and Ambiguity*. Alexander emphasizes this:

> There is a profound connection between the idea of a center, and the idea of symmetry.
>
> …
>
> 1. Most centers are symmetrical. This means that they have at least one bilateral symmetry.
>
> 2. Even when centers are *asymmetrical*, they are always composed of smaller elements or centers which *are* symmetrical ([4], p. 42-43).

4 Symmetry in Object-Oriented Software

In this section, we explore some examples of symmetry groups in object-oriented programming languages.

4.1 A Geometry Basis for Software

Gabriel posited in 1996 that geometry, for us in computer science, translates to the structure of the code ([7], p. 34). That is one geometry of a program, but there are others, including its modular [8] and temporal [9] structure. Coplien's more recent work [13] attempted to establish a geometric basis for C++ idioms for types whose operations have inverses. That effort was an attempt to bring some of the popular Design Patterns [16], which draw in part on those idioms, better in line with Alexander's geometric theories.

Here we attempt to explore symmetry in software from the perspectives of group theory. And interestingly, if one goes back before patterns to the very basics of polymorphism, one finds applicability of group theoretic foundations for object orientation to be strikingly strong despite the fact that no popular link ever joined the two fields. Consider this quote from a *plant physiologist* in a math journal circa 1986:

> …new theories of symmetry treat as equal also such objects (such equalities) which were considered as essentially different in previous theories (respectively, as inequalities). The unique reason why these equalities have been adopted is always the same thing, i.e. the existence of real or/and mental operations making the objects *O*, compared in features *F*, indistinguishable. ([32], p. 396)

This is almost a textbook definition of (object-oriented) polymorphism.

4.2 Classification as Symmetry

A class in an object-oriented program *defines* a symmetry. The common knowledge of the class concept is related to abstract data type and encapsulation. However, we can also say that a class classifies objects. A class establishes an invariance relationship between the class and its objects and makes all its objects *analogues* with respect to the class structure. Class data and functions remain valid to all the objects. A class in this sense defines an *analogy*, which is a symmetry ([28], p. 164).

We can validate the symmetry of a class using Rosen's definition. Recall from Section 2 that symmetry is immunity to a possible change. We can analyze the two aspects of change and invariance as follows:

1. *Change*. A class can be applied to more than one object; the objects to which it is applied can be switched from one to another.
2. *Invariance.* The class structure is immune to the switching-abouts of objects, i.e., what it is true remains true.

Recall also from Section 2 that changes are represented by transformations. The above shows that the order of the objects in a class is immaterial. Mathematically, we can represent the changing orders, switching-abouts as a set of transformations. We can show that these transformations form a group. For example, changing object o_1 to o_2 to o_3 is equivalent to changing from o_1 to o_3. This is the closure property of the

group. Changing is associative, in either direction (inverses); no changing is the identity transformation. A formal proof of these four properties will appear in a separate paper.

A linguistic realization of such transformations is the copy constructor, which satisfies the four properties of the group. More importantly, such transformations are *implicit*, and the fact that a user can create more than one object without worrying about the sequence of the creation owes to the symmetry of the class.

It should be noted that many language features—inheritance, subtyping, overloading, and others—are ways of expressing classification. By the above analysis, all such features are related to symmetries.

4.3 Inheritance as Symmetry

Cook and Palsberg [5] define inheritance as an operation on generators that transform a base class into a (distinct) derived class. In this section, we only consider one use of the inheritance operation, i.e., the use of inheritance for type derivation or subtyping. Only then can we say that inheritance preserves the structure of the base class. Other uses of inheritance, such as for implementation convenience or function overriding, do not preserve the class structure and therefore will not be considered here.

When inheritance is used for subtyping, it creates a type hierarchy [22]. A type hierarchy consists of subtypes and supertypes, where objects of a subtype provide all the behavior of objects of its supertype plus something extra [22]. A type hierarchy satisfies the Liskov Substitution Principle (LSP):

> If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, them S is a subtype of T. [22]

That is, there are invariants (in this case, behavioral invariants) that hold for a program under a transformation that substitutes objects of type *S* for those of type *T*. Behavior is symmetric with respect to subtyping—only here, the constancy of behavior is itself the litmus test by which subtyping is judged, not vice versa.

Given a subtyping path in a type hierarchy, we say that classes belonging to this path are *equivalent* in that they have the same invariants as the base class. We define all the classes in such a single path as a set or a state space as per Rosen's definition in Section 2. We define an equivalence relation for this set as the base invariant equivalence, denoted as $\equiv$, such that it holds for any pair of the classes, e.g., x, y, z, in the set:

1. *Reflexivity*. $x \equiv x$ for all x (Every class of the set is equivalent to itself).
2. *Symmetry*. $x \equiv y \Leftrightarrow y \equiv x$ for all x, y (Any pair of classes is equivalent).
3. *Transitivity*. $x \equiv y, y \equiv z \Rightarrow x \equiv z$ for all x, y, z.

In Section 2 we give the definition that a symmetry transformation is a *bijective* (one-to-one and onto) mapping of a state to an image state *equivalent* to the object state in a state space for an equivalence relation. Here the state space is a set of

classes of a given subtyping path. States in this state space correspond to classes. We represent an inheritance operation or derivation as D such that:

$$x \xrightarrow{D} y \equiv x \quad \text{or} \quad D(x) = y \equiv x$$

for all classes x in the set.

We now prove an inheritance operation is a symmetry transformation. Consequently, we need to prove (1) an inheritance operation is invertible and (2) the set of inheritance operations for a set of classes as defined above form a symmetry group.

To prove the invertibility of the inheritance operation, we need to show:

$$y \xrightarrow{D^{-1}} x \equiv y \quad \text{or} \quad D^{-1}(y) = x \equiv y$$

for all classes y in the set.

What is the inverse of the inheritance operation? Programming languages are graced with pragmatics that are not pure theory, and there is rarely a corresponding language realization of the operation that derives from a subtype to a supertype, i.e., a supertype cannot inherit from a subtype. However, for example, the (now vestigial) class slicing feature of C++ is a way to restore the original state of the system when an instance of some class is supplied in a context where a less derived class is expected. From this point of view, a slicing operation corresponds to or plays a role of inverse. In other programming languages, inverses might be implemented through conversion operators or constructors (e.g. in Smalltalk, many algebraic types respond to messages such as `asInteger`).

Since our goal is to establish a formalism for patterns, we can still define an inverse of an inheritance operation as a mathematical operation whose linguistic support is missing for practical reasons. Hence we have the inheritance operation whose inverse remains to be mathematical.

We shall now prove that the set of all invertible inheritance operations on the classes in a given subtyping path forms a symmetry group.

1. *Inverses* exist in the sense of the above assumption.
2. *Closure* follows from transitivity of the equivalence relation. For the inheritance operation D on all classes x, y, and z, the composition of DD is the result of consecutive application of D, such that:

   $x \equiv y = D(x)$,
   $y \equiv z = D(y) = D(D(x)) = (DD)(x)$,
   $x \equiv z = (DD)(x)$,
   $x \equiv z = D^2(x)$

   Thus for the inheritance operation D their compositions D^2, D^3, … are also inheritance operations.

3. *Associativety* holds for composition by consecutive application. It is evident that using the closure property, we can obtain:
   $D(DD) = (DD)D = D^3$
   Hence we have proved the associativity.
4. The *identity* transformation is a null inheritance operation. In other word, it is a do-nothing operation, such that:
   $x \xrightarrow{I} x \equiv x$ or $I(x) = x \equiv x$
   for all classes x in the set.

All the invertible inheritance operations on the classes through a single subtyping path form a symmetry group for class invariants. We may consider inheritance operations as *translations that transform classes in time* [33]. We can represent such translational symmetries as a linear train of iterations:

$$... D^{-3} = D^{-1}D^{-1}D^{-1}, D^{-2} = D^{-1}D^{-1}, D^{-1}, D^0 = I, D^1 = D, D^2 = DD, D^3 = DDD, ...$$

Translation and other temporal symmetries (transformation over time) are common themes in computer programs, as we shall also see in Section 4.4. In fact, temporal symmetry is a fundamental phenomenon in symmetry breaking in the natural sciences. For example, the idea that time is reversible is called the *T symmetry* in the classic CPT symmetry model of physics; symmetry breaking in all of these symmetries is the very reason that *anything* exists ([19], pp. 79-81).

Generalizing the ideas of classification and temporal symmetry, we can define symmetry in terms of any invariant or notion of equality ([32], pp. 395-396) for a system under consideration.

4.4 Overloaded Operators as Symmetries

Operator overloading, and associated friendship relationships in C++, have consciously been provided to express symmetry ([21], p749). Overloaded arithmetic operators support *reflection symmetry*, freeing the user from distinguishing between the left and the right operands, as A+B = B+A and A*B = B*A.

Overloaded operators not only can support reflection, but also *translation* as well. For example, overloaded “+”, “-”, “*”, “/” free the user from distinguishing between primitive numbers and objects.

5 Symmetry Breaking and Pattern

When a system encounters stress it loses symmetry. The phenomenon of symmetry breaking may be explained in terms of symmetry groups. Consider the common example of a proto-planet rotating in space, a sphere of symmetry group *O(3)*. If it spins too fast it may become pear-shaped, losing one degree of symmetry so as to fall into *O(2)*. If the spinning continues it may break into a planet/moon system which still has overall symmetry *O(2)*, though it has lost the spherical symmetry of the original system. This is called *spontaneous symmetry breaking* in physics [24], or symmetry-breaking as we call it here. Symmetry doesn't really "break", but is just

reduced or redistributed in the effect produced by symmetry in the cause. When a symmetry group is broken, it results in a new group that is a subgroup of the original. Note in this moon-formation example, each of the *parts* still has *O(3)* symmetry, even though the *system* has been reduced to *O(2)*.

Symmetry breaking results in patterns, or more precisely, reveals patterns, for too much symmetry isn't perceived as pattern [30]. For example, a regular n-sided polygon belongs to the *dihedral group* D_n. When n is infinitely large, the polygon is close to a circle, which isn't rich in structure. But if some of the symmetries in this polygon break, or when n becomes smaller, say, 6, 5, 4, or 3, we begin to see the shape of the polygon. The phenomenon of symmetry-breaking was first discovered in 1923 by Ingram Taylor when studied the dynamics of the fluid flow, which was known as hydrodynamic "symmetry paradox":

> This paradox, that symmetry can get lost between cause and effect, is called *symmetry-breaking*. In recent years scientists and mathematicians have begun to realize that it plays a major role in the formation of patterns.
>
> …
>
> From the smallest scales to the largest, many of nature's patterns are a result of broken symmetry; and our aim is to open your eyes to their mathematical unity. ([30], p. xviii)

The idea of symmetry breaking carries through to Alexander's patterns. *Varied Ceiling Heights* [1] is a symmetry-breaking pattern. *Light on Two Sides of Every Room* [1] is another example: a perfectly symmetric room would have either no windows or would have windows on four sides. A room with windows on four sides would be perfectly symmetric and would lack the quality Alexander seeks in his work. He remarked that too much symmetry is a bad thing:

> Living things, though often symmetrical, rarely have perfect symmetry. Indeed perfect symmetry is often a mark of death in things, rather than life.
>
> I believe the lack of clarity in the subject has arisen because of a failure to distinguish overall symmetry from local symmetries. ([2], *The Phenomenon of Life*, 44.)

and:

> In general, a large symmetry of the simplified neoclassicist type rarely contributes to the life of a thing, because in any complex whole in the world, there are nearly always complex, asymmetrical forces at work—matters of location, and context, and function—which require that symmetry be broken. ([2], *The Phenomenon of Life*, 45.)

Symmetry breaking also plays an important role in the characterization of Alexander's 15 structural properties. *Local Symmetry* requires some symmetry be broken. *Roughness, Gradients, Echoes*, and *Levels of Scale* all exhibit symmetry, but lack perfect symmetry characteristic of the next largest symmetry groups (perfect smoothness, bilateral symmetry, and equal size).

The idea of symmetry breaking also extends to software patterns, as we shall see in Sections 6 and 7. For example, class extension is a symmetry; *Bridge* is a pattern that reflects symmetry breaking under the stress of certain design desiderata.

6 Patterns and Symmetry Breaking in Software

As discussed in Section 4, many programming language constructs express symmetry. When those constructs fail to solve design problems, programmers often resort to patterns to express the design in the programming language. This section attempts to show that most software patterns are a result of symmetry breaking.

6.1 Breaking Type Symmetry

Assume we have a class, `List<T>` in some library. We wish to derive from it a new type, `Set<T>`. Let's see if such derivation can preserve class invariants in `List<T>`, as discussed in Section 4.3.

First, a set is not a subtype of a list, nor is the reverse true. For example, by the principle of extensionability, sets with the same members are equal [17]. So adding the same member twice to a set is the same as adding the member only once. However, adding the same member twice to a list is different from a single insertion.

Therefore, deriving a set from a list breaks the symmetry that subtyping attempts to conserve. We start with a List and want to transform it into a Set. The symmetry breaks. Where does it go?

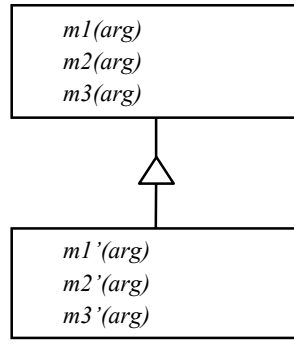


Fig. 1. Inheritance Symmetry.

Imagine if the symmetry did not break, i.e., in the normal subtyping situation, the relationship between a subtype and its supertype would be denoted as in Figure 1. However, since the symmetry is broken, the new relationship lacks the symmetry of Figure 1, reflecting only local symmetry (Figure 2). This is known as the Adapter pattern. In this pattern, the `insert` and `has` methods appear in all three classes.

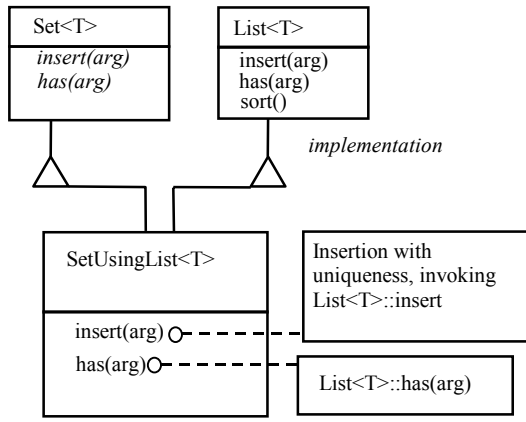


Fig. 2. Adapter Pattern.

Most of the structural patterns (*Adapter, Bridge, Composite, Decorator*, and *Proxy*) [16] deal with the tension between language constructs that express symmetries, and the small perturbations that make those constructs unsuitable for use. Some of the behavioral patterns (*Iterator, Memento, Observer, State, Strategy, Template Method*, and *Visitor*) can also be easily described in terms of temporal symmetry breaking. Other design patterns take more imagination to describe in symmetry-breaking terms, which raises the question of whether they are patterns in the Alexandrian vein.

From the perspective of symmetry breaking, a pattern is a way of redistributing the forces in a symmetric system in light of some instability. Symmetry is reduced, but not lost; it is redistributed in the pattern. The exact way in which it is redistributed is difficult to predict. Global symmetries are lost, but local symmetries are preserved.

We want to point out here that the term *local symmetry* is not an original Alexandrian formulation, but is a standard term in the symmetry group communities of crystallography ([23], p. 29; [29], p. 567) and quantum physics ([24], p. 173).

6.2 Breaking Function Symmetry

Multiple dispatch is a form of symmetry breaking. In a Liskov type hierarchy, subtype methods have a level of equivalent semantics defined by the bounds of pre- and postconditions, semantics that programmers usually associate (however informally) with the method selector. In a classful language these symmetries in a type hierarchy preserve class invariants, as discussed in Section 4.3.

When a type hierarchy is used for dynamic binding, a member function is chosen according to the type of its instance at run time. The symmetry breaks if the member

function depends on the types of more than one object; the LSP no longer holds. . This is a form of symmetry breaking for which the longstanding linguistic solution is *multiple dispatch*. This of course is the *Visitor* pattern [16].

The "Promote and Add" pattern is also symmetry breaking. The "+" operator is interesting not only because it supports symmetries as discussed in Section 4.4, but also because it has a highly symmetric signature:

$$T \, x \, T \rightarrow T$$

If we picture "+" operating between pairs of objects, where each object participates in a class hierarchy, we get a nicely symmetric picture as in Figure 3.

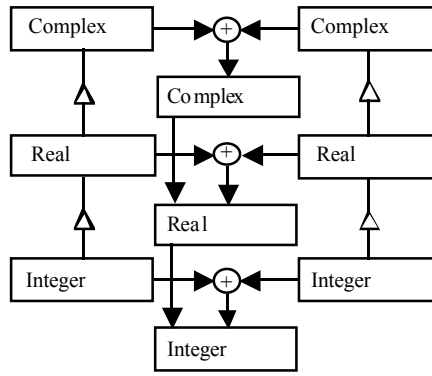


Figure 3. Bilateral Symmetry

But reality is messier because of the need for heterogeneous addition. For example, it makes sense to ask for 1 + 4.5, as in Figure 4—a broken symmetry (and it's independent of multiple dispatch or polymorphism).

The solution is a geometric transformation, a pattern, called *Promote and Add*, originally written up as a *C++* idiom [13], but which can also appear in broad architectural contexts as patterns such as Add a Switch [9]. It adds local symmetries by promoting heir types to their parent types as a prelude to addition, thereby reducing the problem to that of the first picture. It is a structure-preserving transformation that adds centers (the transformations from derived types to base types) to the first picture. The resulting Smalltalk bears out the pattern explicitly in methods such as `sumFromInteger`, `asDouble`, and `ArithmeticValue>>retry`.

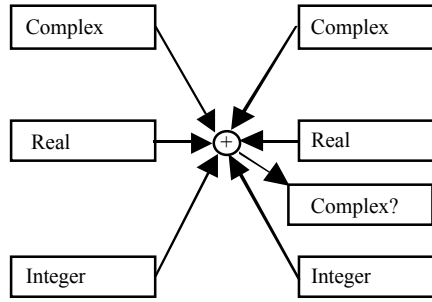


Figure 4. Symmetry Breaking in +

6.3 Breaking Class Symmetry

We usually think of types as abstract specifications with classes as their language realizations. The class structure often follows the type structure. But sometimes, even though the type structure reflects a structure-preserving transformation consistent with the LSP, the class structure does not preserve structure for reasons that owe to language implementation peculiarities. This means that the symmetry of the class structure is broken, and a pattern usually results.

Again consider class `Complex`, whose implementation is two real numbers. Abstractly we can talk of the symmetry group of *type* `Complex` as comprising transformations that preserve class invariants as discussed in Section 4.2. These invariants hold, with respect to member function behavior (type symmetry) for heir classes such as `Real` and `Integer`, as one would expect.

Even though the member functions are preserved in the transformations from `Complex` to derived *types*, the structure of the `Complex` *class* may not be preserved in `Real` and `Integer`. In particular, `Complex` has two reals and `Real` would have only one. Eiffel would call this restriction inheritance ([27], p. 826). In the representation of the class, there is at best a weakened symmetry between `Complex` and `Real` (one could argue that they are symmetric with respect to the preservation of the real component) but all structure is lost by the time we get to `Integer`. Symmetry slowly breaks as we descend the hierarchy. We find the same case for `Ellipse` and `Circle`, as in Meyer ([27], p. 467).

In either of the cases, we lose symmetry when the inheritance transformation is not structure preserving as required in the LSP, so we have symmetry breaking. What results is usually a pattern; in this case, something like *Bridge* might be in order.

7 Other Examples

We can look beyond object-oriented programming language constructs to find many examples of symmetries and symmetry breaking in software design.

Symmetries abound in programs and can be found beneath almost all programming structures. In addition to symmetries we have discussed in Section 4, loops are spiral time symmetries. Simple conditionals can be viewed as symmetry breaking, while case statements can also be viewed as symmetries that hold the entry and exit points invariant. Argument defaulting is a form of overloading where the symmetry is the preservation of some argument types. Rather than focusing on these programming-in-the-small symmetries, in this paper we focus on the design structures that are of greater interest to the field of object orientation.

Patterns are introduced as a result of symmetry breaking in all of these cases. We showed above, in Section 6.2.2, what happened when the symmetry of overloaded operator + broke. This is hardly an object-oriented phenomenon, either. The work in [10] introduced the notion that functional and applicative languages may succumb to these analyses more readily than object-oriented languages do, owing to the largely geometric nature of the source and translated program structures.

The symmetries supported by language constructs might be used as a basis for objectively comparing the relative expressive power of a programming language. For example, the number of symmetries and the type of symmetry might be used as such an objective indicator. We suggest further empirical work in this area.

We find patterns outside the programming language context, too. Patterns such as *Half Object plus Protocol* (HOPP) [25] are an obvious example of symmetry-breaking reminiscent of bifurcation symmetry. It should be noted that there is nothing fundamental about the system in which HOPP is embedded that suggests that symmetry break in a way that results in two objects; it could just as well break into three objects, as it does in the pattern *Three-Part Call Processing* [9].

In the *Cascade* pattern for a public transport system [34], it was observed that both the driver duty object and its builder were cascades. A driver duty can be seen as a result of a translation of its builder (the Prototype pattern) or as a rotated bilateral symmetry. Other examples in telecommunication software are documented in [9].

Model-View-Controller expresses symmetry between a *Model* object and a *User* object. What breaks the symmetry is the need for the *User* object to maintain multiple views of the *Model*. That gives rise to the *View* object itself, and to the *Controller* as a dispatcher from the *User* to the *Model* and *View*. [Trygve Reenskaug, Personal conversation, June 1999].

Fresh Work before Stale [26] is a classic example of temporal symmetry breaking. The pattern creates a broken symmetry in the flow of work that gives priority to new work over pending work, and throughput increases as a result.

Factory Method and *Abstract Factory* [16] break the symmetry of the structures of the object produced by instantiation processes. *Template Method* breaks the symmetry of individual algorithms.

Domain analysis techniques are rooted in axioms of commonality and variation [11]. Group theory may provide constructs for formalizing this structure as well: symmetry is about commonality invariance; symmetry breaking is about variations.

8 Towards a Pattern Formalism

So, what is a pattern? The question is the closest thing to a religious debate in computer science since the debate raged about "What is object orientation?" There are many qualities often used to distinguish, or sometimes distance, patterns from mainstream academic computer science. Among these are beauty, maturity of ideas, and attunement to human comfort. Here we try to put these elements in perspective.

Does Alexander's stipulation that a pattern only exists within a pattern language ([3], p. 312) relate to symmetry and symmetry breaking? Remember that a pattern is a transformation, a transformation that breaks some symmetry present in the original system, adding asymmetry. The original system defines context; without that context, there may in fact be no symmetry to be broken. What might otherwise pass as a pattern, out of context, may simply be a symmetry. We can also explain the relationship between symmetry and symmetry breaking as a causal relationship: the original system exhibiting the original symmetry is the cause; the transformed system in which some symmetry is broken is an effect.

As mentioned early, broken symmetry doesn't necessarily mean asymmetry unless all the symmetry is lost. Although it is not the topic of this article, we wish to point out that symmetry only exists with respect to asymmetry ([28], p. 161) and asymmetry is a frame of reference to symmetry.

One may criticize the design patterns [16] as not meeting the Alexandrian criterion of necessarily emanating from a whole, from a pattern language. However, some of the design patterns constitute patterns that form a pattern language for a particular design problem. Symmetries that these patterns attempt to preserve or break may be identified. This leaves hope that the design patterns may be legitimized as patterns in a fashion tantamount to, or at least analogous to, the closely related idioms work [13].

One key suggestion for future work is to show that a pattern language forms a contextual framework for the formalization of symmetry breaking (the very definition of individual patterns), and that such a framework forms not only a language, but also an algebra of specification. The outcome seems likely; after all, a geometry is essentially an algebra of symmetries.

Based on the above analysis, we posit the following formal definition of a pattern:

> **Definition**: A pattern characterizes a structure resulted from breaking an original symmetry, where the symmetry is defined in terms of an invariant in a system. Examples of such invariants include class structure (in the vulgar sense as in Section 4.2) and behavior (subtyping as discussed in Section 4.3).
>
> In other words, a pattern represents a symmetry effect that is less symmetric than the symmetry cause, where the symmetry cause is produced by a programming language construct. Patterns precipitate from symmetries in response to both internal and external forces that are the analogy of instability.
>
> A pattern is a design solution that language constructs fail to provide. A pattern language defines a composition law for composing patterns in a specific way to solve a larger design problem. A pattern is in relation to a symmetry that in turn is in relation to a geometric context: a symmetry group.
>
> The formal aspect of the definition is necessary but not sufficient. The articulation of a pattern is also subject to a value system, and in particular to

aesthetic and quality considerations. Among these considerations are human comfort, utility, beauty, durability, and maintainability.

This definition is simply a distillation of the findings of this paper, cast in a way that ties the group theoretic foundations with the vernacular definitions. We propose it as a foundation for further work in the formalization of software patterns.

One might use this definition to conclude that because the design patterns reflect different geometric contexts, they could never all be unified in a single pattern language. For example, the context for Observer and Memento is temporal symmetry while the context for most other patterns is more spatial. This contextual modeling might provide a framework for a pattern taxonomy that would be more useful than the current taxonomy (creational, structural, etc.) for the stated purpose of a pattern language: to create whole systems through a piecemeal growth process.

9 Conclusion and Acknowledgements

Other attempts to formalize patterns have striven for implementation automation. That is not our goal here, as automation of pattern assembly is an oxymoron from first principles. Our goal is to provide a formal basis for the characterization of patterns in specific contexts. For example, the symmetry theory foundation for patterns explains why multiple dispatch is a pattern in Smalltalk, and is not a pattern in CLOS. It explains why inheritance for subtyping is a symmetry, lacking the symmetry-breaking properties that are characteristic of patterns, even though Inheritance wrongly appears as a pattern in some collections [31]. This model may provide a foundation for legitimizing other patterns on the basis of temporal symmetry breaking models, even hopelessly non-spatial patterns.

This paper deals with patterns commonly associated with design. There may be other useful patterns of programming, such as indentation style and other modular constructs, that also exhibit symmetry and symmetry breaking and which are equally important to the success of software development.

Ralph Johnson provided initial feedback to the original draft and has continuously provided valuable insights and comments to various versions of the paper. We are also deeply indebted to Michael Benedikt, Ellie D'Hondt, Maja D'Hondt, Ed Remmel, Curtis Tuckey, Gottfried Chen, Edith Adan-Bante, Peter Mataga, Rachel Suddeth, Kevlin Henney, Neil Harrison, Rich Delzenero, Christoph Koegl, and Peter Grogono.

References

[1] Alexander, C., et al. A Pattern Language. New York: Oxford, ©1977.

[2] Alexander, C. The Nature of Order. Pending publication by Oxford, New York, New York. Citations quoted with permission.

[3] Alexander, C. The Timeless Way of Building. New York: Oxford, ©1979.

[4] Alexander, C. A Foreshadowing of 21st Century Art: The Color and Geometry of Very Early Turkish Carpets. New York: Oxford University Press, ©1993.

[5] Cook, W., and J. Palsberg. A Denotational Semantics of Inheritance and its Correctness. IN OOPSLA '89 Conference Proceedings, SIGPLAN Notices 24(10), 1989. New York: ACM SIGPLAN, p. 436.

[6] Coplien, J. Advanced C++ Programming Styles and Idioms. Reading, MA: Addison-Wesley, ©1992.

[7] Coplien, J. Software Patterns. New York: SIGS Publications, ©1996.

[8] Coplien, J. Space: The Final Frontier. C++ Report 10(3). New York: SIGS Publications, March 1998, 11-17.

[9] Coplien, J. Worth a Thousand Words. C++ Report 10(5). New York: SIGS Publications, May/June 1998, ff. 54.

[10] Coplien, J. To Iterate is Human, to Recurse, Divine. C++ Report 10(7). New York: SIGS Publications, July/August 1998, 43-48; 51.

[11] Coplien, J. Multi-Paradigm Design for C++. Addison Wesley, Reading MA. 1999. ISBN 0-201-82467-1

[12] Coplien, J. Take Me Out to the Ball Game. C++ Report 11(5). New York: SIGS Publications, May 1999, 52-8.

[13] Coplien, J. C++ Idioms Patterns. In Pattern Languages of Program Design 4. Reading, MA: Addison-Wesley, ©2000.

[14] Eden, A. H., J. Gil, A. Yehudai. A Formal Language for Design Patterns. 3rd Annual Conference on the Pattern Languages of Programs (Washington University Technical Report WUCS-97-07).

[15] Eden. A. H., J. Gil, A. Yehudai. Precise Specification and Automatic Application of Design Patterns. The Twelfth IEEE International Automated Software Engineering Conference (ASE 1997).

[16] Gamma, E., et al. Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley, ©1996.

[17] Gordon, C. and N. Hindman. Elementary Set Theory. Hafner Press, 1975.

[18] Grenander, U. Elements of Pattern Theory. Baltimore, Maryland: Johns Hopkins University Press, ©1996.

[19] Hawking, S. A Brief History of Time. New York: Bantam, ©1996.

[20] Kappraff, J. Connections: The Geometric Bridge Between Art and Science. McGraw-Hill, 1990.

[21] Lippman, S. B., and J. Lajoie. C++ Primer. 3rd Ed. Addison-Wesley, 1998.

[22] Liskov, B. Data Abstraction and Hierarchy. SIGPLAN Notices 23,5, May 1988, p. 25.

[23] Mackay, A. L. Generalized Crystallography. Computers & Mathematics With Applications, 12B(1/2). Exeter, UK: Pergamon Press, 1986.

[24] Mannheim, P. D. Symmetry and Spontaneously Broken Symmetry in the Physics of Elementary Particles. In Computers and Mathematics with Applications, 12B(1/2). Exeter, UK: Pergamon Press, 1986, 169-183.

[25] Meszaros, G. Pattern: Half-Object plus Protocol (HOPP). In J. O. Coplien and D. Schmidt, eds., Pattern Languages of Program Design, Reading, MA: Addison-Wesley, ©1996, Chapter 8, 129-132.

[26] Meszaros, G. A Pattern Language for Improving the Capacity of Reactive Systems. In Pattern Languages of Program Design - 2. Reading, MA: Addison-Wesley, ©1998, p. 586, "Fresh Work Before Stale."

[27] Meyer, B. Object-Oriented Software Construction. Upper Saddle River, NJ: Prentice-Hall, ©1997.

[28] Rosen, J. Symmetry in Science: An Introduction to the General Theory. New York: Springer-Verlag, 1995.

[29] Senechal, M. Geometry and Crystal Symmetry. In Computers and Mathematics with Applications 12B(1/2). Exeter, UK: Pergamon Press, 1986.

[30] Stewart, I., and M. Golubitsky. Fearful Symmetry: Is God a Geometer? London: Penguin, ©1992, 1993.

[31] Tichy, Walter. Essential Software Design Patterns, http://wwwipd.ira.uka.de/~tichy/patterns/overview.html, n.d.

[32] Urmantsev, Y. Symmetry of System and System of Symmetry. Computers and Mathematics with Applications, 12B(1/2). Exeter, UK: Pergamon Press, 1986, 379-405.

[33] Weyl, H. Symmetry. Princeton University Press, 1952.

[34] Zhao, L, and T. Foster. Driver Duty Constructor: A Pattern for Public Transport Systems. In The Journal of Object-Oriented Programming 12(2), May 1999, 45-51; 77.

Aspect Composition Applying the Design by Contract Principle

Herbert Klaeren[1], Elke Pulvermüller[2], Awais Rashid[3], and Andreas Speck[1]

[1] Wilhelm-Schickard-Institut für Informatik, Universität Tübingen
D-72076 Tübingen, Germany
{klaeren,speck}@informatik.uni-tuebingen.de
[2] Institut für Programmstrukturen und Datenorganisation
University of Karlsruhe, D-76128 Karlsruhe, Germany
pulvermueller@acm.org
[3] Computing Department, Lancaster University
Lancaster LA1 4YR, UK
marash@comp.lancs.ac.uk

Abstract. The composition of software units has been one of the main research topics in computer science. This paper addresses the composition validation problem evolving in this context. It focuses on the composition for a certain kind of units called aspects. Aspects are a new concept which is introduced by aspect-oriented programming aiming at a better separation of concerns. Cross-cutting code is captured and localised in these aspects. Some of the cross-cutting features which are expressed in aspects cannot be woven with other features into the same application since two features could be mutually exclusive. With a growing number of aspects, manual control of these dependencies becomes error-prone or even impossible. We show how assertions can be useful in this respect to support the software developer.

1 Introduction

Composing systems from individual units has been one of the main research goals in the software engineering discipline since its beginning in the 1960s. The first approaches concentrating on modules were followed by decomposition into objects and / or components [23,4,30,29]. All of these approaches aim at managing complexity by dividing a system into smaller pieces.

Once a systems analyst completes the creative process of dividing the system into manageable parts, phases follow where the units are realised and synthesised to form the final system. The units themselves are built from scratch or reused if they already exist provided they can be used in the chosen technical environment.

It has become popular to build systems not only for one purpose but to allow variability to a certain extent. This is due to the goal to save development and maintenance effort and increase software quality by reusing a system multiple times even in slightly different contexts. This development is reflected in the intensified research in the field of product line architectures (PLA) [27]. In a

G. Butler and S. Jarzabek (Eds.): GCSE 2000, LNCS 2177, pp. 57–69, 2001.

concrete context the necessary and available units are reused and the points of variability are adapted to the particular needs [10,8]. This results in a set of valid configurations for one kind of system. The more configurations exist the more flexible is the system concerning reuse and adaptation. However, an increase in variability and flexibility poses a greater challenge in validating the configuration of chosen units. The search for a valid configuration is error-prone if manually practised. Therefore, there is a need for computer support. Such computer support is also realisable since rules can be used to specify valid configurations in advance.

Aspect-oriented programming (AOP) introduces a new concept called aspects [17]. Besides classes, they form an additional type of system unit. Aspects serve to localise any cross-cutting code, i.e. code which cannot be encapsulated within one class but which is tangled over many classes. The existence of this new type of system unit affects the composition validation. Although some composition validation mechanisms already exist [3,7], aspects have not been explicitly considered. In this paper we introduce a composition validation mechanism which in particular concentrates on this new kind of system units.

In our approach we use the technique of assertions [21]. Assertions are a widely accepted programming technique for complex systems to improve their correctness. Taking the validity of a configuration as a property, assertions can be used to ensure that the chosen aspects and classes fit together. In the following, we provide further background information about aspect-oriented programming and assertions which is necessary to understand the remainder. A motivating example demonstrates the problem we address. Sections 3 to 5 describe our different aspect composition validation approaches based on assertions. Section 6 concludes the paper and identifies directions for future work.

2 Background and Motivation

This paper is based on two techniques, i.e. aspect-oriented programming and assertions. Here, assertions are used as a means to validate aspect composition. In the following, the term "aspect composition" is used for the insertion of a set of aspects into one system / class. This set is also called (aspect) configuration.

Aspect-oriented programming (AOP) [17] introduces a new concept, called aspects, in order to improve separation of concerns. It aims at easing program development by providing further support for modularisation at both design and implementation level. Objects are designed and coded separately from code that cross-cuts the objects. The latter is code which implements a non-functional feature that cannot be localised by means of object-oriented programming. Code for debugging purposes or for object synchronisation, for instance, is tangled over all classes whose instances have to be debugged or synchronised, respectively. Although patterns [12] can help to deal with such cross-cutting code by providing a guideline for a good structure, they are not available or suitable for all cases and mostly provide only partial solutions to the code tangling problem. With AOP, such cross-cutting code is encapsulated into separate constructs or

system units, i.e. the aspects. The links between objects and aspects are expressed by explicit or implicit *join points*. An *aspect weaver* is responsible for merging the objects and the aspects. This can be done statically as a phase at compile-time or dynamically at run-time [17,15]. For our research, we have used AspectJ0.4beta7 from Xerox PARC [32,18] for all AOP implementations together with Java JDK 2.

Although AOP allows to achieve a better separation of concerns, the application of aspect-oriented programming bears new challenges. This paper addresses one of them: aspect composition validation.

Let us assume we have an application and a set of aspects which can be woven with the objects of the application. This set may contain aspects which are redundant or even exclusive with respect to other aspects in the set. Alternatively, weaving one aspect may require another aspect to be woven. For example, if there are two different aspects `A1` and `A2` containing debugging functionality, they may be mutually exclusive. `A1` inserts code into the application which realises all tracing. Each function call is recorded in a window with graphical support for user control. As opposed to graphical debug support, `A2` realises the same tracing functionality as ASCII output. Since both aspects implement the same feature (although in different ways) it wouldn't be reasonable to weave both aspects. Depending on the context, the environment or the requirements (e.g. towards efficiency), the one or the other is more suitable.

In [25,28] an example implementation can be found which also reveals this aspect composition problem.

At this point, assertions [21,26] can be used to achieve correctness. The basic ideas behind assertions originate in the theory of formal program validation pioneered by R.W. Floyd [11] C.A.R. Hoare [14] and E.W. Djikstra [9]. The Hoare triples provide a mathematical notation to express correctness formulae. Such a correctness formula is an expression of the form:

$$\{P\}A\{Q\} \tag{1}$$

This means that *"any execution of A, starting in a state where P holds, will terminate in a state where Q holds"* [21]. In terms of software, A denotes an operation or software element whereas P and Q define the assertions, or pre- and postcondition, respectively. By means of assertions it becomes possible to state precisely the formal agreement between caller / client and callee / supplier, i.e. both, what is expected from and what is guaranteed to the other side. This reflects the "Design by Contract" principle [20].

For our purposes, the validity of a set of aspects which are woven in A reflects an assertion.

3 Aspect Composition Validation

Since correctness is always relative [1] [21] we assume to have a certain specification defining (among other things) which features have to be realised in the

[1] This is also expressed by P in equation (1).

application. Therefore, the decision which feature (i.e. here which aspect) should be inserted (i.e. woven) is not part of our solution. Moreover, the knowledge whether a set of aspects is valid (in this paper also called valid configuration) or not, has to be extracted during the analysis phase and captured in the specification (e.g. using finite state automata as outlined in [24]) before assertions can be useful to prove this property.

According to the definition of assertions, we use them to verify that a class or an application contains a suitable set of aspects which forms a valid configuration (with respect to what is written in the specification). With these assertions, the specification is expressed and included in the implementation to assess the correctness of aspect composition. In this paper, we concentrate on this issue of software correctness and omit all considerations about other properties which can be asserted as already known [13].

Expressing the part of the specification concerning the aspect composition within the code can serve various purposes. These range (similar to other kinds of assertions) from treating assertions purely as comments with no effect at run-time to checking the assertions during execution.

Sections 3.1 and 3.2 describe different ways to realise assertions checking for a valid aspect configuration.

3.1 Precondition, Postcondition, or Invariant

B. Meyer defines an assertion as *"an expression involving some entities of the software, and stating a property that these entities may satisfy at certain stages of software execution"* [21]. In our case, the entities of the software involved are the classes or methods whereas the stated property is the validity of the aspect configuration injected into these classes or methods through the aspect weaver.

Three possibilities to assert this property can be identified: preconditions, postconditions or class invariants [21].

First, this property can be asserted as a precondition. The class or method starts its work assuming that a valid set of aspects is woven and active. The precondition checks whether this assumption is true, i.e. whether the contract is fulfilled by the caller and the property to have a valid aspect configuration in the callee holds (cf. figure 1 on the left). This may be reasonable if the caller changes the aspect configuration which is active in the callee (here we assume that the aspects themselves are not changed, i.e. there is no aspect evolution). Applying the design by contract principle, the caller ensures that the changed aspect configuration is valid for the callee. In the precondition the callee checks whether the caller kept the contract.

Although this is possible, we propose [2] to assert the validity of an active aspect configuration in postconditions or class invariants. While pre- and postconditions describe the properties of individual routines, class invariants allow to express global properties of the instances of a class which must be preserved by

[2] It should be avoided to assert a property multiple times due to the disadvantages of redundant checks and defensive programming [21].

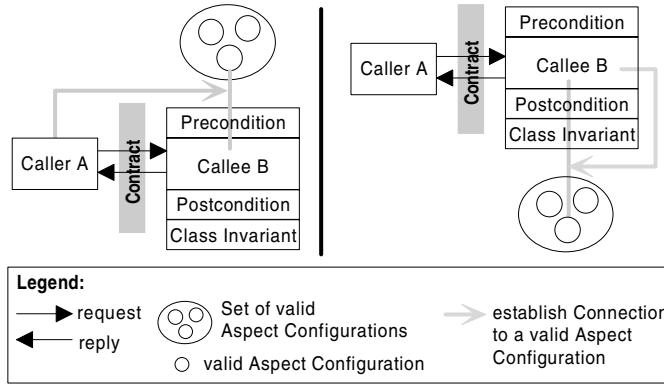


Fig. 1. Different Possibilities to Assert a Valid Aspect Configuration.

all routines. An invariant for a class is a set of assertions (i.e. invariant clauses) that every instance of this class will satisfy at all times when the state is observable. The crucial point is that with both, postconditions and invariants, it is a bug in the callee if the assertion is violated [21]. We believe that in most cases it is more reasonable that the decision about which aspect should be active in a class C at which times is in the responsibility of C. Thus, in our understanding, the callee should decide by itself which aspect instances should be connected to this callee. Consequently, as opposed to a precondition realisation, the callee is also responsible to ensure and to check that the woven aspects which are active in the callee form a valid configuration. Obviously the same argumentation applies symmetrically to the caller which can also be a callee.

Thus, the remainder of the paper concentrates on postconditions and class invariants. It goes without saying that the principles shown are also applicable to preconditions.

3.2 Static or Dynamic

There are two ways to assess software correctness. The property of a class or method to result in a valid aspect configuration can be ensured at run-time (dynamically) or alternatively at (or before) weave-time (statically).

A similar distinction can be identified if the times of aspect configuration changes are considered. Either it is possible to add or remove aspect instances to or from class instances only statically or even dynamically (cf. table 1; + indicates that this combination is possible whereas ++ expresses that this combination is preferable provided there is a choice).

In the following we show the outlined differences between *static* and *dynamic* at some AspectJ0.4beta7 / Java JDK 2 code extracts (cf. figure 2).

In AspectJ0.4beta7, each aspect has (like classes) its name following the keyword `aspect` and contains the advised methods with their names and the

Table 1. Static and Dynamic Change and Validation.

		Change of Aspect Configuration	
		static	dynamic
Aspect Composition Validation	dynamic	+	+
	static	++	usually not possible

class names referring to these methods. Besides method advising [3], it is also possible to introduce whole methods.

Dynamic or Static Change of Aspect Configuration
At first, the change of the set of aspects which play an active role in a class instance can be done at weave-time without later changes during execution time. This situation is depicted in figure 2 on the left. These static connections between an aspect and all instances of a class are expressed with the keyword `static` in AspectJ. Once the weaver injected these static aspects into class `StaticExample` at weave-time, the functionality within these aspects will be active in all instances of `StaticExample` during the whole execution time. The woven aspects augment the class code of `StaticExample` which impacts all its instances. Moreover, it is not possible either to add further static aspects nor to remove statically woven aspects from one or all instances during run-time.

As opposed to statically woven aspects, it is also possible to create new aspect instances and connect them to objects during run-time. In figure 2, this is shown on the right. The code within the light-grey lined rectangle in the `DynamicExample` class [4] establishes the connection between aspects and class instance during run-time. The commands used (e.g. `addObject(...)`) are provided by AspectJ0.4beta7. If other AOP environments are used, similar language constructs are necessary. Alternatively, dynamic weaving capabilities can be used if available [15].

Dynamic or Static Aspect Composition Validation
Assertions express correctness conditions (here: validity of the aspect configuration). Assertion rules check whether such a condition is violated. These rules can be executed during run-time. If possible, a violation check can also be done statically. This requires to have all the necessary information at compile-time or weave-time which is true for statically woven aspects. During execution the static aspect configuration does not change. Verifying and removing assertions statically has the advantage that the overhead of the test during execution is avoided

[3] By using the keyword `before` the aspect weaver injects this advised code at the beginning of the method. The keyword `after` augments the method at the end.

[4] For demonstration purposes we have chosen to present an aspect-directional design [16], i.e. the class knows about the aspect but not vice-versa.

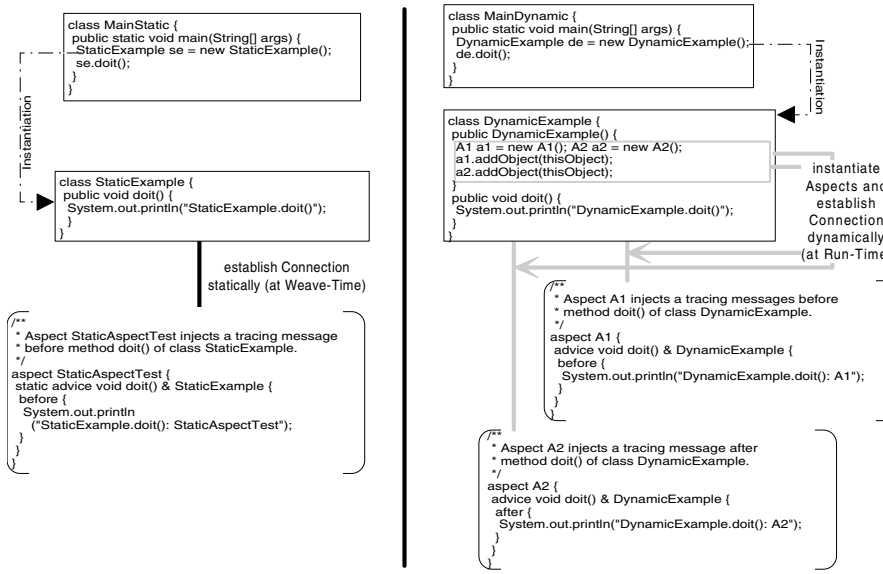


Fig. 2. Example with Statically or Dynamically Connected Aspects.

[13]. Thus, although statically connected aspects can be verified at run-time (in table 1 depicted with $+$), assertions execution at compile-time or weave-time is to prefer (this is expressed by $++$ in table 1). Static assertion violation checking is described in section 5.

Dynamically changing aspect configurations (as depicted on the right in figure 2) can hardly be checked at compile-time or weave-time. Which aspect instance is created and connected to which class instance depends on the dynamic control flow and user input. In this case, the dynamically executed assertions are usually unavoidable. The technique for dynamically asserting aspect configurations is described in the next section.

4 Asserting Dynamically Changing Aspect Configurations

In this chapter, we concentrate on all aspect composition validation which cannot be asserted statically as outlined above. Dynamically created and connected aspect instances have to be checked during program execution. These assertions can be added into the corresponding class. Since aspects can be used to separate assertion functionality from problem domain-related code, it is also possible and reasonable to extract the aspect configuration assertion code from the class and localise it into a separate assertion aspect. Thus, if a class wants to test whether the postcondition (i.e. the property "a valid configuration of active aspects is connected") holds, an assertion aspect has to be woven. As the other aspects,

this aspect can be instantiated and connected dynamically or statically. The implementation of such a static assertion aspect for `DynamicExample` of figure 2 is outlined in figure 3.

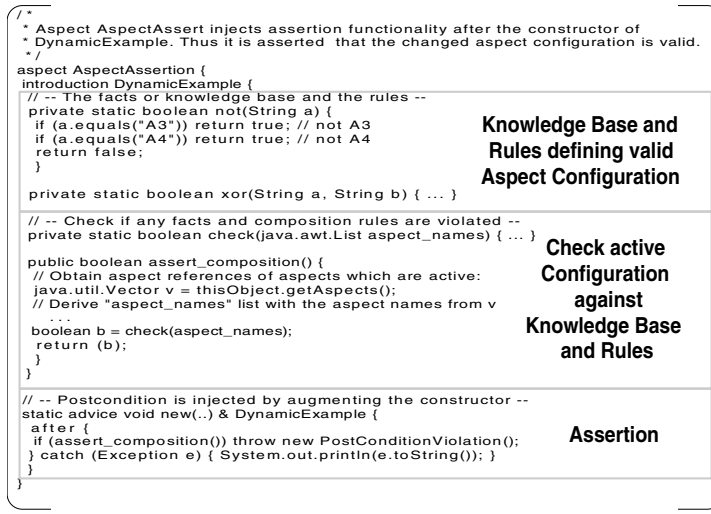


Fig. 3. Assertion Aspect for Dynamic Testing.

Note that with the introduction of these new assertion aspects, these aspects themselves may be ensured by higher-level assertion aspects. This situation can be compared to the abstraction hierarchy in object-orientation ("instance", "class" and "meta-class"). The crucial question is how many abstraction levels are reasonable. These considerations are beyond the scope of this paper [31].

According to figure 3, the knowledge base and rules part expresses the knowledge contained in the specification. It defines all valid aspect configurations. The actual configuration is determined with the AspectJ command `thisObject.getAspects()` during run-time which returns all active aspect instances connected to the class instance. Such a possibility to determine all woven and active aspect instances at run-time is crucial to their dynamic assertion. This determined set of active aspect instances is then checked against the knowledge base and rules. If this check proves that the postcondition is true, an invalid aspect configuration is detected and an exception is raised (cf. figure 3).

An interesting part is the one expressing the knowledge base and the rules of a valid aspect configuration. Some basic types of rules can be identified which allow to express the relevant dependencies between the aspects and between the aspects and a certain class instance (with respect to the specification). For our various example implementations the following rules proved to be sufficient:

- `not A1`: A not-clause expresses that the aspect with name `A1` is not allowed to be woven into the class instance.
- `requires A1`: Note that this term differs from *require* clauses sometimes used to express assertions (e.g. in the programming language Eiffel [19]). Here, it is meant that weaving aspect `A1` into the class instance is mandatory. This rule may also have multiple arguments (e.g. `requires A1 A2 A3`) which indicates that at least one of these aspects is mandatory (in the example either `A1` or `A2` or `A3` is mandatory).
- `xor A1 A2`: This expression indicates that either `A1` or `A2` may be woven into a class instance but never both of them.
- `and A1 A2`: An `and`-clause expresses that `A1` and `A2` have to be woven together into the same class instance.

In figure 3 the methods `not` and `xor` outline a possible implementation of two types of rules in AspectJ0.4beta7 / Java JDK 2. The `check(...)` method describes the application of these rules to a set of aspects.

Although these dependency rules are expressed in AspectJ0.4beta7 / Java syntax in the presented implementation a multi-paradigm approach [6,5] would be suitable here. With a logic programming language (e.g. Prolog) the implementation of the knowledge or dependency rules contained in the specification would lead to improved understandability. Generally speaking, a domain-specific language [7] based on the predicate calculus would improve the implementation. Alternatively, a domain-specific language based on finite state automata can be used to express these dependencies as described in [24].

An assertion aspect for a certain software system may be also generated from a file containing the knowledge base and rules [28].

5 Asserting Static Aspect Configurations

While the validation of the dynamic aspects results in performance penalties, these can be avoided for static aspects since the configuration can be assured before the execution (cf. table 1). It is a common technique to remove assertions at compile-time provided they can be checked in a static analysis [13].

The principle of such static validation procedure is as follows: Assuming we have a class implementation including static assertion clauses to assure a valid aspect configuration. Then, the compiler, weaver or any tool operating before execution time can examine this class implementation. The statically known information about what has to be asserted can be extracted from the code and checked before run-time. The dynamic tests during execution can be avoided since the already statically checked assertions can be eliminated.

Based on this observation, we built the "*Aspect Composition Validation*" tool (cf. figure 4) in order to automate the static verification. The tool realises a slightly different verification procedure compared to the principle described above. We chose not to inject assertion code into the classes. This eliminates the need to remove this code before execution. The developer decides by menu which classes the tool has to assert. The assertion itself (i.e. the property that is

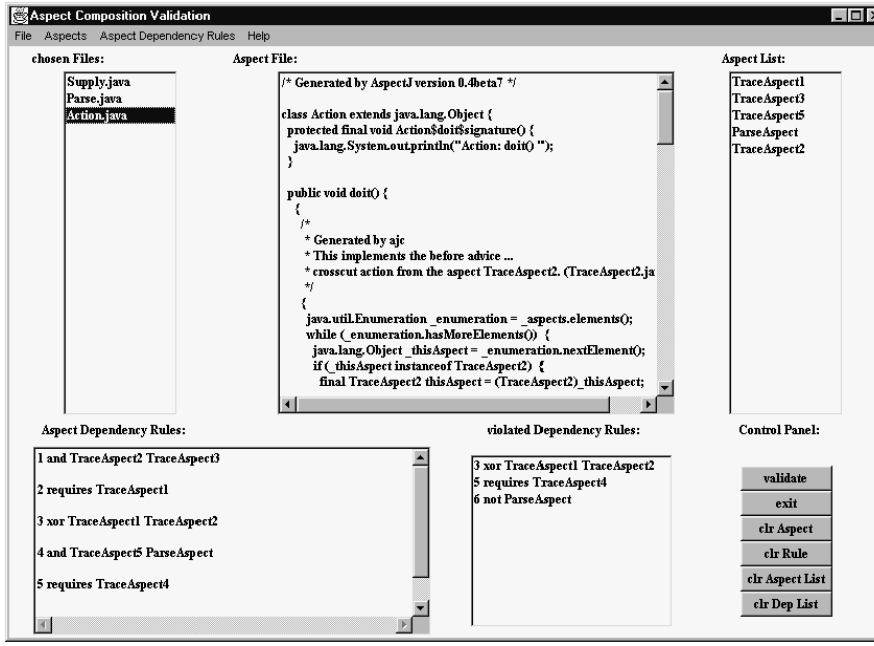


Fig. 4. *Aspect Composition Validation* Tool.

to be asserted) is obvious: the aspects connected to the class have to be checked according to aspect dependency rules derived from the specification.

The tool can be used to ensure the property of a valid static aspect configuration within a specific class. Such a class-wide validation corresponds to the concept of class invariants described in [21]. Moreover, the tool is also able to verify the aspect configuration of a set of classes. Such a set may be a package, a component or a subsystem. Therefore, this is an extension of B. Meyer's understanding of assertions (which is limited to pre-, postconditions and class invariants) to more than a single class, i.e. to more coarse-grained building blocks of the system. In the tool, the single file or the set of files containing the class(es) to be validated are listed in the *chosen Files* list with their aspects displayed in the field *Aspect List* as shown in figure 4. Additionally, the highlighted file in the file list is presented in the *Aspect File* text area.

The tool verifies the aspect configuration (consisting of all the aspects which are included in the listed files on the left) according to the aspect dependency rules (initiated by pressing the *validate* button). Violated aspect dependency rules are displayed in the list named *violated Dependency Rules*. The aspect dependency rules expressing the invariants of aspect combination are the same as those described in section 4 (`not`, `requires`, `and` and `xor`). The software developer can insert or change these aspect rules directly in the *Aspect Rules* text area.

Alternatively, they can be read from a user-defined file containing these rules written in the domain-specific language.

The *Aspect Composition Validation* tool extracts the static aspects by parsing the files containing the already woven source. AspectJ0.4beta7 marks all woven code sections with comments. Alternatively, the woven aspects could be derived from specific documentation files (with extension `.corr`) provided by AspectJ0.4beta7. This file documents which aspect is woven in which class or method. The woven aspect set could also be obtained from the original source files directly.

6 Conclusion

Although aspect-oriented programming can improve software due to a better separation of concerns, the software developer is faced with new challenges. One of them is the aspect composition validation which is not yet examined sufficiently in AOP and therefore is addressed in this paper.

Both the dependencies between the aspects woven with a class instance and the dependencies between these woven aspects and the class instance itself have to be identified in a specification. On this basis we demonstrate how to use assertions to ensure the correctness of these dependencies with respect to the specification. Faulty aspect configurations (e.g. if the set of woven aspects embraces aspects which are contradictory) can be detected using assertions similar to other bugs. Dynamically changing aspect configurations are checked at runtime. Due to the performance penalty in case of dynamic tests, we also presented a static analysis which is preferable in case of static aspect configurations. Although the feasibility of the principles and concepts in this paper is shown with AspectJ0.4beta7 and Java JDK 2 implementations the approach is independent of the concrete technology. For instance, using composition filters [2], [1] to achieve a better separation of concerns also leads to the problem that both the order of the filters and the filters of the different objects within one application have to fit semantically. Assertions can be used similarly in this case.

Our further research will concentrate on transferring these concepts to component or object composition validation with respect to already existing support in this domain [3]. There is a growing need for computer support in the field of composing systems of reusable parts which are stored in repositories. As with aspects, there might be multifarious dependencies between these parts. Another field is the graphical representation of the dependencies and of all violated dependency rules within a certain application. Besides this graphical feature, a tool environment can include further support for the software developer. The description of a valid configuration, i.e. of the knowledge base and the rules according to the specification, should be separated. Further investigation in the sense of domain engineering is necessary there. Moreover, a partial automatic generation of assertion aspects from such separated descriptions is possible and will be another research activity in our future work.

References

1. M. Aksit. Composition and Separation of Concerns in the Object-Oriented Model. *ACM Computing Surveys*, 28(4), December 1996.
2. M. Aksit and B. Tekinerdogan. Aspect-Oriented Programming Using Composition Filters. In *ECOOP 1998 Workshop Reader*, page 435, Springer-Verlag, 1998.
3. D. Batory and B.J. Geraci. Composition Validation and Subjectivity in GenVoca Generators. In *IEEE Transactions on Software Engineering*, pages 67–82, 1997.
4. G. Booch. *Object-Oriented Analysis and Design.* Benjamin/Cummings, Redwood City, CA, second edition, 1994.
5. J.O. Coplien. Multi-Paradigm Design. In A. Speck and E. Pulvermüller, editors, *Collection of Abstracts of the GCSE'99 YRW*, `http://www-pu.informatik.uni-tuebingen.de/users/speck/GCSE99_Young_Research/abstracts/Jim_Coplien_gcseYR99.html`, September 1999.
6. J.O. Coplien. *Multi-Paradigm Design for C++.* Addison-Wesley, 1999.
7. K. Czarnecki. *Generative Programming, Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models.* PhD thesis, Technical University of Ilmenau, Ilmenau, Germany, 1999.
8. K. Czarnecki and U.W. Eisenecker. Synthesizing Objects. In *Proceedings of ECOOP'99*, Lecture Notes in Computer Science LNCS 1628, pages 18–42. Springer-Verlag, June 1999.
9. E.W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.
10. U.W. Eisenecker. Generative Programming GP with C++. In H.-P. Mössenböck, editor, *Proceedings of Joint Modular Programming Languages Conference*, LNCS 1204. Springer-Verlag, 1997.
11. R.W. Floyd. Assigning Meanings to Programs. In J.T. Schwartz, editor, *Proc. Am. Math. Soc. Symp. in Applied Math.*, volume 19, pages 19–31, Providence, R.i., 1967. American Mathematical Society.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Abstractions and Reuse of Object-Oriented Software.* Addison-Wesley, Reading, MA, 1994.
13. J. Gough and H. Klaeren. Executable Assertions and Separate Compilation. In H.-P. Mössenböck, editor, *Proceedings Joint Modular Languages Conference*, LNCS 1204, pages 41–52. Springer-Verlag, 1997.
14. C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–583, October 1969.
15. P. Kenens, S. Michiels, F. Matthijs, B. Robben, E. Truyen, B. Vanhaute, W. Joosen, and P. Verbaeten. An AOP Case with Static and Dynamic Aspects. In *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP98*, 1998.
16. M. A. Kersten and G. C. Murphy. Atlas: A Case Study in Building a Web-based Learning Environment using Aspect-oriented Programming. OOPSLA, 1999.
17. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Lecture Notes in Computer Science LNCS 1241*, ECOOP. Springer-Verlag, June 1997.
18. C. V. Lopes and G. Kiczales. Recent Developments in AspectJ. In *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP98*, 1998.
19. B. Meyer. *Eiffel: The Language.* Prentice Hall, Englewood Cliffs, 1991.
20. B. Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, October 1992.
21. B. Meyer. *Object-Oriented Software Construction.* Prentice Hall PTR, Upper Saddle River, NJ, second edition, 1997.

22. Object Management Group. *The Common Object Request Broker: Architecture and Specification*, February 1998.
23. D. L. Parnas. On The Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.
24. L. Pazzi. Explicit Aspect Composition by Part-Whole Statecharts. In *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99*, 1999.
25. E. Pulvermüller, H. Klaeren, and A. Speck. Aspects in Distributed Environments. In *Proceedings of the International Symposium on Generative and Component-Based Software Engineering GCSE'99*, Erfurt, Germany, September 1999.
26. D.S. Rosenblum. A Practical Approach to Programming With Assertions. *IEEE Transaction on Software Engineering*, 21(1):19 –31, January 1995.
27. Y. Smaragdakis and D. Batory. Implementing Layered Designs with Mixin Layers. In *Lecture Notes in Computer Science LNCS 1445*, pages 550–570, 1998.
28. A. Speck, E. Pulvermüller, and M. Mezini. Reusability of Concerns. In C. V. Lopes, L. Bergmans, M. DHondt, and P. Tarr, editors, *Proceedings of the Aspects and Dimensions of Concerns Workshop, ECOOP2000*, Sophia Antipolis, France, June 2000.
29. C. Szyperski. *Component Software.* Addison-Wesley, ACM-Press, New York, 1997.
30. P. Wegner. The Object-Oriented Classification Paradigm. In P. Wegner and B. Shriver, editors, *Research Directions in Object-Oriented Programming*, pages 479–560. MIT Press, 1987.
31. J.F.H. Winkler and S. Kauer. Proving Assertions is also Useful. *SIGPLAN Notices*, 32(3):38–41, 1997.
32. XEROX Palo Alto Research Center, `http://aspectj.org`. *Homepage of AspectJ*, 2000.

Towards a Foundation of Component-Oriented Software Reference Models

Thorsten Teschke[1] and Jörg Ritter[2]

[1] Oldenburger Forschungs- und Entwicklungsinstitut für Informatik-Werkzeuge und -Systeme (OFFIS)
thorsten.teschke@offis.de
[2] OSC – OFFIS Systems and Consulting GmbH
joerg.ritter@o-s-c.de

Abstract. The increasing number of available software components and competing interoperability standards render the selection, composition, and configuration of software components increasingly complex. In order to support the domain expert in these processes comprehensive, comparable, and sufficiently abstract component descriptions are required. In this paper, we abstract from the specifics of the component models COM, EJB, and CCM, and propose a unifying component description language for integrated descriptions of structure and behaviour of software components and component-based software systems.

1 Introduction

The notion of component-based software development denotes an approach pursued in every mature engineering discipline: systems (here: software) shall be developed by reusing and combining well-tried methods and tested products (here: software components) [8]. Szyperski defines a software component (further on only referred to as component) as "[...] a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties" [25]. In this context a contract represents an agreement between a client and a provider concerning a set of services. It specifies how to use these services and imposes requirements on their implementation.

Within the field of component-based software development two kinds of standardization efforts can be distinguished: the technical standardization of component interfaces (e. g., by component models such as Component Object Model (COM), Enterprise JavaBeans (EJB), and CORBA Component Model (CCM)) on the one hand provides for improved exchangeability of components on a technical level, competing interoperability efforts (e. g., by OAGIS, OMG, RosettaNet) on the other define a multitude of domain interfaces which may be supported by software components. These standardization efforts intend to permit the combination of "best of breed" solutions, however, they overwhelm the domain expert with the resulting variability. Therefore, the selection, composition,

G. Butler and S. Jarzabek (Eds.): GCSE 2000, LNCS 2177, pp. 70-84, 2001.

and configuration of components and component-based application systems increasingly require suitable methods and tools supporting the domain expert. A comparable problem can be found in the field of standard software. Current standard software products excel by their high configurability and are therefore applicable in many different contexts [1]. In order to render this richness of variants more comprehensible and manageable for domain experts, standard ERP software products are described by *software reference models* (cf. [12]). Software reference models illustrate the business processes, organizational and data structures supported by a particular software product as well as its configuration alternatives on a level of abstraction which is suitable for domain experts.

We suggest to transfer the idea of software reference models to component-based software development in order to facilitate the sound selection, composition, and configuration of components and component-based software systems. Component-oriented software reference models are to support the domain expert in dealing with component software by abstracting from technical information and focusing on domain-related information instead. We envisage deriving component-oriented reference models in the shape of, e. g., business process models, message sequence charts, and controlled language documents from uniform, comprehensive, and sufficiently formal component descriptions. The specification languages employed by current component models are improper as a basis for such component-oriented software reference models. On the one hand, they align themselves too much with the specifics of the respective component model, and on the other, they are restricted to the signature level of interface descriptions. Important information on interaction protocols and terminological semantics of components may not be formulated using these languages. In this paper we therefore propose a language *CDL* for the integrated description of structure and behaviour of components and component-based software systems. CDL provides a comprehensive, unified view on the standard component models COM, EJB, and CCM by abstracting from technical specifics, yet ensuring the ability to express the most important concepts. In addition to a component's signature, CDL descriptions may also comprise its formal semantics in the shape of interaction protocols.

The structure of our paper is as follows. Section 2 outlines the most important concepts of the component models COM, EJB, and CCM, and presents the CDL component model which is intended to serve as a unified view on COM, EJB, and CCM components. In Sect. 3 the description language CDL is introduced using a simple example. Section 4 gives an outlook on future enhancements of CDL and presents related research. Finally, Sect. 5 summarizes our contribution and relates it to complementary research interests.

2 A Unified View on Standard Component Models

In this section we first delineate the currently most popular component models COM, EJB, and CCM. We concentrate on the relationships between contracts, implementations and runtime objects of their components. A more detailed com-

parison of COM+, COM's current successor, EJB, and CCM can be found in [24]. Thereafter, we present the CDL component model which is intended to serve as a unified, abstract view on component software. A brief comparison of the CDL component model with the component models COM, EJB, and CCM is presented in a slightly extended version of this paper [26].

2.1 Component Object Model (COM)

In Microsoft's *Component Object Model (COM)* the services provided by a component are specified by *interfaces* which represent sets of methods. *COM classes* are named implementations of at least one interface, its instances are referred to as *COM objects*. COM objects are created by means of *class factories*. A *COM component* packages one or more COM classes and constitutes the unit deployable in software development projects. COM does not support inheritance between components [6]. Since COM classes do not explicitly publish the interfaces they support, it is in general necessary to validate a COM class's suitability for a given application context by successively checking the availability of required interfaces (*interface negotiation*). In order to avoid this often large number of checks, COM provides the concept of *categories*. Categories denote sets of interfaces and make the notion of contracts explicit [25]. Figure 1 shows the basic concepts of COM and the relationships between. The figure insinuates an object-oriented implementation of COM classes, however, a non-object-oriented implementation is also conceivable.

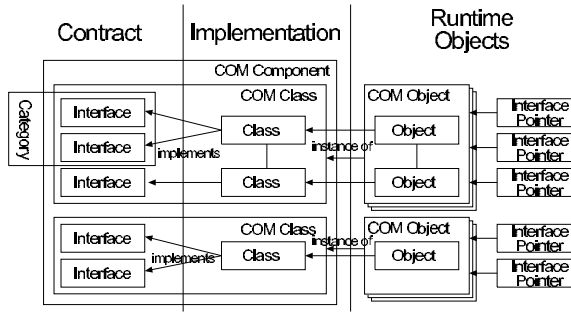


Fig. 1. Component Object Model (COM)

An interesting feature of COM is the fact that a COM class may support different versions of an interface simultaneously. COM prohibits changes to interfaces once they are specified. Thereby COM avoids the *syntactic and semantic fragile base class problem* which pertains to the question whether a base class (a component) can evolve without breaking independently developed subclasses (dependent components) [25]. In order to be able to provide new or advanced functionality nevertheless, new versions of a COM class may offer additional interfaces comprising new or updated versions of existing functionality.

2.2 Enterprise JavaBeans (EJB)

Sun Microsystems's *Enterprise JavaBeans (EJB)* is oriented towards the development of distributed server-side applications [14]. A *server* provides an environment for the execution of applications. Within a server, *containers* manage the components of the EJB component model: the *enterprise beans*. Containers provide interfaces for the platform independent development of both clients and enterprise beans. These interfaces are specified by means of two contracts. The *client contract* constitutes the contract between client and container. It governs a client's access to the enterprise beans deployed within the container. Every enterprise bean defines a *home interface* which offers methods for creating, finding, and destroying instances of the respective bean, and a *remote interface* providing application specific methods of the bean. The *component contract* specifies the interface between a container and the enterprise beans it contains. It requires a container to implement both an enterprise bean's home and remote interface. The implementation of a remote interface thereby has to delegate method invocations issued by a client to the *ejb-class* coming with the enterprise bean and providing the implementation of the bean's functionality. Instances of the implementations of home and remote interface are referred to as *EJBHome* and *EJBObject*, respectively [14]. EJB does not support inheritance between enterprise beans. Figure 2 depicts the views "contract", "implementation" and "runtime objects" on EJB.

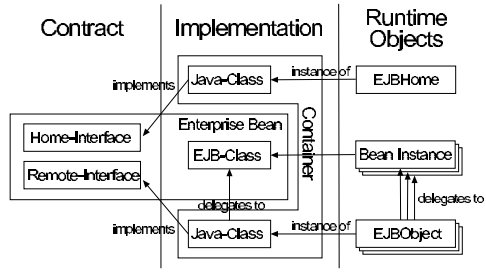


Fig. 2. Enterprise JavaBeans (EJB)

2.3 CORBA Component Model (CCM)

The specification of the Object Management Group's (OMG) *CORBA Component Model (CCM)* is not yet finished. The following survey on the CORBA Component Model grounds on a joint revised submission to the OMG made by a number of renowned software development companies [20].

The CORBA Component Model combines concepts from COM and EJB and thus represents in a way a "best of both worlds" component model. A *component*

type denotes a specification of a component's features and may encapsulate several implementations (e. g., for different platforms). Similar to EJB, instances of component types, the *CORBA components (CORCs)*, are managed by *homes*. A CORBA component has a single distinguished reference whose interface conforms to the component's *equivalent interface*. This interface publishes the component's features (referred to as *ports*). The kinds of ports supported by CCM are facets, receptacles, event sources and sinks, and attributes. Similar to non-standard interfaces in COM, *facets* represent the functional application interfaces of a CORBA component. A CORBA component may explicitly support additional interfaces next to its facets. *Receptacles* represent an abstraction for the creation and management of typed connections between a CORBA component and an object (e. g., a reference to a facet or the home of another CORBA component). *Event sources* push events to consumer interfaces provided by *event sinks*. *Attributes* serve the configuration of components. Figure 3 illustrates the most important concepts of the CORBA Component Model in the acquainted form. For the sake of clarity the possibility of single inheritance between component types is not shown in this figure.

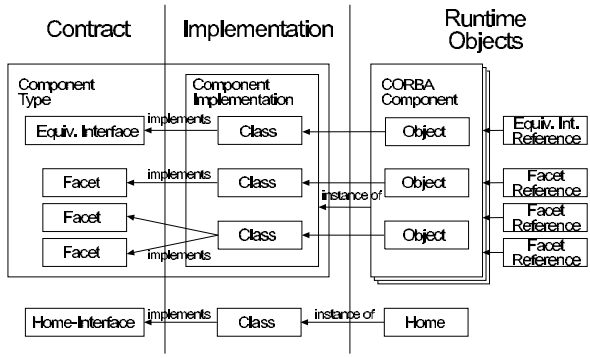


Fig. 3. CORBA Component Model (CCM)

2.4 The CDL Component Model

The *CDL component model* is intended to provide a unified abstraction from the component models introduced in the preceding paragraphs. According to Szyperski components in general may provide two different kinds of interfaces. *Direct interfaces* correspond to procedural interfaces known from classical software libraries. *Indirect interfaces* are interfaces of objects provided by components [25]. Analogously to these types of interfaces, the CDL component model comprises *components* which may be called using a direct interface, and *classes* which are packaged by components and whose instances – the *business objects* – may be called by means of an indirect interface. The CDL component

model distinguishes between *atomic*, i. e. non-composite components, and *complex* components, i. e. composites of atomic and/or complex components. Atomic components can be further subdivided into *closed* components and *component frameworks*. While closed components are not extensible, component frameworks represent the basis for the construction of complex components. They provide extension points ("hot spots") for their connection with other components.

An important issue in a component-based software system is the *substitutability* of its components. If, e. g., a company wanted to substitute a more modern production planning component for the one contained in its business application system, the new component is required to show the structure and behaviour expected by the application system. Within the CDL component model, the notion of substitutability is addressed by *contracts*. Component frameworks use contracts to specify types of components which may be "plugged into" their extension points. Similar to components comprising classes, contracts may contain *objecttypes*. An objecttype specifies a type of business object. A contract may be implemented by arbitrary components, and conversely a component may support an arbitrary number of contracts. Each of a component's classes may realize one or more objecttypes of a contract. Figure 4 illustrates the concepts of the CDL component model.

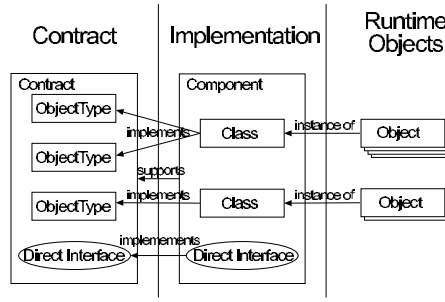


Fig. 4. CDL Component Model

3 Comprehensive Description of Software Components

In this section we introduce *CDL (Component Description Language)*, a revised version of the SCDL (Simple Component Description Language) proposed in [23]. CDL grounds on the CDL component model outlined in the preceding section.

3.1 The Component Description Language

The primary goal of CDL is to provide a unified foundation for the description of components of different component models. CDL descriptions are intended to

serve as a basis for the derivation of domain-oriented models like, e. g., event-driven process chains (EPCs), message sequence charts or controlled language documents, which facilitate the assessment of software components especially for domain experts. The orientation of CDL towards domain aspects is underpinned by the idea to explicitly support underspecifications of components and contracts. Such underspecifications might result from incomplete descriptions of method and procedure parameters as well as non-deterministic choices. An example of underspecifications are the EPCs of the R/3 reference model [12]. These render the behaviour of the R/3 system only fragmentarily. However, they serve the purpose of supplying users and consultants with a sufficient overview over the functionality of the R/3 system.

The description languages of the component models COM, EJB, and CCM show various deficiencies and are therefore not suitable as a foundation for domain-oriented descriptions of components and component-based software systems. COM IDL [2] is constrained to the description of collections of methods. EJB's deployment descriptors [14] or CCM's component assembly descriptors [20] permit the formulation of structural information like, e. g., on the architecture of component-based application systems. However, both EJB and CCM descriptions do not explicitly specify the methods comprised by a component or business object interface. This information can only be retrieved indirectly by means of introspection (in this case the component has to be present) or access to an interface repository, respectively. From our point of view the gravest deficiency, however, is the fact that non of these description languages regards interaction protocols, i. e. they are restricted to the signature level. The specification of a component's behaviour in the shape of the interactions the component may engage in is indispensable for semantically rich, domain-oriented descriptions. For this reason CDL comprises constructs for the description of both structural and dynamic aspects of components and component-based software systems. According to Harel and Gery a minimal set of languages for the description of object-oriented systems consists of object-model diagrams and statecharts [9]. In addition to statecharts various other approaches to behavioural descriptions do exist. Pre- and post-conditions constitute a solid formal foundation, however, they are not operational and do therefore not represent an adequate basis for domain-oriented models [27]. Process terms based on process calculi like CCS [15], CSP [11] and the π-calculus [17] permit an operational representation of processes and interactions. Another approach to behaviour descriptions grounds on the insight that different process terms may denote the same set of communication sequences (traces). Trace specifications therefore abstract from process terms and describe behaviour declaratively as sets of traces, thereby losing the operational representation [19]. CDL grounds on process terms on the basis of the polyadic π-calculus and concepts of statecharts for the specification of component behaviour as well as object-model diagrams for the description of structural properties. The polyadic π-calculus [16] represents an extended version of the original π-calculus. It permits tuples and compound types to be sent along communication channels.

CDL distinguishes between component and contract descriptions. Apart from underspecification, a component description is intended to define an upper bound to the set of a component's possible interaction sequences. This semantics is interesting for safety reasons: a component does not engage in more interactions than asserted by its description. Accordingly, a contract description specifies a lower bound to the set of expected interaction sequences (liveness). This perspective guarantees that a component which supports a particular contract provides the services required. Ebert and Engels coined the terms *observable behaviour* and *invocable behaviour* for these semantics of behaviour descriptions [5]. Analogous formulations can also be found in the field of process terms: a process "may engage" or "must engage" in a set of traces, respectively [19].

Describing Atomic Components. In this section we introduce those aspects of the component description language CDL which are relevant to the description of atomic, closed components and corresponding contracts. Figure 5 shows a description of an atomic, closed component `AccountComponent` which may be employed for the management of accounts.

Component descriptions begin with the keyword `component`. They may contain descriptions of arbitrary classes which are marked by the keyword `class` and specify implemented types of the component's business objects. The component `AccountComponent` in Fig. 5 comprises only one class named `Account`. A component's direct interface is represented by procedures. Within our example, the direct interface of `AccountComponent` consists of a procedure `openAccount()` for opening a new account. The indirect interface of a component is represented by its classes and their methods. The class `Account` defines methods `deposit()` and `withdraw()` for depositing money in an account and withdrawing it, and `close()` for closing an account. Parameters of procedures and methods are directed (`in`, `out`, `inout`) and typed using the classes, contracts, and objecttypes known within a component's scope as well as base types like `int`, `float` and `string`.

In addition to these structural description elements, a component description may also encompass descriptions of a component's and its classes's behaviour. For this purpose we employ the idea underlying communication within the π-calculus: a procedure or method identifier followed by a "!" indicates the sending of a call to the procedure or method. The readiness for the receiption of such a call is specified by a "?" following the identifier. A procedure or method is executed, if a send action can be matched with a corresponding receive action. The description of a component's or class's behaviour using CDL is based on the idea to separate the specification of the active behaviour, i. e. the procedure and method calls issued, from the specification of the passive behaviour, i. e. the *readiness* to receive such calls (cf. [19]). The active behaviour of a component or class is specified in procedure and method bodies, while the passive behaviour is formulated in specific behaviour descriptions. The latter consist of states and transitions, whereby each behaviour description comprises two standard states: initially, an instance of a component or a class is in the state `created`. This state

```
component AccountComponent {
   class Account {
      deposit(in sum: float) {
         return;          /* increase of balance not described */
      };
      withdraw(in sum: float) {
         return;          /* decrease of balance not described */
      };
      close() {
         return;          /* closing of account not described */
      };

      created()              =   initialized();
      initialized()          =   deposit?(sum).balance(sum)
                               + close?().closed();
      balance(value: float) =   deposit?(sum).balance(value+sum)
                               + on withdraw?(sum)
                                   if (sum <= value)
                                     do balance(value-sum)
                               + on close?()
                                   if (value == 0)
                                     do closed();
      closed()               =   ();
   };

   openAccount(out account: AccountComponent.Account) {
      new(AccountComponent.Account account);
      return(account);
   };

   created()            = initialized();
   initialized()        = openAccount?(account).initialized();
};
```

Fig. 5. Component AccountComponent

can be associated with an initialization script, i.e. a sequence of procedure or method calls. After this initialization the component or class instance progresses to the `initialized` state which represents its effective start state. Behaviour descriptions associated with this or a subsequent state may only describe the readiness to receive procedure or method calls, i.e. its passive behaviour.

Both the component `AccountComponent` and the class `Account` do not require any initialization. For this reason they directly progress from state `created` to the successor state `initialized`. `AccountComponent` is then ready to accept calls to the procedure `openAccount()`. Whenever this procedure is called, a new instance of class `Account` is created and returned using the `out` parameter `account`. Subsequently, `AccountComponent` returns to the state `initialized`. The new instance of class `Account` is initially prepared to accept deposits. If

a deposit of amount `sum` is made by issuing a call to `deposit()`, the instance progresses to the state `balance(sum)`, i. e. states may have parameters of the base types mentioned earlier. As an alternative to making deposits the `Account` instance may be closed again. This choice is *global non-deterministic*, i. e. it depends on the interaction behaviour of the clients, and is specified by the operator `+`. After calling the method `close()` the instance progresses to the state `closed` which only comprises the empty process term `()` and therefore rules out any further interaction with the `Account` instance. Associated with the state `balance` we describe the conditional readiness of an `Account` instance to accept calls to its method `withdraw()`. The amount of money to be withdrawn is limited by the current balance. The semantics of the construct `on <call> if <condition> do <statechange>` is derived from the idea of *call events* employed within UML *state machines* [21]: in case of a call to the procedure or method specified by `<call>` the condition `<condition>` is checked. If this condition holds, the call is executed, otherwise it is rejected and an *exception* is generated. Concerning the communication of exceptions, we employ the convention that every call returns a truth value (*true*, if the call was successful, *false*, otherwise). The calling component or class instance may define reactions to these truth values by means of an `if-then` statement.

Descriptions of contracts and components do hardly differ syntactically: merely the keywords `component` and `class` for the description of components need to be replaced by the keywords `contract` and `objecttype` for the description of contracts.

Describing Complex Components. In order to present the constructs provided by CDL for the description of complex components we enhance our example by introducing a component framework which will be extended by a component. Figure 6 shows the description of a component framework `BankComponent` which realizes a fragment of a bank's functionality. The `BankComponent` comprises a class `Bank`, whose instances represent individual banks. The direct interface of `BankComponent` permits the creation of an arbitrary number of `Banks`. The interface of the class `Bank` defines methods `openAccount()` for opening an account and `transfer()` for transferring money between two accounts. The description of the method `transfer()` refers to the truth values returned by each method call (cf. Sect. 3.1): in case the withdrawal required for the transfer fails, no subsequent deposit will be made.

Component frameworks differ from closed components in being extensible by other components. In CDL such extension points of component framework descriptions are specified using the keyword `slot`. Slots are typed with contracts which define requirements to compatible components. The description of `BankComponent`, e. g., comprises a slot named `accComp` into which components complying with the contract `AccountContract` depicted in Fig. 7 may be "plugged". The contract description `AccountContract` specifies a set of components which may be employed by the `BankComponent` for account management. It differs from

```
component BankComponent {
   slot AccountContract accComp;

   class Bank {
      openAccount(out account: AccountContract.CAccount) {
         accComp.openAccount!(account);
         return account;
      };
      transfer(in source: AccountContract.CAccount,
               in dest: AccountContract.CAccount, in sum: float) {
         if source.withdraw!(sum) then
           dest.deposit!(sum);
         return;
      };

      created()     = initialized();
      initialized() = (  openAccount?(account)
                       + transfer?(source, dest, sum) ).initialized();
   };

   createBank(out bank: BankComponent.Bank) {
      new (BankComponent.Bank bank);
      return(bank);
   };

   created()     = initialized();
   initialized() = createBank?(bank).initialized()
};
```

Fig. 6. Component Framework `BankComponent`

the component description `AccountComponent` shown in Fig. 5 in particular in not requiring the opportunity to close accounts.

A complex component represents a configuration of a component framework where every slot is filled by a component complying with the respective contract. It is defined by specifying a component framework and a set of components to be "plugged into" the framework's slots. If one regards component frameworks as templates, a complex component represents an instance of such a template. Figure 8 depicts the description of a complex component `XYBank` which is composed from the components `BankComponent` and `AccountComponent`. The composition of `BankComponent` and `AccountComponent` represents a valid configuration, since `AccountComponent` complies with `AccountContract`. The description of `AccountComponent` encompasses every trace specified by the contract description `AccountContract` and defines the same set of possible *failures*, i. e. rejected traces (cf. [18]). The fact that an `AccountComponent` can be closed is irrelevant from the `BankComponent`'s perspective. If the compliance of `AccountComponent` with `AccountContract` had been known at the component description's development time, this could have been explicitly expressed in CDL

```
contract AccountContract {
   objecttype CAccount {
      deposit(in sum: float) {
         return;          /* increase of balance not described */
      };
      withdraw(in sum: float) {
         return;          /* decrease of balance not described */
      };

      created()              =   initialized();
      initialized()          =   deposit?(sum).balance(sum);
      balance(value: float) =   deposit?(sum).balance(value+sum)
                              + on withdraw?(sum)
                                  if (sum <= value)
                                    do balance(value-sum);
   };

   openAccount(out account: AccountContract.CAccount) {
      new(AccountContract.CAccount account);
      return(account);
   };

   created()              = initialized();
   initialized()          = openAccount?(account).initialized();
};
```

Fig. 7. Contract AccountContract

```
complex XYBank = BankComponent(AccountComponent);
```

Fig. 8. Complex Component XYBank

by a supports statement (cf. Fig. 9). Such an explicit specification, however, is not a prerequisite for "plugging" a component into an extension point of a component framework.

```
component AccountComponent supports AccountContract {
   class Account {
      ...
};
```

Fig. 9. Explicit Support of Contract

To summarize, CDL permits the integrated description of structure and behaviour of components and component-based software systems. Moreover, CDL explicitly supports the composition of components by its ability to describe component frameworks and complex components.

4 Future and Related Work

Clearly, the ability to map the essential concepts of the standard component models COM, EJB, and CCM to CDL does not suffice for a comprehensive component description language. In its current shape CDL only supports the specification of functional properties of components and component-based software systems. In this regard we intend to extend the description of exceptions thrown by failed method calls by typed exceptions. Moreover, the restriction of supported events to *call events* will be revised. From our rather business process-oriented perspective, however, support for additional types of events is of subordinate importance. Inheritance between components will not be supported in CDL since we do not expect inheritance to play a major role in component-based software development. In the context of independent actors developing and composing components we consider the concepts of structural and behavioural subtyping to be more important. Regarding non-functional properties of components we intend to enhance CDL by adding constructs for the description of properties such as time and space requirements, and service levels. In its current version CDL is restricted to information on the level of signatures and interaction protocols. In order to remedy this deficiency, we are eventually interested in questions regarding the semantics of the terminology employed within CDL component descriptions (*semantic level of interoperability*).

A number of different approaches to the description of components and component-based software systems do exist, some of which have influenced the design of CDL. Wing and Zaremski employ pre- and post-conditions for the specification of components. They define relaxed notions of generalization, specialization, and substitutability of components [27]. The architecture description language *Darwin* is based on the π-calculus. In Darwin an architecture is made up of components which are connected through the services they require and provide [13]. Canal et al. extend CORBA interfaces by adding behavioural descriptions based on the π-calculus. They aim at questions related to substitutability and compatibility of CORBA objects [3]. The information system specification language *TROLL* combines both declarative and operational concepts for the specification of object systems. The behaviour of objects and object systems is described using temporal logic and some dialect of CSP [10]. *BOCA CDL* is the specification language of the *Business Object Component Architecture*. BOCA CDL is a declarative language for the specification of enterprise object interfaces, relationships between enterprise objects, and behaviour of communities of such objects [4]. *CSP-OZ* is an integrated formal method which combines the state-oriented method Object-Z with the process algebra CSP. A CSP-OZ specification describes a system as a collection of interacting objects, each of which is described regarding structural and dynamic aspects [7].

5 Concluding Remarks

Starting from an outline of the component models Component Object Model (COM), Enterprise JavaBeans (EJB), and CORBA Component Model (CCM)

we have proposed a unified abstraction from component software in the shape of the CDL component model. On the basis of the CDL component model we have proposed CDL, a language for the integrated description of structure and behaviour of components. CDL is a revised version of SCDL, which has been formally defined by Ritter by means of a translation to PICT [22], an asynchronous variant of the π-calculus [23]. We have introduced CDL by means of a simple example indicating how to describe components and component-based software systems using CDL.

Our work concerning CDL is strongly linked to the research project KOSOBAR (Component-Based Software Development Based on Reference Models) which is currently conducted at OFFIS. Within the scope of this project we are concerned with the development of tools and methods for the distributed development of component-based software. Thereby we assume that corresponding software development processes ground on the interaction between actors like component developers, application architects, consultants, and users. Current work within this project deals with mapping the CDL component model to the *Unified Modeling Language (UML)* [21] and developing a configuration description language for component-based software systems. A proposal for such a language on the basis of description logics can be found in [23]. Future work besides the enhancements of CDL outlined in Sect. 4 will be concerned with the derivation of domain-oriented models like, e. g., EPCs, message sequence charts, interaction diagrams or controlled language documents from CDL descriptions.

References

1. Appelrath, H.-J., and Ritter, J. *SAP R/3 Implementation. Methods and Tools.* SAP Excellence. Springer-Verlag, Berlin, Heidelberg, 2000.
2. Brockschmidt, K. *Inside OLE*, 2. ed. Microsoft Press, 1995.
3. Canal, C., Fuentes, L., Troya, J. M., and Vallecillo, A. Extending CORBA interfaces with π-calculus for protocol compatibility. In *Proceedings of TOOLS Europe 2000* (Dpto. de Lenguajes y Ciencias de la Computacin, Universidad de Mlaga 2000), IEEE Press.
4. Data Access Technologies. *Business Object Component Architecture Overview*, May 1998. `http://www.d-a-t.com/Documents/CdlConcepts.PDF`.
5. Ebert, J., and Engels, G. Observable or invocable behaviour - you have to choose! Tech. Rep. tr94-38, University of Koblenz, University of Leiden, 1994.
6. Eddon, G., and Eddon, H. *Inside Distributed COM.* Microsoft Press, 1998.
7. Fischer, C. CSP-OZ: A combination of Object-Z and CSP. In *Proceedings of Formal Methods for Open Object-Based Distributed Systems (FMOODS '97)* (1997), H. Bowmann and J. Derrick, Eds., vol. 2, Chapman & Hall, pp. 423–438.
8. Griffel, F. *Componentware.* dpunkt-Verlag, Heidelberg, 1998.
9. Harel, D., and Gery, E. Executable object modeling with statecharts. *IEEE Computer 30*, 7 (1997).
10. Hartel, P., Hartmann, T., Kusch, J., and Saake, G. Specifying information systems dynamics in TROLL. In *Proceedings of Workshop Formal Methods for Information Systems Dynamics* (University of Twente, 1994), pp. 53–64.
11. Hoare, C. A. R. *Communicating Sequential Processes.* Prentice Hall, New York, 1985.

12. KELLER, G., AND TEUFEL, T. *SAP R/3 Process Oriented Implementation: Iterative Process Prototyping.* Addison Wesley Publishing Company, 1998.
13. MAGEE, J., DULAY, N., EISENBACH, S., AND KRAMER, J. Specifying distributed software architectures. In *Proceedings of 5th European Software Engineering Conference (ESEC '95)* (Barcelona, 1995).
14. MATENA, V., AND HAPNER, M. *Enterprise JavaBeans Specification.* Sun Microsystems, 1999.
15. MILNER, R. *Communication and Concurrency.* Prentice Hall, New York, 1989.
16. MILNER, R. The polyadic π-calculus: A tutorial. In *Logic and Algebra of Specification*, F. L. Hamer, W. Brauer, and H. Schwichtenberg, Eds. Springer-Verlag, 1993.
17. MILNER, R., PARROW, J., AND WALKER, D. A calculus of mobile processes, part I and II. *Journal of Information and Computation 100*, 1 (1992), 1–77.
18. NIERSTRASZ, O. Regular types for active objects. In *Object-Oriented Software Composition*, O. Nierstrasz and D. Tsichritzis, Eds. Prentice Hall, 1995, pp. 99–121.
19. OLDEROG, E.-R. *Nets, Terms and Formulas.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, 1991.
20. OMG. *CORBA Components: Joint Revised Submission, OMG TC document orbos/99-02-05.* Object Management Group, 1999.
21. OMG. *Unified Modeling Language Specification, Version 1.3.* Object Management Group, 1999.
22. PIERCE, B. C., AND TURNER, D. N. Pict: A programming language based on the π-calculus. Tech. Rep. CSCI 476, Computer Science Department, Indiana University, 1997.
23. RITTER, J. *Prozessorientierte Konfiguration komponentenbasierter Anwendungssysteme.* Dissertation, Carl von Ossietzky Universität Oldenburg, 2000.
24. STAL, M. Reich der Mitte – Die Komponententechnologien COM+, EJB und CORBA Components. *OBJEKTspektrum 3* (2000), 26–33.
25. SZYPERSKI, C. *Component Software - Beyond Object-Oriented Programming.* Addison-Wesley, 1998.
26. TESCHKE, T., AND RITTER, J. Towards a foundation of component-oriented software reference models. In *Proceedings of Net.ObjectDays2000* (2000), pp. 479–493.
27. ZAREMSKI, A. M., AND WING, J. M. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology 6*, 4 (1997), 333–369.

Grammars as Contracts

Merijn de Jonge and Joost Visser

CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
{Merijn.de.Jonge,Joost.Visser @cwi.nl

Abstract. Component-based development of language tools stands in need of meta-tool support. This support can be offered by generation of code – libraries or full-fledged components – from syntax definitions. We develop a comprehensive architecture for such syntax-driven meta-tooling in which grammars serve as contracts between components. This architecture addresses exchange and processing both of full parse trees and of abstract syntax trees, and it caters for the integration of generated parse and pretty-print components with tree processing components.
We discuss an instantiation of the architecture for the syntax definition formalism SDF, integrating both existing and newly developed meta-tools that support SDF. The ATerm format is adopted as exchange format. This instantiation gives special attention to adaptability, scalability, reusability, and maintainability issues surrounding language tool development.

1 Introduction

A need exists for meta-tools supporting component-based construction of language tools. Language-oriented software engineering areas such as development of domain-specific languages (DSLs), language engineering, and automatic software renovation (ASR) pose challenges to tool-developers with respect to adaptability, scalability, and maintainability of the tool development process. These challenges call for methods and tools that facilitate reuse. One such method is component-based construction of language tools, and this method needs to be supported by appropriate meta-tooling to be viable.

Component-based construction of language tools can be supported by meta-tools that generate code – subroutine libraries or full-fledged components – from syntax definitions. Figure 1 shows a global architecture for such meta-tooling. The bold arrows depict meta-tools, and the grey ellipses depict generated code. From a syntax definition, a parse component and a pretty-print component are generated that take input terms into trees and vice versa. From the same syntax definition a library is generated for each supported programming language, which is imported by components that operate on these trees. One such component is depicted at the bottom of the picture (more would clutter the picture). Several of these components, possibly developed in different programming languages can interoperate seamlessly, since the imported exchange code is generated from the same syntax definition.

In this paper we will refine the global architecture of Figure 1 into a comprehensive architecture for syntax-driven meta-tooling. This architecture embodies the idea

G. Butler and S. Jarzabek (Eds.): GCSE 2000, LNCS 2177, pp. 85-99, 2001.

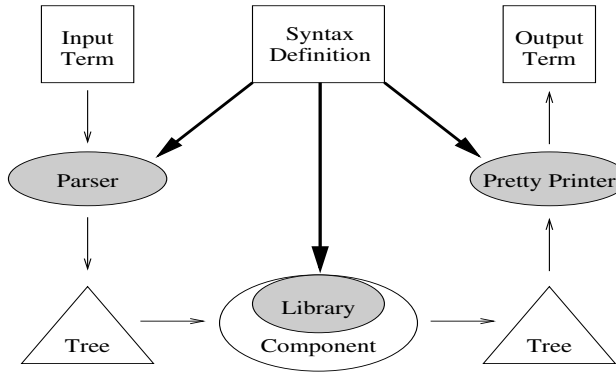


Fig. 1. Architecture for meta-tool support for component based language tool development. Bold arrows are meta-tools. Grey ellipses are generated code.

that grammars can serve as contracts governing all exchange of syntax trees between components and that representation and exchange of these trees should be supported by a common exchange format. An instantiation of this architecture is available as part of the Transformation Tools package XT.

The paper is structured as follows. In Sections 2, 3, and 4 we will develop several perspectives on the architecture. For each perspective we will make an inventory of meta-languages and meta-tools and formulate requirements on these languages and tools. We will discuss how we instantiated this architecture: by adopting or developing specific languages and tools meeting these requirements. In Section 5 we will combine the various perspectives thus developed into a comprehensive architecture. Applications of the presented meta-tooling will be described in Section 6. Sections 7, and 8 contain a discussion of related work and a summary of our contributions.

2 Concrete syntax definition and meta-tooling

One aspect of meta-tooling for component based language tool development concerns the generation of code from *concrete* syntax definitions (grammars). Figure 2 shows the basic architecture of such tooling. Given a concrete syntax definition, parse and pretty-print components are generated by a parser generator and a pretty-printer generator, respectively. Furthermore, library code is generated, which is imported by tool components (Figure 2 shows no more than a single component to prevent clutter). These components use the generated library code to represent parse trees (i.e. *concrete* syntax trees), read, process, and write them. Thus, the grammar serves as an interface description for these components, since it describes the form of the trees that are exchanged.

A key feature of this approach is that meta-tools such as pretty-printer and parser generators are assumed to operate on the same input grammar. The reason for this is that having multiple grammars for these purposes introduces enormous maintenance costs in application areas with large, rapidly changing grammars. A grammar serving

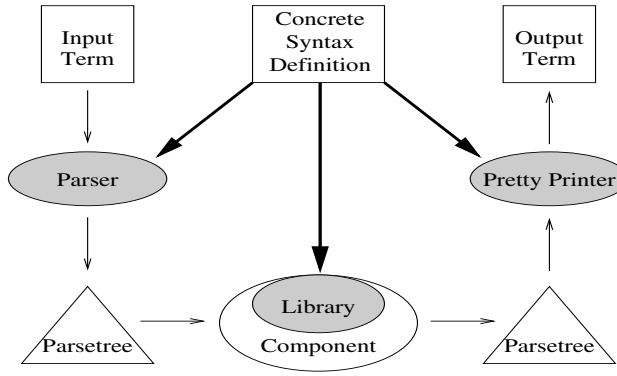


Fig. 2. Architecture for *concrete syntax* meta-tools. The concrete syntax definition serves as contract between components. Components that import generated library code interoperate with each other and with generated parsers and pretty-printers by exchanging parse trees adhering to the contractual grammar.

as interface definition enables smooth interoperation between parse components, pretty-print components and tree processing components. In fact, we want grammars to serve as contracts governing all exchange of trees between components, and having several contracts specifying the same agreement is a recipe for disagreement.

Note that our architecture deviates from existing meta-tools in the respect that we assume full parse trees can be produced by parsers and consumed by pretty-printers, not just abstract syntax trees (ASTs). These parse trees contain not only semantically relevant information, as do ASTs, but they additionally contain nodes representing literals, layout, and comments. The reason for allowing such concrete syntax information in trees is that many applications, e.g. software renovation, require preservation of layout and comments during tree transformation.

2.1 Concrete syntax definition

In order to satisfy our adaptability, scalability and maintainability demands, the concrete syntax definition formalism must satisfy a number of criteria. The syntax definition formalism must have powerful support for modularity and reuse. It must be possible to extend languages without changing the grammar for the base language. This is essential, because each change to a grammar on which tooling is based potentially leads to a modification avalanche. Also, the grammar language must be purely declarative. If not, its reusability for different purposes is compromised.

In our instantiation of the meta-tool architecture, the central role of concrete syntax definition language is fulfilled by the Syntax Definition Formalism SDF [11]. Figure 3 shows an example of an SDF grammar. This example definition contains lexical and context-free syntax definitions distributed over a number of modules. Note that the orientation of productions is flipped with respect to BNF notation.

```
definition
module Exp
exports
  context-free syntax
    Identifier                          Exp {cons(var)
    Identifier "(" {Exp "," * ")"       Exp {cons(fcall)
    "(" Exp ")"                         Exp {bracket
module Let
exports
  context-free syntax
    let Defs in Exp      Exp {cons(let)
    Exp where Defs       Exp {cons(where)
module Def
exports
  aliases
    {( Identifier "=" Exp ) "," +      Defs
module Main
imports Exp Let Def
exports
  sorts Exp
  lexical syntax
    [ ␣ t n]      LAYOUT
  context-free restrictions
    LAYOUT?       -/- [ ␣ t n]
```

Fig. 3. An example SDF grammar.

SDF offers powerful modularization features. Notably, it allows modules to be mutually dependent, and it allows alternatives of the same non-terminal to be spread across multiple modules. For instance, the syntax of a kernel language and the syntaxes of its extensions can be defined in separate modules. Also, mutually dependent non-terminals can be defined in separate modules. Renamings and parameterized modules further facilitate syntax reuse.

SDF is a highly expressive syntax definition formalism. Apart from symbol iteration constructors, with or without separators, it provides notation for optional symbols, sequences of symbols, optional symbols, and more. These notations for building compound symbols can be arbitrarily nested. SDF is not limited to a subclass of context-free grammars, such as LR or LL grammars. Since the full class of context-free syntaxes, as opposed to any of its proper subclasses, is closed under composition (combining two context-free grammars will always produce a grammar that is context-free as well), this absence of restrictions is essential to obtain true modular syntax definition, and "as-is" syntax reuse.

SDF offers disambiguation constructs, such as associativity annotations and relative production priorities, that are decoupled from constructs for syntax definition itself. As a result, disambiguation and syntax definition are not tangled in grammars. This is beneficial for syntax definition reuse. Also, SDF grammars are purely declarative, ensuring their reusability for other purposes besides parsing (e.g. code generation, pretty-printing).

SDF offers the ability to control the shape of parse trees. The alias construct (see module *Def* in Figure 3) allows auxiliary names for complex sorts to be introduced without affecting the shape of parse trees or abstract syntax trees. Aliases are resolved by a normalization phase during parser generation, and they do not introduce auxiliary nodes.

2.2 Concrete meta-tooling

Parsing SDF is supported by *generalized* LR parser generation [15]. In contrast to plain LR parsing, generalized LR parsing is able to deal with (local) ambiguities and thereby removes any restrictions on the context-free grammars. A detailed argument that explains how the properties of GLR parsing contribute to meeting the scalability and maintainability demands of language-centered application areas can be found in [7]. The meta-tooling used for parsing in our architecture consist of a parse table generator, and a generic parse component, called `sglr`, which parses terms using these tables, and generates parse trees [16].

Parse tree representation In our architecture instantiation, the parse trees produced from generated parsers are represented in the SDF parse tree format, called AsFix [16]. AsFix trees contain all information about the parsed term, including layout and comments. As a consequence, the exact input term can always be reconstructed, and during tree processing layout and comments can be preserved. This is essential in the application area of software renovation.

Full AsFix trees rapidly grow large and become inefficient to represent and exchange. It is therefore of vital importance to have an efficient representation for AsFix trees available. Moreover, component based software development requires a uniform exchange format to share data (including parse trees) between components. The ATerm format is a term representation suitable as exchange format for which an efficient representation exists. Therefore AsFix trees are encoded as ATerms to obtain space efficient exchangeable parse trees ([5] reports compression rates of over 90 percent). In Section 3.2 we will discuss tree representation using ATerms in more detail.

Pretty-printing We use GPP, a generic pretty-printing toolset that has been defined in [13]. This set of meta-tools provides the generation of customizable pretty-printers for arbitrary languages defined in SDF. The layout of a language is expressed in terms of pretty-print rules which are defined in an ordered sequence of pretty-print tables. The ordering of tables allows customization by overruling existing formatting rules.

The standard distribution of GPP contains a formatter which operates on AsFix parse trees and supports comment preservation. An additional formatter which operates on ASTs is distributed as part of XT.

Since GPP is an open system which can be extended and adapted easily, support for new output formats (in addition to plain text, LaTeX, and HTML which are supported by default) and language specific formatters can be incorporated with little effort.

3 Abstract syntax definition and meta-tooling

A second aspect of meta-tooling for component based language tool development concerns the generation of code from *abstract* syntax definitions. Figure 4 shows the architecture of such tooling. Given an abstract syntax definition, library code is generated, which is used to represent and manipulate ASTs. The abstract syntax definition language serves as an interface description language for AST components. In other words, abstract syntax definitions serve as tree type definitions (analogous to XML's document type definitions).

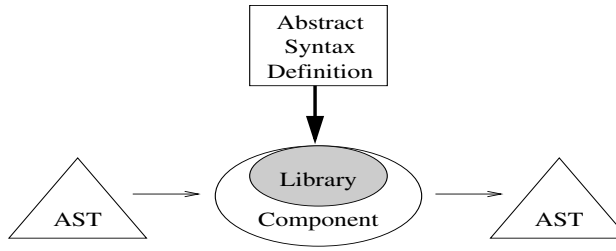


Fig. 4. Architecture for *abstract syntax* meta-tools. The abstract syntax definition, prescribing tree structure, serves as a contract between tree processing components.

3.1 Abstract syntax definition

For the specification of abstract syntax we have defined a subset of SDF, which we call AbstractSDF. AbstractSDF was obtained from SDF simply by omitting all constructs specific to the definition of *concrete* syntax. Thus, AbstractSDF allows only productions specifying prefix syntax, and it contains no disambiguation constructs or constructs for specifying lexical syntax. AbstractSDF inherits the powerful modularity features of SDF, as well as the high expressiveness concerning arbitrarily nested compound sorts. Figure 5 shows an example of an AbstractSDF definition.

The need to define separate concrete syntax and abstract syntax definitions would cause a maintenance problem. Therefore, the concrete syntax definition can be annotated with abstract syntax directives from which an AbstractSDF definition can be generated (see Section 3.3 below). These abstract syntax directives consist of optional constructor annotations for context-free productions (the "cons" attributes in Figure 3) which specify the names of the corresponding abstract syntax productions.

3.2 Abstract syntax tree representation

In order to meet our scalability demands, we will require a tree representation format that provides the possibility of efficient storage and exchange. However, we do not want a tree format that has an efficient binary instantiation only, since this makes all tooling necessarily dependent on routines for binary encoding. Having a human readable instantiation keeps the system open to the accommodation of components for which such routines are not (yet) available. Finally, we want the typing of trees to be *optional*, in order not to preempt integration with typeless, generic components. For instance, a generic tree viewer should be able to read the intermediate trees without explicit knowledge of their types.

ASTs are therefore represented in the ATerm format, which is a generic format for representing annotated trees. In [5] a 2-level API is defined for ATerms. This API hides a space efficient binary representation of ATerms (BAF) behind interface functions for building, traversing and inspecting ATerms. The binary representation format is based on maximal subtree sharing. Apart from the binary representation, a plain, human-readable representation is available.

```
definition
module Exp
exports
  syntax
    "var" ( Identifier )          Exp
    "fcall" ( Identifier, Exp* )  Exp
module Let
exports
  syntax
    "let" ( Defs, Exp )           Exp
    "where" ( Exp, Defs )         Exp
module Def
exports
  aliases
    ( Identifier Exp )+           Defs
module Main
imports Exp Let Def
```

Fig. 5. Generated AbstractSDF definition.

AbstractSDF definitions can be used as type definitions for ATerms by language tool components. In particular, the AbstractSDF definition of the parse tree formalism AsFix serves as a type definition for parse trees (See Section 2). The AbstractSDF definition of Figure 5 defines the type of ASTs representing expressions. Thus, the ATerm format provides a generic (type-less) tree format, on which AbstractSDF provides a typed view.

3.3 Abstract from concrete syntax

The connection between the abstract syntax meta-tooling and the concrete syntax meta-tooling can be provided by three meta-tools, which are depicted in Figure 6. Central in this picture is a meta-tool that derives an abstract syntax definition from a concrete syntax definition. The two accompanying meta-tools generate tools for converting full parse trees into ASTs and vice versa. Evidently, these ASTs should correspond to the abstract syntax definition which has been generated from the concrete syntax definition to which the parse trees correspond.

An abstract syntax definition is obtained from a grammar in two steps. Firstly, concrete syntax productions are optionally annotated with prefix constructor names. To derive these constructor names automatically, the meta-tool `sdfcons` has been implemented. This tool basically collects keywords and non-terminal names from productions and applies some heuristics to synthesize nice names from these. Non-unique constructors are made unique by adding primes or qualifying with non-terminal names. By manually supplying some seed constructor names, users can steer the operation of `sdfcons`, which is useful for languages which sparsely contain keywords.

Secondly, the annotated grammar is fed into the meta-tool `sdf2asdf`, yielding an AbstractSDF definition. For instance, the AbstractSDF definition in Figure 5 was obtained from the SDF definition in Figure 3. This transformation basically throws out literals, and replaces mixfix productions by prefix productions, using the associated constructor name.

Together with the abstract syntax definition, the converters `parsetree2ast` and `ast2parsetree` which translate between parse trees and ASTs are generated. Note that the first converter removes layout and comment information, while the second inserts *empty* layout and comments.

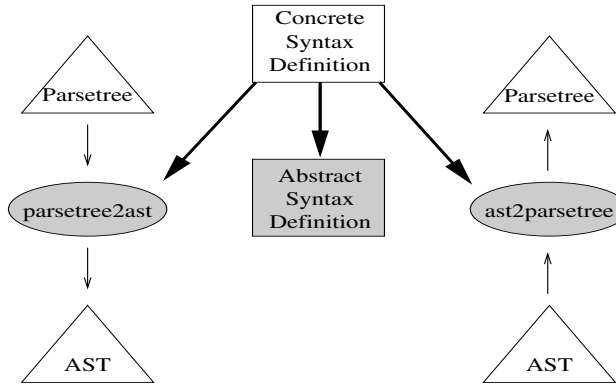


Fig. 6. Architecture for meta-tools linking abstract to concrete syntax. The abstract syntax definition is now generated from the concrete syntax definition.

Note that the high expressiveness of SDF and AbstractSDF, and their close correspondence are key factors for the feasibility of generating abstract from concrete syntax. Standard, Yacc-like concrete syntax definition languages are not satisfactory in this respect. Since their expressiveness is low, and LR restrictions require non-natural language descriptions, generating abstract syntax from these languages would result in awkwardly structured ASTs, which burden the component programmers.

4 Generating library code

In this section we will discuss the generation of library code (see Figures 2 and 4). Our language tool development architecture contains code generators for several languages and consequently allows components to be developed in different languages. Since ATerms are used as uniform exchange format, components implemented in different programming languages can be connected to each other.

4.1 Targeting C

For the programming language C an efficient ATerm implementation exists as a separate library. This implementation consists of an API which hides the efficient binary representation of ATerms based on maximal sharing and provides functions to access, manipulate, traverse, and exchange ATerms.

The availability of the ATerm library allows generic language components to be implemented in C which can perform low-level operations on arbitrary parse trees as well as on abstract syntax trees.

A more high-level access to parse trees is provided by the code generator `asdf2c` which, when passed an abstract syntax definition, produces a library of match and build functions. These functions allow easy manipulation of parse trees without having to know the exact structure of parse trees. These high-level functions are type-preserving with respect to the AbstractSDF definition.

4.2 Targeting Java

Also for the Java programming language an implementation of the ATerm API exists which allows Java programs to operate on parse trees and abstract syntax trees. As yet, there is no code generator for Java available to provide high level access and traversals of trees similar to the other supported programming languages. Such a code generator has been designed and is being developed. It will represent syntax trees as object trees, and tree traversals will be supported by generated libraries of refinable visitors.

4.3 Targeting Stratego

Our initial interest was to apply our meta-tooling to program transformation problems, such as automatic software renovation. For this reason we selected the transformational programming language Stratego [17] as the first target of code generation. Stratego offers powerful tree traversal primitives, as well as advanced features such as separation of pattern-matching and scope, which allows pattern-matching at arbitrary tree depths. Furthermore, Stratego has built-in support for reading and writing ATerms. Stratego also offers a notion of pseudo-constructors, called *overlays*, that can be used to operate on full parse trees using a simple AST interface.

Two meta-tools support the generation of Stratego libraries from syntax descriptions. The library for AST processing is generated by `asdf2stratego` from an AbstractSDF definition. The library for combined parse tree and AST processing is generated by `sdf2stratego` from an SDF grammar. The latter library subsumes the former.

The Stratego code generation allows programming on parse trees as if they were ASTs. Underneath such AST-style manipulations, parse trees are processed in which hidden layout and literal information is preserved during transformation. This style of programming can be mixed freely with programming directly on parse trees. Since Stratego has native ATerm support, there is no need for generating library code for reading and writing trees.

4.4 Targeting Haskell

Work has also been done on targeting Haskell. Code generated in this case is of various kinds. Firstly, datatypes are generated to represent parse trees and ASTs. These datatypes are quite similar to the signatures generated for Stratego. Secondly, code is generated for reading ATerm representations into these Haskell datatypes and writing them to ATerms. Finally, full-fledged transformation frameworks consisting of (monadic) paramorphisms and corresponding algebras are generated to facilitate purely functional transformational programming. The reader is referred to [14] for details and for a software renovation case study.

Note that not only general purpose programming languages of various paradigms can be fitted into our architecture, but also more specialized, possibly very high-level languages. An attribute grammar system, for instance, would be a convenient tool to program certain tree transformation components.

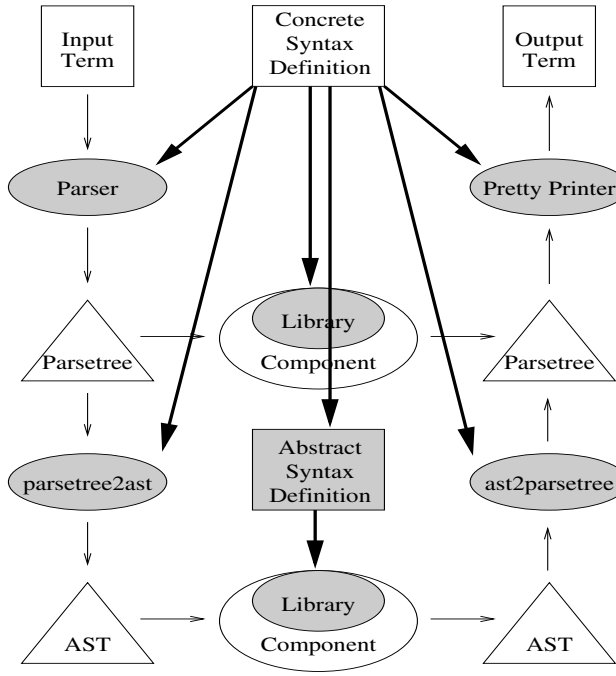


Fig. 7. Complete meta-tooling architecture. The grammar serves as the contract governing all tree exchange.

5 A comprehensive architecture

Combining the partial architectures of the foregoing subsections leads to the complete architecture in Figure 7. This figure can be viewed as a refinement of our first general architecture in Figure 1, which does not differentiate between concrete and abstract syntax, or between parse trees and ASTs.

The refined picture shows that all generated code (libraries and components), and the abstract syntax definition stem from the same source: the grammar. Thus, this grammar serves as the single contract that governs the structure of all trees that are exchanged. In other words, all component interfaces are defined in a single location: the grammar. (When several languages are involved, there are of course equally many grammars.) This single contract approach eliminates many maintenance headaches during component based development. Of course, careful grammar version management is needed when maintenance due to language changes is not carried out for all components at once.

5.1 Grammar version management

Any change to a grammar, no matter how small, potentially breaks all tools that depend on it. Thus, sharing grammars between tools or between tool components, which is

a crucial feature of our architecture, is potentially at odds with grammar *change*. To pacify grammar change and grammar sharing, grammar management is needed.

To facilitate grammar version management, we established a *Grammar Base*, in which grammars are stored. Furthermore, we subjected the stored grammars to simple schemes of grammar version numbers and grammar maturity levels.

To allow tool builders to unequivocally identify the grammars they are building their tool on, each grammar in the Grammar Base is given a name and a version number. To give tool builders an indication of the maturity of the grammars they are using to build their tools upon, all grammars in the Grammar Base are labeled with a maturity level. We distinguish the following levels:

volatile	The grammar is still under development.
stable	The grammar will only be subject to minor changes due to bug fixing.
immutable	The grammar will never change.

Normally, a grammar will begin its life cycle at maturity level *volatile*. To build extensive tooling on such a grammar is unwise, since grammar changes are to be expected that will break this tooling. Once confidence in the correctness of the grammar has grown, usually through a combination of testing, bench-marking, and code inspection, it becomes eligible for maturity level *stable*. At this point, only very local changes are still allowed on the grammar, usually to fix minor bugs. Tool-builders can safely rely on stable grammars without risking that their tools will break due to grammar changes. Only a few grammars will make it to level *immutable*. This happens for instance when a grammar is published, and thus becomes a fixed point of reference. If the need for changes arises in grammars that are stable or immutable, a *new* grammar (possibly the same grammar with a new version number) will be initiated instead of changing the grammar itself.

5.2 Connecting components

The connectivity to different programming languages allows components to be developed in the programming language of choice. The use of ATerms for the representation of data allows easy and efficient exchange of data between different components and it enables the composition of new and existing components to form advanced language tools.

Exchange between components and the composition of components is supported in several ways. First, components can be combined using standard scripting techniques and data can be exchanged by means of files. Secondly, the uniform data representation allows for a sequential composition of components in which Unix pipes are used to exchange data from one component to another. Finally, the ToolBus [3] architecture can be used to connect components and define the communication between them. This architecture resembles a hardware communication bus to which individual components can be connected. Communication between components only takes place over the bus and is formalized in terms of Process Algebra [1].

6 Applications

Only preliminary experience is available about actually applying the meta-tooling presented in the previous sections. We will present a selection of such experiences.

To start with, the meta-tooling has been applied for its own development, and for the development of some other meta-tools that it is bundled with in the Transformation Tools package XT. These bootstrap flavored applications include the generation of an abstract syntax definition for the parse tree format AsFix from the grammar of SDF. From this abstract syntax definition, a modular Stratego library for transforming AsFix trees was generated and used for the implementation of some AsFix normalization components. Also, the tools `sdf2stratego`, `sdfcons`, `asdf2stratego`, `sdf2asdf`, and many more meta-tools were implemented by parsing, AST processing in one or more components, and pretty-printing.

Apart from SDF and AbstractSDF, the domain specific languages BOX (for generic formatting), and BENCH (for generating benchmark reports), have been implemented with syntax-driven meta-tooling. In the BOX implementation, a grammar for pretty-print tables was built by reusing the SDF grammar and the BOX grammar. New BOX components were implemented in Stratego and connected to existing BOX components programmed in other languages.

The generated transformation frameworks for Haskell are being applied to software renovation problems. In [14], a COBOL renovation application is reported. It involves parsing according to a COBOL grammar, applying a number of function transformers to solve a data expansion problem, and unparsing the transformed parse trees. The functional transformers have been constructed by refining a transformation framework generated from the COBOL grammar. Application to the development of documentation generators [10] has commenced.

7 Related work

Syntax-driven meta-tools for language tool development are ubiquitous, but rarely do they address a combination of features such as those addressed in this paper. We will briefly discuss a selection of approaches some of which attain a degree of integration of various features.

- Parser generators such as Yacc [12] and JavaCC are meta-tools that generate parsers from syntax definitions. Compared with SDF and `sglr` , they offer poor support for *modular* syntax definition, their input languages are not sufficiently declarative to be reusable for the generation of other components than parsers, and they do not generally target more than a single programming language.
- The language SYN [4] combines notations for specifying parsers, pretty-printers and abstract syntax in a single language. However, the underlying parser generator is limited to LALR(1), in order to have both parse trees and ASTs, users need to construct two grammars, and code the mapping between trees by hand. Moreover, the expressiveness of the language is much smaller than the expressiveness of SDF, and the language is not modular. Consequently, SYN and its underlying system can not meet our adaptability, scalability and maintainability requirements.

- Wile [20] describes derivation of abstract syntax from concrete syntax. Like us he uses a syntax description formalism more expressive than Yacc's BNF notation in order to avoid warped ASTs. Additionally, he provides a procedure for transforming a Yacc-style grammar into a more "tasteful" grammar. His BNF extension allows annotations that steer the mapping with the same effect as SDF's aliases. He does not discuss automatic name synthesis.
- AsdlGen [19] provides the most comprehensive approach we are aware of to syntax-driven support of component-based language tools. It generates library code for various programming languages from abstract syntax definitions. It offers ASDL as abstract syntax definition formalism, and *pickles* as space-efficient exchange format. It offers no support for dealing with concrete syntax and full parse trees.
  AsdlGen targets more languages than our architecture instantiation does at the moment. The choice of target languages, including C and Java, has presumably motivated some restrictions on the expressiveness of ASDL. ASDL lacks certain modularity features, compared to AbstractSDF: no mutually dependent modules, and all alternatives for a non-terminal must be grouped together. Furthermore, ASDL is much less expressive. It does not allow nesting of complex symbols, it has a very limited range of symbol constructors, and it does not provide module renamings or parameterized modules.
  Unlike ATerms, the exchange format that comes with ASDL is always typed, thus obstructing integration with generic components. In fact, the compression scheme of ASDL relies on the typedness of the trees. The rate of compression is significantly smaller than for ATerms [5]. Furthermore, pickles have a binary form only.
- The DTD notation of XML [8] is an alternative formalism in which abstract syntax can be defined. Tools such as HaXML [18] generate code from DTDs. HaXML offers support both for type-based and for generic transformations on XML documents, using Haskell as programming language. Other languages are not targeted. Concrete syntax support is not integrated.
  XML is originally intended as mark-up language, not to represent abstract syntax. As a result, the language contains a number of inappropriate constructs, and some awkward irregularities from an abstract syntax point of view. XML also has some desirable features, currently not offered by AbstractSDF, such as annotations, and inclusion of DTDs (abstract syntax definitions) in documents (abstract terms).
- Many elements of our instantiation of the architecture for syntax-driven component-based language tool development were originally developed in the context of the ASF+SDF *Meta-Environment* [2,11,9]. This is an integrated language development environment which offers SDF as syntax definition formalism and the term rewriting language ASF as programming language. Programming takes place directly on concrete syntax, thus hiding parse trees from the programmers view. Programming, debugging, parsing, rewriting and pretty-printing functionality are all offered via a single interactive user interface. Meta-tooling has been developed to generate ASF-modules for term traversal from SDF definitions [6].
  The ASF+SDF Meta-Environment offers a single programming language (ASF), programming on abstract syntax is not supported. Support for component-based development is (currently) limited to gluing compiled ASF programs that read and write flat terms.

To provide support for component-based tool development, we have adopted the SDF, AsFix, and ATerm formats from the ASF+SDF Meta-Environment as well as the parse table generator for SDF, the parser `sglr` , and the ATerm library. To these we have added the meta-tooling required to complete the instantiation of the architecture of Figure 7. In future, some of these meta-tools might be integrated into the Meta-Environment.

8 Contributions

We have presented a comprehensive architecture for syntax-driven meta-tooling that supports component based language tool development. This architecture embodies the vision that grammars can serve as contracts between components under the condition that the syntax definition formalism is sufficiently expressive and the meta-tools supporting this formalism are sufficiently powerful. We have presented our instantiation of such an architecture based on the syntax formalism SDF. SDF and the tools supporting it have agreeable properties with respect to modularity, expressiveness, and efficiency, which allow them to meet scalability and maintainability demands of application areas such as software renovation and domain-specific language implementation. We have shown how abstract syntax definitions can be obtained from grammars. We discussed the meta-tooling which generates library code for a variety of programming languages from concrete and abstract syntax definitions. Components that are constructed with these libraries can interoperate by exchanging ATerms that represent trees.

Acknowledgments The authors thank Arie van Deursen and Eelco Visser for valuable discussions.

References

1. J. Baeten and W. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.
2. J. A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989.
3. J. A. Bergstra and P. Klint. The ToolBus coordination architecture. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models (COORDINATION'96)*, volume 1061 of *Lecture Notes in Computer Science*, pages 75–88. Springer-Verlag, 1996.
4. R. J. Boulton. SYN: A single language for specifying abstract syntax trees, lexical analysis, parsing and pretty-printing. Technical report, Computer laboratory, University of Cambridge, 1996.
5. M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Software, Practice & Experience*, 30(3):259–291, 2000.
6. M. G. J. van den Brand, M. P. A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In *Proceedings Fourth Working Conference on Reverse Engineering*, pages 144–153. IEEE, 1997.
7. M. G. J. van den Brand, M. P. A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In *Proceedings of the sixth International Workshop on Program Comprehension*, pages 108–117. IEEE, 1998.

8. T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. Technical Report REC-xml-19980210, World Wide Web Consortium, 1998.
9. A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.
10. A. van Deursen and T. Kuipers. Building documentation generators. In *Proceedings; IEEE International Conference on Software Maintenance*, pages 40–49. IEEE Computer Society Press, 1999.
11. 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.
12. S. C. Johnson. YACC - Yet Another Compiler-Compiler. Technical Report Computer Science No. 32, Bell Laboratories, Murray Hill, New Jersey, 1975.
13. M. de Jonge. A Pretty-Printer for Every Occasion. In I. Ferguson, J. Gray, and L. Scott, editors, *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000)*. University of Wollongong, Australia, 2000.
14. J. Kort, R. Lämmel, and J. Visser. Functional transformation systems. In *Proceedings of the 9th International Workshop on Functional and Logic Programming*, Sept. 2000.
15. J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.
16. E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.
17. E. Visser. Strategic pattern matching. In *Rewriting Techniques and Applications (RTA'99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 30–44. Springer-Verlag, 1999.
18. M. Wallace and C. Runciman. Haskell and XML: Generic Combinators or Type-Based Translation? In *International Conference on Functional Programming (ICFP'99), Paris, France, ACM SIGPLAN*, Sept. 1999.
19. D. C. Wang, A. W. Appel, J. L. Korn, and C. S. Serra. The Zephyr abstract syntax description language. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, pages 213–28, Berkeley, CA, Oct. 15–17 1997. USENIX Association.
20. D. S. Wile. Abstract syntax from concrete syntax. In *Proceedings of the 19th International Conference on Software Engineering (ICSE '97)*, pages 472–480, Berlin - Heidelberg - New York, May 1997. Springer.

Generic Components: A Symbiosis of Paradigms

Martin Becker

System Software Research Group
University of Kaiserslautern
D-67653 Kaiserslautern, Germany
mbecker@informatik.uni-kl.de

Abstract. Software reuse and especially the paradigm of software components are promising approaches to increase the efficiency of software development. One of the basic problems of software reuse, however, is the tradeoff between the abstraction from project-specific aspects and the actual contribution of a reusable artifact. Stringent resource constraints further complicate the application of these approaches in domains, where efficient and therefore specialized solutions are required, e.g. in the domain of embedded systems. Generic components – designed to be adaptable to new application scenarios – allow to overcome these limitations, esp. if they automate the essential modifications. This paper presents a concept of generic components that has been developed to facilitate the construction of highly specialized embedded operating systems. Besides the illustration of the underlying concept, the paper discusses the external representation of generic components and the internal realization of the required variability and reflects some of our experiences in constructing generic components.

1 Introduction

The highly competitive and dynamic field of software development implies the need to continuously increase the efficiency of the construction and evolution of software products. For a long time reuse of software artifacts in general and of standard components in particular have been promising approaches [16, 18] to achieve this goal through the consequent and efficient exploitation of the commonalities among produced software artifacts. However, there are inherent problems with both techniques. Firstly, software artifacts have to be made sufficiently abstract to be reusable, i.e. to be applicable in different deployment situations, while at the same time need to be sufficiently concrete to bear a reuse benefit at all. With component-based software development this gets even worse, since the components are typically intended to be reused in a black box fashion and thus often do not support their customization at compile-time in regard to individual demands. Secondly, the evolution of prefabricated software artifacts – e.g. caused by changing requirements – in most cases results in a rise of complexity that further degrades the performance and thus often prohibits their reuse. This holds especially if tight resource constraints have to be met, e.g. in the domain of embedded systems.

G. Butler and S. Jarzabek (Eds.): GCSE 2000, LNCS 2177, pp. 100-113, 2001.

To deal with the required variability, generative programming techniques, e.g. as presented in [7], are sound approaches, since they also pursue reuse on the level of the production process instead solely on the product level. The variability among the artifacts thus can be more effectively managed and even be formulated in the context of the problem space. But there are problems with pure generative approaches, too. Domain-specific default assumptions have to be applied, otherwise the complexity of the problem specification and the underlying generator would be of the same magnitude as the implementation complexity of the whole problem itself. Again, the trade-off between default solutions and the need for individual optimizations is likely to appear. As a consequence manual specializations have to be considered within the reuse activities as well, if the efficiency of the results is essential. The challenge is to combine different reuse strategies in a way that exploits the benefits of each approach. Code generation can be used to supplement the reuse of software components by automating their otherwise time-consuming and error-prone customization. Pre-planned manual optimizations within the reusable artifacts can be supported by providing adequate implementation constraints and knowledge, etc.

We have developed an approach based on the component and the generation paradigm in the domain of embedded operating systems to address the aforementioned problems. It has been coined by the stringent resource constraints, such as scarce memory and low cost CPUs, in combination with varying nonfunctional requirements, like robustness and timeliness. The resulting efficiency constraints – often the most crucial criterion for the success or failure of a software project within this domain – make conventional reuse approaches especially problematic in this context, since they necessitate even unique tailor-made solutions that are very hard to reuse efficiently in following projects. To address this problem, the concept of *generic components* has been developed. Generic components provide means to ease their tailoring according to individual requirements; they combine the technique of reusing prefabricated software artifacts with the goal of producing specialized software. Most of the variability provided by them can be automated by *generators* accomplishing even fine-grained optimizations on the level of source code, but also unique customizations are supported. Obviously, the combination of the component and the generation paradigm only becomes possible through a weakening of the conventional component notion. With generic components also grey-box reuse has to be considered.

Although this integrative concept has been developed for the domain of embedded operating systems, it should also be applicable in other domains – provided that a known and manageable set of component variants is likely to cover the majority of possible requirements. This paper presents our concept of generic components and focuses on their external representation and some general facts concerning the internal realization of the provided variability. The rather process-related questions of mapping requirements to appropriate component selections and configurations in our approach is covered by companion papers [3, 4].

The remainder of this paper is structured as follows: Section 2 introduces the concept of generic components in-depth. Section 3 covers the external representation of generic components. In Section 4 several approaches to implement generic components are discussed. Finally, in Section 5, we draw some conclusions.

2 The Concept of Generic Components

Conventional component-based software development not considering customizations at compile-time reaches its limits where tailor-made solutions are required to meet stringent nonfunctional requirements. To address this, the concept of generic components [2] has been developed, an approach for delivering software solutions that – despite of relying on the reuse of components – are customized to the settings of a specific application domain. Generic components combine the techniques of reusing prefabricated software artifacts with the goal of producing specialized software.

A *generic component* can be defined as a software module allowing to choose its properties to a certain degree without necessarily having to write or change code manually. The choice of properties may concern both functional and nonfunctional aspects, e.g. the choice of algorithms or data structures and the adaptation of the component's interface. To support this kind of customization, so-called *generic parameters* together with their respective range of supported values are identified at the component's *customization interface*. These are the externally visible setscrews to control the instantiation of the generic component into a conventional one (cf. Fig. 1). Within the generic components so-called *variation points* – placeholders for the variable parts of code – can be identified that are replaced with concrete code fragments produced by *generators* in accordance to the actual values chosen for the generic parameters during the instantiation. Typically only few parameters have an impact on a variation point; they form the input to the specific generator that is associated with this variation point. To subsume, customizing a generic component hence means to choose actual values for the supplied generic parameters, and let appropriate generators accomplish the necessary modifications. Besides a pure reduction of customization effort, customizations on a higher level of abstraction can be supported through the application of generative techniques, e.g. template meta programming, or sophisticated code compilers. The application developer, for instance, applying generic components to construct embedded control systems does not need to know in detail what changes within the components are required to provide thread preemption or a `delay()` functionality. He only has to decide if the functionality is needed and has to be informed about the additional resource requirements.

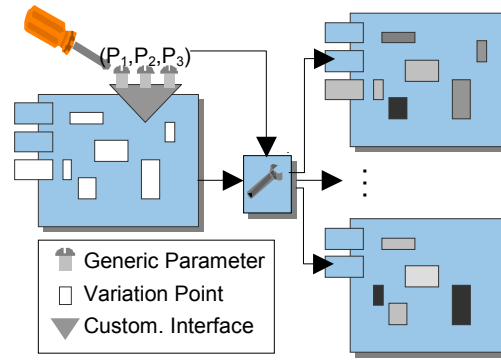


Fig. 1. Conceptual view on generic components.

From the conceptual point of view a generic component is a kind of abstract representative of a set of closely related software components with slightly different properties. Since it is only sensible to integrate such components in a generic component that share a considerable amount of common code, generic components – according to [17] – can be considered as a component family. The different family members can be distinguished by the tuple of actual values, chosen from the subset of the Cartesian Product over the generic parameters that contains the consistent parameter selections.

As with program families, a thorough analysis of the presumably needed properties, and the resulting commonalities and variabilities should take place before generic components are implemented. Domain Analysis methods [8, 18] or the Multi-Paradigm Design [6] can be used to this end.

2.1 Generic Parameters

Generic parameters – the pendants of run-time configuration parameters at compile-time – allow to tailor the component's properties by simply choosing one of the supported parameter values. Trivial properties like the value of a constant can be instantiated as well as code fragments realizing special data structures or algorithms. Specific adaptations concerning the runtime environment of the component, e.g. the type of used hardware, can thus be handled within the component, instead of providing external wrappers causing additional costs. Especially in the case of directly interacting components, using function calls to communicate with each other, adaptations of the calling interface are very useful. Furthermore, the interfaces of generic components can be the target of the customization. Superfluous complexity [5] can be reduced in this way by providing only the needed functionality. Finally, a tailoring of nonfunctional properties of the component can be supported, e.g. an optimization concerning the processing time or memory consumption at run-time. This tailoring is usually driven by the wish to construct software components that meet the essential given requirements and moreover support as many as possible of the desired properties, e.g. little resource demands, reduction of the interface complexity, etc.

Three major approaches to tailor code that is used to replace a variation point can be identified (cf. Fig. 2). Variation points are filled by *selection* if they are simply replaced with discrete entities that are selected from a finite set of alternatives. Entities in this sense are pre-built parts of a component implementation, e.g. data structures, algorithms, or other code fragments. In the case of *generation*, the variation points are placeholders for code fragments produced by associated generators depending on the provided actual values of the generic parameters. The range of generator functionality can reach from the simple insertion of values for constants over the expansion of small macros to sophisticated code generation, where abstract descriptions in form of the actual values are translated into concrete artifacts. With this type any generative technique can be utilized in generic components. Where the aforementioned tailoring types do not provide sufficient flexibility, variation points can be replaced by parameter *substitution*, i.e. a variation point is substituted with the actual value of the respective parameters. In this way, user-defined code can be inserted into the component. This type of tailoring offers the highest degree of flexibility to the user and is typically used to handle code

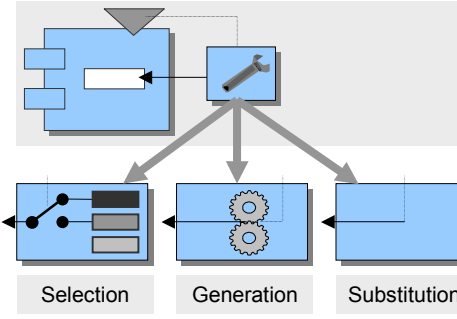


Fig. 2. Different types to tailor code that replaces a variation point.

fragments that are unique in each component instantiation, e.g. individual optimizations. Besides the pure replacement of the parameter an important and nontrivial task during the instantiation of a generic component can be the monitoring of the insertion. In this case, the generator checks the provided code against some constraints or predefined limits formulated by the implementer of the component. If a generic parameter only controls customizations of the same type, esp. in the case of substitution, it is sensible to speak of a *selection*, *generation* or *substitution parameter*. The mentioned tailoring types represent different kinds of reuse. While a selection reuses concrete artifacts, generation reuses the production methods, and substitution supports the reuse of knowledge and constraints about prospective solutions.

In principle, it is reasonable to perform the customization hierarchically, i.e., new generic parameters can be introduced with the code fragments that replace variation points. However, a recursive introduction of new parameters while choosing parameter settings adds a considerable amount of complexity to the configuration process. Therefore, it is sensible to restrict the introduction of new generic parameters to the tailoring by selection as illustrated in Fig. 3. A typical scenario where hierarchical parameter introductions are used is the following: a fixed number of implementations exist for a variation point, e.g. different hardware drivers , but the opportunity to add a new implementation has to be given to the user, too. This can be realized with a selection between the existing code fragments and a frame that provides a substitution parameter. If the user decides to provide new code, he selects the frame and afterwards supplies the new code by substituting the new parameter.

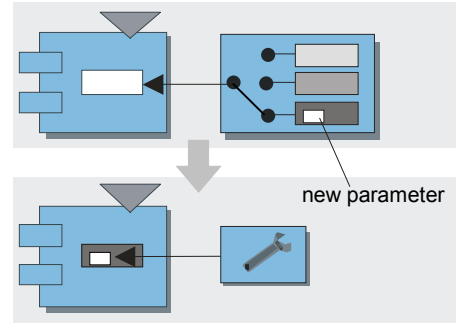


Fig. 3. Hierarchical tailoring: introduction of a new parameter though selection.

2.2 Objectives of the Approach

The concept of generic components has been developed to make the component paradigm applicable even in domains where the black-box reuse of components typically will not work, i.e. that tailor-made solutions are required for some reasons. Through the identification of variation points and of generic parameters the customization at compile time is supported allowing the tailoring of components in respect to stringent requirements. Doing so, the generic components help to overcome the inherent tension between reusability and usefulness of conventional components. If feasible, the tailoring complexity can be captured in pre-built fragments or sophisticated generators providing implementations for the variation points' instantiation. Otherwise substitution parameters can be used to support the creation of new specializations through the clear identification of the relevant variation points and the specification of implementation constraints and knowledge. Some kind of abstract components can be realized in this way that are very useful in the case of highly individual and therefore not

reusable aspects, or yet unknown requirements and solutions. The clear identification and the factorization of the common and variable parts is advantageous for the following reasons: Firstly, the evolution of the generic components gets easier, e.g. as common code is not replicated in several component variants or as new variants created by the user can be easily integrated into the respective generators. Secondly, the number of related components in reuse pools and therefore the complexity of their retrieval can be reduced in case of orthogonal variabilities. If a thread manager component, for instance, offers generic parameters for the number of supported threads and the kind of the used stacks, then a lot of different thread manager variants would have to be in the reuse pool in order to meet stringent requirements. In the case of a generic component only one component with corresponding generators is needed. With the ability to flexibly adapt the implementation of the reused components, conventional approaches to improve component reusability, which typically interfere with the overall system performance, can be avoided, e.g. the strict uncoupling of components, and large inheritance hierarchies. To subsume, the concept of generic components approximates the advantages of manual customization and optimization, while on the other hand sidestepping the obstacles of time-consuming and error-prone coding by reusing prefabricated and tested implementations.

3 The User's Perspective on Generic Components

The reusability of software components strongly depends on their external representation. Comprehensive descriptions of the provided functionality, the interfaces, the dependencies on other artifacts and their architectural embedding, etc are essential, especially as the components are typically reused in a black-box fashion. This also holds for generic components. Due to the variability provided by the generic components, however, further information to represent the generic component in a comprehensive manner is required. From an external point of view, generic components mainly differ from conventional ones in the provision of an additional customization interface and in the fact, that some of the components' properties are kept variable. Furthermore, some mechanism is supplied that supports the automatic tailoring of the component according to the actual values chosen for the generic parameters. In the following, however, we will concentrate on the customization interface and elaborate the required information that has to be presented to the user of the generic component to support its consistent configuration and application.

3.1 Customization Interface

A generic component in our sense is a kind of abstract representative of a family of software components. It can be put into concrete shapes with the assignment of actual values to the provided generic parameters. Therefore, at least the generic parameters must be clearly identified to give the user an idea what can be tailored at all – but that will not suffice. To scope the variability of the parameters, the range of supported values for each of them has to be specified as well as the interdependencies among them, typically restricting the number of consistent parameter selections. As the actual

values of substitution parameters are directly inserted into the generic component, it is impossible to enumerate all supported values for this kind of parameter – otherwise the substitution parameter should be replaced by a selection parameter. However, at least the language and an abstract description of the expected code fragment can be stated to facilitate the consistent parameter assignment in that case.

Since the users of generic components are naturally more interested in the results of their tailoring than the tailoring itself, further information regarding the variable properties should be given too. In our approach, this information is structured in so-called feature types and features. A feature type comprises clearly distinguishable features and represents a variable property of the component, e.g. scheduling strategy, memory management strategy, or synchronization mechanism. Feature types thus span the space of components that can be instantiated from a generic component. This information, together with the description of the common features, is typically used to determine in a very early stage, whether a generic component is applicable for the specific problem or not.

Obviously the feature types of a generic component are somehow correlated with its generic parameters. If this correlation is explicitly represented in the customization interface, it can be used to determine the concrete features of the tailored component before the final instantiation step. This feedback can be used for composability tests and for further adjustment of the component, and therefore facilitates the selection of an appropriate actual parameter set. For each feature type, a functional dependency on a subset of the generic parameters can be identified. In our approach, this set of feature-dependency functions is used to represent the correlation between a feature type and its parameters. One special case is worth to be mentioned: feature-dependency functions that are invertible can be used favorably to determine parameters based upon the desired features. This significantly facilitates the customization of a component. Fig. 4 subsumes the aforementioned points into a formal representation of the customization interface. As the comprehensible description of the listed aspects is non-trivial and expensive, it may be reasonable to provide less information at the cost of user support, e.g. the feature-dependency functions could be omitted.

Representation of the customization interface CI:

$$CI =_{df} (P, (P_i), VS, F, (F_i), (FD_i))$$

With:

P	$=_{df}$	$\{P_1, \ldots, P_n\}$, the set of generic parameters
P_i	$=_{df}$	$\{p_1, \ldots, p_{m(i)}\}$, supported values for each parameter
PP	$=_{df}$	$P_1 \times \ldots \times P_n$, the space of parameter selections
VS	$=_{df}$	$\{pp \in PP \mid pp \text{ is a valid selection}\}$, the set of valid selections
F	$=_{df}$	$\{F_1, \ldots, F_k\}$, set of feature types
F_i	$=_{df}$	$\{f_1, \ldots, f_{m(i)}\}$, variety of features for the feature type F_i
PF	$=_{df}$	$F_1 \times \ldots \times F_k$, the space of varying features
FD_i	$=_{df}$	$VS \rightarrow F_i$, feature-dependency function

Fig. 4. Representation of the customization interface.

3.2 Parameter Dependencies

After the discussion of the information that has to be presented at the configuration interface, we take a closer look at the configuration step itself. In practice, the generic

parameters often are not independent of each other. Typically, it is possible to identify hierarchical dependencies between them; i.e. the presence of some generic parameter depends on the choice of actual values for others. For instance, a generic parameter that is only supplied in the context of a distinct data structure does not have to be considered if the data structure is not used in an actual configuration. Although this information is already represented by the set of valid selections, the customization complexity for the user can be further reduced if these parameter dependencies are explicitly considered during the configuration step. Instead of a flat configuration, where all parameters are visible throughout the customization, although a part of them is not really needed, a hierarchical configuration should be preferred. Here only those parameters that bear a meaning for the current level of choices are presented. This approach typically results in less parameters and a more intuitive way of configuration, but requires more support in the customization interface and the instantiation of the component. Eventually some backtracking has to be done in the case that a lower level of configuration reveals the insufficient selection of a parameter on a higher configuration level. In Fig. 5 an example to illustrate the different configuration approaches is given. The arrow between A and B denotes that B only has to be considered in the case A is set to 2. Obviously, the hierarchical approach facilitates the configuration, e.g. no values have to be selected for parameters B and E unless the value 2 is chosen for A. The example additionally points out the two major reasons for hierarchical dependencies. They are either caused by an implicit requires-dependency as shown between A and B, or hierarchical tailoring (cf. Section 2) as with A and C.

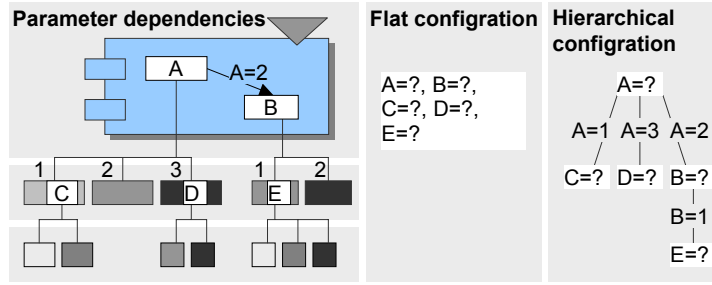


Fig. 5. Parameter dependencies and the different configuration approaches.

3.3 Tool Support

Several approaches to represent the customization interface and to support the selection of actual values in a user-friendly way are reasonable. However, it becomes clear that tool support, esp. for the feature-dependency functions and hierarchical configuration, is indispensable. Therefore, we apply tools [3] based on the technique of extended design-spaces, a multi-dimensional classification scheme that allows to express such correlations between features and parameters as well as hierarchies of parameters in a very natural and concise way [1, 14, 15]. The required information is kept in so-called component spaces that are used by our tools to control the customization process. Besides the customization of generic components, a mechanism has to be supplied that supports the automatic tailoring of the component in correspondence with the chosen configuration. These two mechanisms can favorably be integrated

into a single tool, thus automating the customization and the instantiation of generic components. Our D-Space-1 tool [3], for instance, facilitates the application of the customization interface of our generic components through a graphical interface and controls the tailoring of the generic components by calling the appropriate generators, which accomplish the specified modifications.

4 Internals of Generic Components

From the implementers' point of view, the descriptive elements of generic components – especially their code – significantly differ from those of conventional ones. Generic components comprise a set of variable implementations and generic parameters to determine them. In addition to the conventional description elements, such as code, header files, and resources, the required mechanisms to automate the tailoring must be supplied in some way by the implementer of the generic component. A wealth of well-known possibilities addressing different facets like code organization and modification, customization of structure and behavior, extensibility of existing implementations, or code generation are available for this kind of meta programming, e.g. appropriate language constructs, pre-processors, external generators. All of these viable implementation approaches have to somehow integrate the common and the variable description parts that form a generic component. Before discussing the different tailoring techniques themselves, we want to look at the preconditions needed to implement the tailoring facilities in the generic components.

4.1 Integration of Commonalities and Variabilities

Different techniques can be applied to tailor the description fragments that replace the variation points in the concrete component implementation. All of these approaches have at least two concepts in common. Firstly, the variable description parts have to be associated with the common ones in some way. Secondly, links between the internally used generic parameters and their externally visible pendants at the customization interface have to be established to propagate the external settings. Besides the mere description of the variability, the following aspects thus have to be addressed in some manner: the identification of the variation points, the association of tailoring transformations, and the representation of the generic parameters. Although the clear identification of the variation points might seem to be dispensable in some cases, esp. if the same language is used to describe the common and variable parts of a generic component, it is extremely helpful if the variability of the generic component has to be revised or new variants have to be added. With this little overhead, tool support for the localization of variable implementations throughout the generic component can easily be provided. The variation points can be identified by a special markup that obviously must not be used within the description of the common parts. For this reason, the number of different markups should be kept as small as possible. The coherent use of unique markups further eases the construction of tools for the identification of the variation points within the generic component in that they do not have to be aware of the actual techniques applied to describe the variability.

Concerning the association of the tailoring transformations with the variation points, two major methods are conceivable: inline and link. The tailoring is specified inline, if it is described directly in the variation point. Since this results in a mixture of different kind of description types in most of the cases, doing so is only advisable if the transformation is not too complex and cannot be applied elsewhere. More flexibility is provided through an external tailoring description or a stand-alone tool that is linked with the variation points. This link can be realized with an entry in the variation point that is used to call the external generator, or an external identification of the variation point that can be handled with a unique variation point identifier. In the latter case, several variation points can be substituted with one transformation. As generic parameters typically affect multiple variation points, the required elementary transformations sometimes can be combined into one comprehensive transformation.

As stated above, the generic parameters that control the instantiation of a variation point have to be represented in a way that allows the propagation of the respective actual values provided at the customization interface. If a transformation is specified inline no further representation of the generic parameters is required, since they can be identified and replaced within the transformation descriptions if a uniform naming scheme is used. If a transformation is associated with a variation point via a link, the required parameters have to be specified along with the link somehow, e.g. as call parameters of the external tool.

4.2 Tailoring Techniques

After the discussion of methods to integrate the common and variant parts of generic components, we will now focus on the tailoring techniques applicable to realize the variability. As the concept of generic components is primarily aimed at compile-time tailoring, almost any text transforming or text generating technique can be deployed. Based on our experience, the necessary tailoring can be favorably realized through a combination of the following techniques:

Separate files – separately implemented variant parts of components are selected by choosing the corresponding source file for compilation and linking. This is the most simple approach; neither modifications of the descriptions nor special actions for instantiation are necessary. The selection of the files can be accomplished through external file management tools like RCS or CVS.

Pre-processors – scan the descriptions for particular tokens – e.g. programming language constructs or specific pre-processor statements –, parse their semantic context, and perform associated pre-defined actions. In their full extent, pre-processors are problem-specific compilers similar to those for simple programming languages and can be built using scanner and parser generators like lex/yacc. With special preprecessors other programming paradigms such as Aspect-Oriented Programming [13] can be applied within generic components, e.g. to realize efficient implementations of variable nonfunctional aspects. However, a great deal of variability can be realized with the pre-processor functionality provided by the programming languages or scripting tools like sed or perl in a very straightforward way.

Generic mechanisms of programming languages – most programming languages and the corresponding compilers somehow support generic programming, e.g.

through object-orientation, templates, or compile time parameters and optimizations. Wherever it does not contradict efficiency and comprehensibility, the deployment of such mechanisms is a natural and well-known way to realize the required variability. Examples are the approaches of template meta programming [20][19] and generative programming [7] that both extensively utilize the template mechanism offered by C++ to build variable implementations. One drawback of these mechanisms is that the implementation of the common and the variable parts are mixed up, thus complicating the location of the variable implementations in most cases.

Code generators – the most powerful, but also most complex way to perform the tailoring. Higher level descriptions of the desired functionality are passed through the generic parameters to external generators producing tailored description parts, which afterwards replace the correspondig variation points. The use of generators is especially tempting wherever the modification of pre-built components does not appear to be sufficient to cope with heavily varying and unforeseen requirements. One problem with generators is that they must apply domain-specific defaults for those aspects that are not explicitly specified by their input, mostly resulting in code that in less efficient as manually optimized code.

4.3 A Glance at Implementations

In the following, two approaches to implement the tailoring of a generic component are exemplarily presented. In the first example given in Fig. 6, the UNIX stream editor sed [9] is employed to tailor the get function of a generic dispatcher that supports multi-processor environments. To allow for optimizing the source code for single-processor environments, references to the parameter n have been marked by comments of the form GPx_My. An appropriate sed script can search for the respective lines of source code and apply whatever changes are necessary. According to the sed command syntax, line 3 for instance searches for lines with the mark GP2_M10, then takes the first occurrence of the

```
TCBptr Dispatcher::get(size_t n){// GP2_M10 TP:CHANGE
   TCBptr old;
   old = running[n];    // GP2_M11 TP:CHANGE
   old->setLastCPU(n);  // GP2_M13 TP:REMOVE
   // GP5_M7 TP:INSERT
   ProcContext.All.Save(old->TCB.Register);
   return old;
}
```

source code

```
s/\(Dispatcher\.[hc][hc] -\)/\1 Single Processor -/
...
/GP2_M10/ s/\(size_t n\)//
/GP2_M11/ s/\[n\]//
s/\/\/GP2_M13//
...
```

sed script

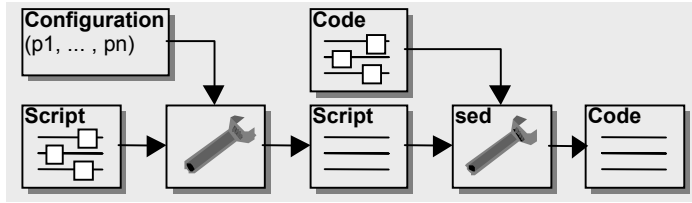


Fig. 6. A small excerpt from a generic dispatcher component together with a sed script allowing to tailor the source code.

string “size_t n“, and replaces it by an empty string. Variation points and the corresponding generic parameters are identified with the GPX_MXX markup. The tailoring description is specified externally in the sed script and is therefore associated via links – the markup in this case – with the variation points. It may be questioned, how the actual values of the generic parameters are considered in this approach. Apparently, these values must be propagated into the script. For this, a generic sed script is provided along with the generic component (cf. Fig. 6). In this script, placeholders for the generic parameters are identified. After the parameter assignment the generic parameters in the generic sed script are replaced by the actual values. The result of this replacement is a sed script that is finally used to tailor the component.

In the next example, a quite different approach is presented. Figure 7 shows an excerpt of a generic scheduler component. Java is used as a meta-programming language to implement the customization. The tailoring is described inline in the source code. All lines that start with a //. mark contain instructions of the Java meta-program. In lines 1 – 2, for instance, the generic parameters are declared, and lines 5-7 show an if-block. The meta-programming results in a source code description denoted as pseudo code that can be transformed through a simple generator into an executable Java class. Advantageous in this approach is that, although a kind of pre-processing is used to implement the tailoring, the complete expressiveness of a full-fledged programming language is available, and that finally an executable class is generated, which encompasses the generic component and automates the required tailoring. Any external generator can be called from within the pseudo code, and interactive support for substitution parameters becomes feasible. Conceptually, this approach is closely related with the work presented in [12] in the way that a general purpose programming language is used to automate variability within components. The both approaches differ in the scope of the meta programming, since we use this technique only to implement the generic components not to compose them. Our composition approach can be found in [3].

pseudo code

```
//. generic parameters:
//.            String schedStrategy;
[...]
SCHEDULER_C::~SCHEDULER_C() {
//.  if ( schedStrategy.equals("Prio") ) {
       delete [] TABLE;
//.  }
}
[...]
void SCHEDULER_C::ReadyToRun(THREAD_P pclThread){
  int _iPriority = pclThread->GetPriority();
//. if ( schedStrategy.equals("Prio") ) {
//.   includeFile("ReadyToRun_prio.cpp");
//. } else if ( schedStrategy.equals("EDFPrio")){
//.   iEntries++; }
```

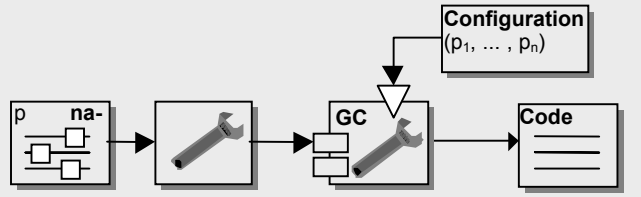


Fig. 7. Generic scheduler component that is transformed into an executable generic component automating its own configuration.

5 Conclusions

The presented concept of generic components aims at improving the reusability of prefabricated software components by explicitly planning for specific types of foreseeable adaptations, and by providing mechanisms and tools to perform such adaptations automatically. Generic components are implemented to be easily modified within prescribed bounds and to expose generic parameters, which provide the external handles for choosing and fine-tuning component properties. Tools like configuration management systems, pre-processors, or full-fledged code generators then allow to flexibly customize components for specific application environments with minimized manual intervention. Although the individual techniques used to implement and customize generic components are all well known, their explicit combination within the concept of generic components presents a new and promising way to pursue component reuse, even in the domain of embedded systems. With the ability to flexibly adapt the implementation of reused components, conventional approaches to improve component reusability, which typically interfere with the overall system performance, can be avoided. The concept of generic components approximates the advantages of manual customization and optimization, while on the other hand sidestepping the obstacles of time-consuming and error-prone coding by reusing prefabricated and tested implementations.

We have applied the presented techniques and the notion of generic components to build a number of reusable building blocks for embedded operating systems. As software reuse is especially beneficial – not to say: only works – within one type of architecture [10], these building blocks are designed for specific reference architectures, respectively. For a small operating system kernel for embedded systems, for instance, a total of 12 generic components provide the flexibility to generate more than a hundred different kernel variants covering a wide spectrum of possible requirements. Thanks to the fine-grained modifications that are applied to the generic components during instantiation, the resulting system properties – especially concerning critical nonfunctional aspects such as the timing behavior and the memory consumption – are comparable to manually tailored implementations. On the downside, however, the implementation effort for generic components is significantly higher than for conventional components. According to our experience, making component implementations generic approximately accounts for additional 50 to 100% of the development effort as compared to conventional implementations. And for most cases, this is just half the way: in order to further support the selection and configuration of components by tools as presented in chapter 3 or in [3], comprehensive component descriptions are required. Such descriptions roughly make up another 100% of the initial effort for conventional implementations, ultimately summing up to tripled development costs for generic components – an investment, after all, that is typically compensated by the increased probability of reuse. To further reduce this overhead, we are currently extending our research to the questions of how to specifically support the development of generic components. The possibilities range from ergonomic editors smartly handling the variant parts of the source code, to automated detection of component interdependencies and automatic deduction of properties of composed artifacts.

References

1. Baum, L., Geyer, L., Molter, G., Rothkugel, S., Sturm, P.: Architecture-centric Software Development Based on Extended Design Spaces, Proc. of the 2nd ARES Workshop (Esprit 20477), Las Palmas de Gran Canaria, February 1998
2. Baum, L.: Towards Generating Customized Run-time Platforms from Generic Components, Proc. of the 11th Conference on Advanced Information Systems Engineering (CAISE'99), 6th DC, Heidelberg, Germany, June 1999
3. Baum, L., Becker, M., Geyer, L., Molter, G.: Mapping Requirements to Reusable Components using Design Spaces, Proc. of the IEEE Int'l Conference on Requirements Engineering (ICRE-2000), June 19-23, Schaumburg/Chicago, USA, 2000
4. Baum, L., Becker, M., Geyer, L., Gilbert, A., Molter, G., Tamara, V.: Supporting Component-Based Software Development Using Domain Knowledge, 4th World Multiconference on Systemic, Cybernetics and Informatics (SCI 2000), 2000
5. Biggerstaff, T.: The Library Scaling Problem and the Limits of Concrete Component Reuse, Proc. of IEEE Int'l Conference on Software Reuse, November 1994
6. Coplien, J.O.: Multi-Paradigm Design for C++, Addison Wesley Publishing Company, 1998
7. Czarnecki, K., Eisenecker, U.: Components and Generative Programming, Software Engineering Notes, vol. 24, no. 6, 1999
8. Czarnecki, K., Eisenecker, U.: Generative Programming, Addison-Wesley, 2000
9. Dougherty, D., Robbins, A.: sed & awk, 2nd Edition, O' Reilly, 1997
10. Garlan, D., Allen, R., Ockerbloom, J.: Architectural Mismatch: Why Reuse Is So Hard, IEEE Software, 12(6), 1995
11. Jacobson, I., Griss, M., Jonsson P.: Software Reuse - Architecture, Process and Organization for Business Success, ACM Press/Addison-Wesley, 1997
12. Kamin, S., Callahan, M., Clausen, L.: Lightweight and Generative Components I: Source-level Components, 1st International Symposium on Generative and Component-Based Software Engineering (GCSE'99), 1999
13. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopez, C., Loingtier, J. M., Irwin, J.: Aspect-Oriented Programming, European Conference on Object-Oriented Programming (ECOOP), 1997
14. Lane, T. G.: Studying Software Architecture Through Design Spaces and Rules, Technical Report CMU/SEI-90-TR-18, Carnegie Mellon Univ., 1990
15. Lane, T. G.: Guidance for User-Interface Architectures, in: Garlan, D., Shaw, M.: Software Architecture – Perspectives on an Emerging Discipline, Prentice Hall, 1996
16. McIlroy, M.D.: Mass-produced software components, Software Engineering: Report on a Conference by the NATO Science Committee, (Naur, P., Randell, B., eds.). NATO Scientific Affairs Division, Brussels, 1976
17. Parnas, D.L.: On the Design and Development of Program Families, IEEE Transactions on Software Engineering, SE-2:1-9, March 1976
18. Schäfer, W., Prieto-Diaz, R., Matsumoto, M (eds.): Software Reusability, Ellis Horwood, New York, 1994
19. Veldhuizen, T.: Using C++ template metaprograms, C++ Report, vol. 7, no. 4, May 1995
20. Veldhuizen, T.: Template Metaprograms, http://www.cs.rpi.edu/~musser/ap/blitz/meta-art.html , 1998

Design and Implementation Constructs for the Development of Flexible, Component-Oriented Software Architectures

Michael Goedicke[1], Gustaf Neumann[2], and Uwe Zdun[1]

[1] Specification of Software Systems, University of Essen, Germany
{goedicke,uzdun}@cs.uni-essen.de
[2] Department of Information Systems, Vienna University of Economics, Austria
gustaf.neumann@wu-wien.ac.at

Abstract Component-orientation is an emerging paradigm that promises components that are usable as prefabricated black-boxes. But components have the problem that they should be changeable and flexibly adaptable to a huge number of different application contexts and to changing requirements. We will argue, that sole parameterization – as the key variation technique of components – is not suitable to cope with all required change scenarios. A proper integration with multiple other paradigms, such as object-orientation, the usage of a scripting language as a flexible component glue, and the exploitation of high-level interception techniques can make components be easier (ex)-changeable and adaptable. These techniques can be applied without interfering with the component's internals.

1 Introduction

The task of a software engineering project is to map a model of the real world (existing or invented) onto a computational system. The complexity and diversity of concrete real world systems can be overwhelming. This is no complexity in the algorithmic sense, but an complexity of an overwhelming amount of details and of particularities in the universe of discourse. By developing a model we reduce this complexity by finding and extracting commonalities. The key instruments of modeling are abstraction and partitioning. Analyses of commonalities let us understand the common elements of a targeted system. The aim of any analysis of commonalities is to group related members of a family, regardless whether the members are components, objects, modules, functions, etc.

Orthogonal to the task of modeling commonalities (where details are removed) is the task of engineering variability. It makes absolutely no sense to create abstractions to understand a family as a whole, if we do not introduce proper means for variation in the family members [3]. Finding commonalities in software eases understanding and reduces the need for changes, while finding proper variabilities enables us to use the software at all, because we have to re-adapt the found abstractions to the concreteness of the modeled real world

G. Butler and S. Jarzabek (Eds.): GCSE 2000, LNCS 2177, pp. 114–130, 2001.

situation. Commonality and variability are competing concerns and its hard to find a proper balance between them by approaches that (a) model the real world from the scratch and then (b) try to reuse the common aspects in such upfront design models. The forces in the steps (a) and (b) can normally not be well integrated. We rather propose in this work to model only the interfaces and keep them variable. Techniques for "programmable interfaces" let us flexibly glue the application parts together.

There are recurring ways for finding good abstractions and partitionings. "Good" means that they provide a tenable amount of commonalities to let us understand the problem and produce long-lasting software, but still enable us to easily introduce (expected and unexpected) changes. Such patterns of organizing abstraction around commonalities and variations are popularly called "paradigms". In software engineering a paradigm is a set of rules for abstraction, partitioning, and modeling of a system. E.g., the object-oriented paradigm structures the design/program around the data, but focuses on behavior [23]. It allows us to introduce variations in data structures/connections and algorithm details. Each paradigm has a key commonality and variation.

If we implement a system, we have to deal with a broad variety of paradigms. Coplien [3] discusses the need for multi-paradigms. In fact nearly any good real world software system is designed and implemented using multiple paradigms, simply because nearly no complex real situation exists, that can be described with one paradigm sufficiently. E.g., in nearly every large C++ program a mixture of object-oriented, procedural, template, and various outboard paradigms exists. Here, outboard paradigm [3] means a paradigm that is not supported by the programming language itself, but by a used technology, like the relational paradigm adopted from a relational database.

In the focus of this paper are language constructs and concepts for design and implementation that overcome current problems of the component- and object-oriented paradigms and their integration. Firstly, we will discuss these paradigms and their current integration problems. Afterwards we present some language concepts of the language XOTcl: Firstly we will discuss concepts which can be mapped manually to current mainstream languages, then we will present some interception techniques that are missing in current mainstream languages. Finally we will generalize our approach and compare to related work.

2 Combination of Component- and Object-Oriented Paradigm

2.1 Component-Oriented Paradigm

The very idea of component-based development is to increase productivity of building software systems, by assembling prefabricated, widely-used components. Components are self-contained, parameterizable building blocks with explicit interfaces. Component-based development aims at the replaceability of components and the transferability of components to a different context, thus enabling component reuse.

The idea of the component-oriented key abstraction is not new. E.g., in many large C systems self-contained components (or modules) that can be accessed via an explicit API can be found. Component-based development, as it is proposed today, mainly adds interface definition languages or other means to enforce that all component accesses conform to the component interface, platform and/or language independence, support for distribution, and accompanying services.

Current component approaches, such as component frameworks in scripting languages, like Tcl [19], or the component models of popular middleware approaches, such as CORBA, Java Beans, or DCOM, induce mainly a black-box component approach. Unfortunately often there are several factors for development organizations that drive them not to adopt the component-based approach for components developed by a third-party or even by a different in-house development team. Often used arguments are, that internals of a black-box component can not be changed, therefore, *reaction on business process changes* can become more difficult. Generally, *bugs in the component are harder to fix*, because it is hard to build a work-around for a bug, that is not located in your own code. The organization relies on the ability and will of the component developer to fix the bug. Moreover, using black-boxes often means that the *development team looses expertise on component's domain.*

These factors can be observed in many real world applications and they apply not only for black-boxes, but also (to a smaller degree) to components with available source code. The component abstraction seeks for building blocks that can be produced and maintained separately from the systems they are used in. Variations have to be treated separately and are mainly introduced by means of parameterization. The key problem of component-based software engineering is: On the one hand, components aim at extracting the commonalities to a level, where we can use them as prefabricated building blocks. On the other hand it is hard to maintain and to cope with changes in a piece of software without access to internals. Since parameters are the main variation technique of black-box components, the changeable parameters have to be foreseen by the component developer. But in reality often the requirement changes are not foreseeable at component development time.

2.2 Object-Orientation and Components

Object-orientation is a paradigm sharing properties with component-based approaches. Many component-based approaches are implemented using objects. Object-orientation arranges structures around the commonalities of data, but focuses on behavior. Traditional object-oriented programming and design maps entities of the modeled real world to a single programming language/design construct: the class. Object oriented design expresses computational artifacts through a mapping onto several classes and their relationships. In most object-oriented approaches, these relationships are association/aggregation (most often both are expressed through the same language construct), and inheritance or extension.

Object-orientation promises separation of the involved orthogonal concerns through encapsulation and class inheritance. The abstraction into general parts with inheritance or delegation in object-based system should help us to concentrate on common and special properties at different times. These abstractions also promise to gain modularity and to anticipate changes by designing general modules, that can be specialized in different ways and that support incremental extensions.

These promises were only partially achieved by traditional object-oriented approaches. Studies of the amount of reuse gained through object-orientation indicate that reuse is much smaller than expected in its early promises. E.g., Ousterhout [19] points out on basis of empirical evidence that the reuse of components – as they are used in scripting languages – is by far higher, than the reuse gained solely by object-orientation. We believe a main reason for this divergence is that large object-oriented frameworks tend to require an intimate knowledge of the framework's internals.

The component-oriented key abstraction of consequently exploiting parameterizable black-boxes is more suitable for reuse. But when taking a closer look at the success factors of scripting languages, one can observe that they combine the components with a highly-flexible glue language. Object-orientation especially helps us to understand structural complexity and to make it explicit by architectural means. These architectural means can be extremely valuable in complex design situation in the glue language (see [18]), because a relatively small glueing application can become very expressible and complex through the number of involved components. And object-oriented language constructs can also be valuable in the internal design of the self-contained components. Our approach relies on the two-level concept of scripting languages, like Tcl. We distinguish into reusable, self-contained components, mostly written in a system language, and a high-level, object-oriented scripting language that combines these components flexibly.

Complex design problems are a focus of object-oriented approaches, but a weak point of component combination with a scripting language, like Tcl. Object-oriented design patterns capture the practically successful solutions of the field of object-orientation. Our research on language support for design pattern, as for instance in [14], has shown that pattern variations cannot be described solely through parameterization. Reusable pattern implementation variants have to be fitted to the current context, especially when patterns are used in the hot spots [20] of software systems. These parts of the application – where an elegant and sufficient solution requires variabilities beyond pure parameterizations – are the parts that are hard to cover with the component-oriented abstraction, since its key variability is parameterization. The combination of the two paradigms with high-level design/implementation language functionalities lets the object-oriented constructs cover the weak points of the black-box component approach and vice versa.

3 Components and Component Configuration in XOTcl

Extended Object Tcl (XOTcl) [18] (pronounced *exotickle*) is an object-oriented extension of the language Tcl. Scripting languages gain flexibility through language support for dynamic extensibility, read/write introspection, and automatic type conversion. The inherent property of scripting languages such as Tcl is that they are designed as two-level languages, consisting of components written in efficient and statically typed languages, like C or C++, and of scripts for component glueing.

Our assumption is that just "glueing" is not enough. XOTcl enhances Tcl with language constructs giving architectural support, better implementation variants, and language support for design patterns, and explicit support for composition/decomposition. All object-oriented constructs are fully introspectable and all relationships are dynamically changeable. XOTcl offers a set of basic constructs, which are singular objects, classes, meta-classes, nested classes, and language support for dynamic aggregation structures. Furthermore, it offers two message interception techniques *per-object mixin* and *filter*, to support changes, adaptations, and decorations of message calls.

In XOTcl a component is seen as any assembly of several structures, like objects, classes, procedures, functions, etc., to a self-contained entity. Components are conveniently packed into packages that can be loaded dynamically. A component can also consist of a C or C++ extension of Tcl. Each component has to declare its name and optional version information with Tcl's `package provide` with the following syntax:

`package provide` *componentName ?version?*

The system automatically builds up a component database. With `package require` an XOTcl program can load a component dynamically with a name and optional version restrictions at arbitrary times. `package require` has nearly the same syntax:

`package require` *componentName ?version?*

Components expose an explicit interface that can be used by other programs without interfering with the components internals. But still we have to integrate the components with the application and make the component internals adaptable and dynamically fit-able to a changing application context.

3.1 Component Wrapping

Component wrappers can wrap black-box components written in various languages and structured with multiple paradigms. The component wrappers are object-oriented Wrapper Facades [21] that shield the components from direct access (see Figure 1). Note, that often a set of interacting component wrappers has to be used to wrap a complex component properly. Above the component wrapper layer a set of implementation objects define the hot spots of the design. All objects (including the component wrappers) can exploit the dynamic

and introspective language functionalities of XOTcl. Since a black-box component is never accessed directly, but always with the indirection of the component wrapper, we gain a central place, that is a proxy or placeholder for a component. The component wrapper is a white-box for the development team of the application. Here, changes can be applied centrally and adaptations can be introduced without affecting the components' internals.

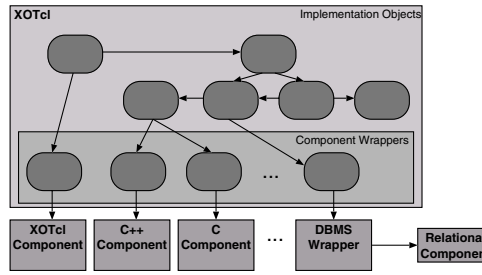


Figure 1. Integration of Components and Objects through Component Wrappers.

Generally each component wrapper is implemented with an abstract interface and a concrete component wrapper implementation (see Figure 2). Clients use

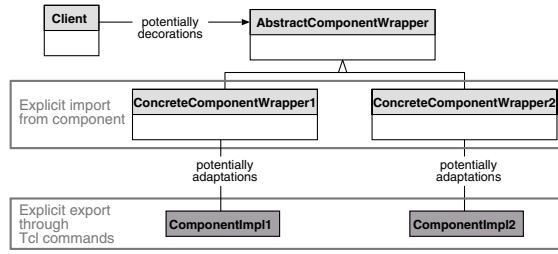


Figure 2. Class-Based Component Wrapper Interface.

the components as Strategies [4] to make components easily exchangeable by providing a new concrete component wrapper and by dynamically changing to a new Strategy. The concrete component wrappers forward the received messages as Wrapper Facades [21] to the components that implement the functionality. At the connection between client and component wrapper we can easily enhance the functionality of the component with Decorators [4]. At the connection between component wrapper and component we can use Adapters [4], e.g. to perform interface adaptations.

3.2 Export/Import Component Configuration

The implementation of the component's functionality (e.g. in C or C++) is integrated into XOTcl with Tcl commands (see Figure 3). A component explicitly defines its *export* by explicitly defining a set of Tcl commands (function names with argument lists). These commands can be mapped onto one (or more) wrapper objects, that configure the component usage and adapt the Tcl Commands to an object-oriented interface. The component wrapper explicitly declares which of the exported methods are the *import* of this component usage. This way the component's client defines the required interface that an implementing component has to conform to. The component implementation can be replaced by any other implementation that conforms to the required interface. Finally, the actual implementation objects, which are using the component, call the methods of the component wrapper.

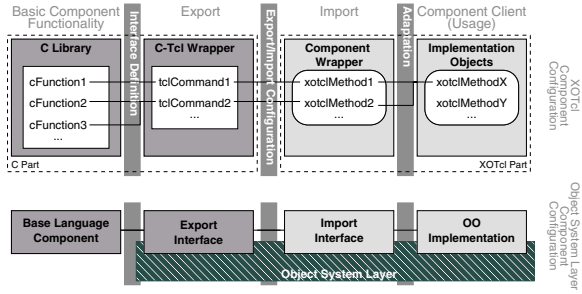


Figure 3. Three-Level Component Configuration with Explicit Export/Import.

The implementation objects can be used to build an application or a new component. If a new component is built from existing components, it can export an interface through a component wrapper consisting of XOTcl methods. But, since any Tcl program can be embedded in a C program, a new component can also export C functions (which can be used by any C program).

The component concept relies at runtime on the concept of *component configuration* [5]. The first configuration step maps a C library component with an interface design into the scripting language. Then this functionality is imported and adapted by the component wrapper. Finally, the implementation objects use the adapted import in their application framework. Each configuration step allows us to actualize the configurations with different implementations that conform to the interfaces. The integration of C components is presented in the upper half of Figure 3.

The general technique of applying an *Object System Layer* to a base language and to implement the components with an object-oriented implementation is presented below. This technique is used in various languages and applications and is documented as the *Object System Layer* architectural pattern [12].

Component configuration – as used in this work – is the runtime technique of combining components. In XOTcl each component configuration can be changed dynamically at arbitrary times. The component import interfaces can be dynamically fitted to the new context. In order to keep track with this runtime flexibility an important functionality of the XOTcl language is introspection. It allows us to query the import interface for method names, argument list, and method implementations. The currently configured components can be queried to trace the components, their configuration, and the used interfaces at runtime. Runtime inspection tools can be written with a few lines of code.

4 Interception Techniques for Flexible Component Wrapping

The techniques discussed so far can (mainly) be implemented in any object-oriented design/programming language. The only difference of using XOTcl is that XOTcl language supports the discussed component concept already, i.e., it offers dynamic package loading mechanisms, language support for dynamic aggregation, dynamics and introspection in all language constructs, etc. In other languages we have to program the concept implementations by hand. But implementing flexible component wrappers solely with class constructs has several disadvantages:

- *Transparency:* The client should use the abstract interface without knowledge of concrete implementation details. The component wrapper should not appear to be scattered over several implementation objects.
- *Concerns that cross-cut the component wrapper hierarchy*: Often a complex class hierarchy is necessary to implement component wrappers sufficiently. E.g., most widget sets offer widgets as C or C++ components. In order to compose compound widgets out of simpler widgets, we may need object hierarchies as in the Composite pattern [4]. Concerns that are of a broad structural size and that cross-cut the hierarchy, such as painting of the whole compound widget, conventionally have to be programmed by hand.
- *Object-specific component wrapper extensions or adaptations*: Often adaptations have not to be performed for all objects of a certain component wrapper type, but only for one object. We should be able to object-specifically enhance components, without sacrificing the transparency. The intrinsic component wrapper implementation and the implementation of extension/adaptation parts should remain decomposed.
- *Coupling of Component and Wrapper*: Component and component wrapper should appear as one runtime entity to clients, but they should be decomposed in the implementation.
- *Dynamics in Component Loading*: Components should be dynamically loadable, replaceable, and removable.
- *Runtime Traceability*: Components are loaded (possibly dynamically) into the system. To know which components are already loaded, the connections between wrapper and component should be traceable at runtime.

In this section we will briefly explain two interception techniques of XOTcl, that overcome these problems by flexible adaptation of the component wrapper calls to the concrete implementation. Both are transparent for the client. The per-object mixin implements concerns that are object-specific extensions, while the filter implements concerns that cross-cut class hierarchies. Filters and per-object mixins form runtime traceable entities with the intercepted objects at runtime, but are decomposed in the implementation.

4.1 Per-Object Mixins for Object-Specific Component Wrapper Extensions

A per-object mixin [13] is a language construct, that enhances a single object with a class that is object-specifically mixed into the precedence order of an object in front of the precedence order implied by the class hierarchy. Through object-specific, transparent, and dynamic interception of the messages that should reach the object, object-specific pattern variants [15] and object-specific roles [13] can be implemented conveniently. Per-object mixins allow us to handle orthogonal aspects not only through multiple inheritance, but since they are themselves classes and use class inheritance, they co-exist with the object's heritage order.

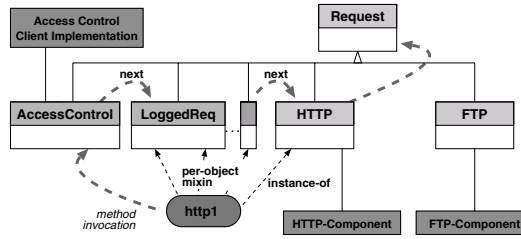


Figure 4. Request Logging/Access Control with Per-Object Mixins.

In Figure 4 we can see an example of a per-object mixin. An abstract class `Request` has two subclasses, one handling `HTTP` requests and one for `FTP` requests. The class definitions may look like:

```
Class Request                                ;# Abstract class definition
Request abstract instproc open {}            ;# Abstract method
...
Class HTTP -superclass Request               ;# HTTP class definition
HTTP instproc open {} {                      ;# Method definition
  ...                                        ;# Method forwards to HTTP
}                                               component
Class FTP -superclass Request                ;# FTP class definition
...
```

`HTTP` and `FTP` objects are Wrapper Facades [21] to components that implement the actual requests as black-boxes. Orthogonal to the tasks of a requests are the

tasks of request logging, which can operate on both mentioned request types. In many cases only certain specified request objects should be logged, as in the example `http1`.

We do not want to interfere with the internals of the components that implement the requests in order to gain request logging. Therefore, a solution with single or multiple inheritance would not suffice, because it would either make all requests logged or create unnecessary intersection classes [13], like `LoggedHttpRequest` and `LoggedFtpRequest`. A solution with a reference from a logging object, as in the Decorator pattern [4], would require the client to maintain a reference to the perhaps volatile logging object and, therefore, it would be not transparent to the client. A solution with a reference to a logging object, as in the Strategy pattern [4], would not be transparent to the request object and unnecessarily interfere with the internals of the component wrapper. Both solutions suffer from the fact that – from the viewpoint of the client – one conceptual entity is split up into two runtime entities.

The solution with the per-object mixin, as in Figure 4, does not suffer from any of these problems. It attaches the role of being a logged request and an access control mechanism as a second orthogonal aspect object-specifically to the request object, either in Decorator or Strategy style (as required). The access control mechanism is actually performed in an imported component, while the rather simple task of logging is handled by the mixin class. The per-object mixin is transparent to client and request object. The logged request appears as one conceptual entity to the client. There is only one object `http1` that can be accessed and it always has the same intrinsic class `HTTP`. But still logging and request tasks are decomposed into different classes and can be dynamically connected/disconnected. Per-object mixins can be attached in chains and specialized through inheritance. The per-object mixin solution may look like:

```
HTTP http1                                ;# Instantiation of http1 object

Class LoggedReq                           ;# Logged request class definition
LoggedReq instproc open {} {
  # logging implementation
  next
}
...
Class AccessControl
...
http1 mixin {AccessControl LoggedReq} ;# Mixin registration
```

`LoggedReq` and `AccessControl` are ordinary classes of XOTcl. As an example method, we define a method `open` for `LoggedReq` that logs all `open` calls and forwards the message afterwards with the `next` language primitive to the next mixin or the actual method implementation. We dynamically register the mixin classes for `http1`.

4.2 Filters for Cross-Cutting of Class Hierarchies

A second interception technique, called filter [14], is able to operate on a class hierarchy, instead of a single object (as the per-object mixins). A filter is a special instance method registered for a class C. Every time an object of class C or one of its sub-classes receives a message, the filter is invoked automatically. A prominent concern for usage of filters is to implement larger artifacts of the modeled world, like object-oriented design patterns, as instantiable entities of the programming language. In [14] we show how to express such concerns through a meta-class with a filter. The filter operates on all classes derived from the meta-class. Filters can be defined once and can then be registered dynamically. One can introspect the filters that are registered for a class. All filter invocations are transparent for clients.

As an example for filters, we present the implementation of a Composite pattern [4] variant in a reusable component. The context of the composite pattern is to build up an object tree structure and to derive a set of classes from an abstract component type. E.g., a composite widget component `Canvas` can aggregate other widgets. This way we can build up compound widgets. All widgets confirm to the same abstract widget component interface. A recurring problem of such structures is that leaf object, like button widgets, are not allowed to aggregate other objects. The solution to the problem in a purely class-based environment is, that only aggregating composite objects contain an aggregation relationship. Leaf objects, like buttons in the widget component example, have no children. An intrinsic property of composite hierarchies is that certain operations on the root component, as in the widget example a painting of the compound widget, have to be propagated through the object tree. Composite objects forward these operations to all their children, while leaf objects only execute the operation, but do not forward.

There are several problems that can be identified with the class-based implementation of the pattern. Concerns that cross-cut the composite hierarchy, like the forwarding of messages or the life-time responsibility of the whole for its parts (in the widget example: if a top-level widget is destroyed, all the constituent components have to be destroyed), are not expressed properly, since their semantics are not handled automatically. There is a certain implementation overhead, e.g. due to unnecessary forwarding of messages. The pattern as a conceptual entity of our design is neither traceable in the program code nor at runtime, but it is scattered across several implementation constructs. This variant of pattern implementation can hardly be reused, when the implementation language lacks proper (dynamic) means to fit a general variant implementation to a new context.

The implementation with a filter, a meta-class, and dynamic object aggregation, as in Figure 5, does not suffer from these problems. The filter is stored in a dynamically loadable component that contains a meta-class with a filter. The filter implements the reusable pattern implementation variant. Classes, like the widget component, that are superclasses of composite types are of the `Composite` meta-class type. Automatically they get the filter registered. The filter acts on

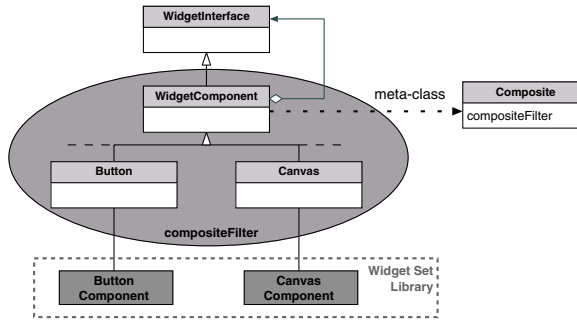


Figure 5. Composite Pattern with Filters/Dynamic Object Aggregations.

the whole composite hierarchy and implements the concerns that cross-cut the hierarchy. Here, the forwarding of composite messages on a set of registered operations, like `paint`, is such a concern. The filter in the meta-class is transparent to clients. The compound widget appears as every other ordinary widget. The dynamic object aggregation language construct automatically handles the aggregation issues, like assurance of the tree structure and the life time responsibility of wholes for their parts.

The class definition of the `Composite` meta-class firstly has to define the meta-class. The composite filter is an ordinary instance method that determines the called method with an introspection option. It calls the message on all children with a loop and then on the current object. The component may look like:

```
package provide Composite 0.8
...
Class Composite -superclass Class          ;# Meta-class definition
Composite instproc compositeFilter args {  ;# Composite filter method
  ...
  set r [[self] info calledproc]           ;# Determine called operation
  foreach child [[self] info children] {   ;# Loop over all children
                                              objects
    eval $child $r $args                   ;# Forward message to children
  }
  return [next]                            ;# Call message on
                                              'self'-object
}
```

We can load the Composite implementation from a component with `package require`. Afterwards we define the class hierarchy. `WidgetComponent` is defined by the meta-class `Composite` and automatically handles the forwarding of messages for all sub-classes transparently.

```
package require Composite
...
Composite WidgetComponent -superclass WidgetInterface
Class Button -superclass WidgetComponent
Class Canvas -superclass WidgetComponent
```

5 Components for Development of Flexible Software Architectures

The outcome of software development is a piece of software, a sustainable intellectual structure, which manifests the results of the design. Often it is handed over from the development team to the maintainers. The ability to modify or to understand the software especially for a person that was not involved in the development details, depends on the software architecture.

There are several properties that we expect from a "good" software architecture: It should be flexible, evolvable, understandable, predictable and maintainable. We expect the architectures to offer a significant amount of code reuse to speed up the development process and to achieve more reliable software systems. Certainly, the systems should be highly efficient. In summary we can make the following assessments on the architectural impact of the approach discussed in this paper:

- *Heterogeneous, Multi-Paradigm Black-Box Components*: Black-box components from various languages, like Tcl, XOTcl, C, or C++, are reused. The components are implemented with the most suitable paradigms. Unfortunately flexibility and evolvability of our software architectures suffer from our inability to change or react on problems of the components internals. In this paper we have discussed two approaches – scripting and high-level object-orientation – to overcome these problems.
- *Object-Oriented Scripting Language as a Component Glue*: Scripting languages combine components flexibly, by means of a highly flexible, introspective, and dynamic glueing language. But scripting languages are not very suitable to express the complexity of large application frameworks (see [18,14]). Their original language design aims at smaller applications, partly because of runtime efficiency. However, time critical parts can always be put into components written in more efficient languages, like C. Still complex scripting applications, like several compound widgets in TK, were very slow in the early days of Tcl/TK. But nowadays CPU speed allows us to build very complex scripting applications without a reasonable speed penalty.
  In contrast, the combination with object-orientation gives us architectural support for composition/decomposition. The hot spots of the application, which are expectable changing parts, are kept in the high-level object-oriented language with its dynamic and introspective language means. Design experience of the object-oriented community with complexity of applications and with introducing flexibility in framework hot spots, helps us to make the component wrappers easily (ex-)changeable and evolvable.
- *Component Wrappers*: Object-oriented component wrappers integrate the components into the scripting language (if necessary). With the set of component wrappers a component's client explicitly defines its import from the component. In turn, the component explicitly defines its export through the Tcl wrapper. In a three-level process of configuration we actualize the interfaces with concrete implementations. These can be exchanged against other

implementations transparently, what leverages evolvability and flexibility of our architectures. The implied indirection fosters understandability, since we can understand components and clients independently. The component wrappers are introspectable white-boxes to the application. Often changes in the component wrapper – without interference with the component's internals – are sufficient to cope with new requirements for a component.
- *Interception Techniques*: Object-oriented interception techniques, like filters and per-object mixins, enable us to deal with concerns that are hard to express with current object-oriented constructs. They are especially valuable on the component wrapper, since they allow us to transparently and dynamically introduce multiple views onto a component implementation (at runtime).

To back up the results in this work we have provided two case studies of systems build with the techniques described in this paper. In [16] we describe our web server implementation in XOTcl. We have compared efficiency with the pure C-based Apache web server. In the worst case our implementation was 25% slower than Apache, in some cases, our implementation was faster. In [17] we present a high-level framework for XML/RDF text parsing and interpretation. Again we have performed speed comparisons with implementations in Java and C. The Java based implementation was 2-4 times slower than our implementation in the scripting language (using off-the shelf C components), the pure C implementation was only 1.5-3.5 times faster than the scripting implementation.

6 Related Work

In [22] the (mainly black-box) component models of current standards, like CORBA, COM, or Java Beans are discussed. These approaches offer the benefits of black-box component reuse, but have problems, when the internals of a component have to be changed or adapted. Currently none offers an integrated concept for component reuse and adaptation, that solves all the problems raised in this paper, but all approaches have extension in this direction.

Java Beans implement limited dynamic loading and reflection abilities, but not all interesting data can be introspected (e.g., the important information on caller/callee of a call can not be retrieved automatically). Java Beans offer a distinct component model. But Java Beans offer only the Java Native Interface for integration of components written in other programming languages. Java does not support powerful language means for configuration/adaptation of components.

In [7] an interception system for COM objects that can intercept object instantiations and inter-object calls is presented. COM is a binary object model and the approach relies on direct manipulation of the function pointers that call the methods. In contrast to the approaches in this paper the approach is a very low-level approach and does not support suitable introspection mechanisms.

Orbix filters [8] (and similar techniques in other ORBs) implement limited interception abilities. They do only operate on distributed method calls and do

not offer sophisticated introspection techniques. A more general form of such abstractions of the message passing mechanisms in distributed systems are composition filters [1]. Abstract communication types are used as first-class objects that represent abstractions over the interaction of objects. They encapsulate and enforce invariant behavior in object communications, can achieve the reduction of complexity of object interactions, and can achieve reuseability of object interaction forms.

The new CORBA 3.0 standard specifies a component model that is based in part on the Java EJB component concepts, but goes beyond that, by providing the component model to work with various languages. The new CORBA standard includes a scripting language specification, which is a framework to integrate scripting languages with a CORBA mapping. Interestingly, the goals for primary applications of the scripting language specification are the same issues, for which we have given architectural language support in this work, i.e., customizations of components, legacy wrapping, a service-based callback architecture, and flexible component glueing. It is likely that XOTcl and the techniques discussed in this work would ease it to reach these goals (though the specification is still in an early state).

A role, as in [11], is used to express the combination with extrinsic properties that can be dynamically taken/abandoned. The approach does not define a comprehensive read/write introspection mechanism. It does not provide an abstraction at a broader structural size, as the filter that applies a role on a whole class hierarchy.

Aspect-oriented programming [10] is a programming technique for decomposing concerns into aspects, that have to be coordinated with other concerns across component boundaries. Aspects *cross-cut* component boundaries, while components are characterized by being cleanly encapsulated. An "aspect weaver" (a kind of a compiler) weaves components and aspects together. The approach does only introduce limited (dynamic) changeability through re-weaving, but it can express concerns that cross-cut several components properly. In contrast to our approach, aspect-orientation is not inherently a black-box approach, but requires knowledge of the component's internals to build useful aspects.

Meta-object protocols, as in [9], divide a system into meta-level and base-level. Meta-level objects impose behavior over base-level objects. Generally meta-object protocols are low-level, but powerful; they can achieve reflection, dynamics, and transparency. Our approach provides meta-classes with similar abilities, but most often the interceptors are more powerful, easier composable, and provide more introspection facilities.

Bosch proposes in [2] a component adaption technique based on layers, which is similar to the presented interceptors: it is also transparent, composable and reusable, but it is not introspective, not dynamic and a pure black-box approach. Layers are given in delegating compiler objects, that are statically defined before compilation time. This approach can be hardly used for expressing runtime dynamics in component composition, since changes in layer definitions require recompilations.

Subject-oriented programming models [6] offer different views for a client onto a concern. Classes can be composed with composition rules. In contrast to composition with interceptors, it does not provide introspective and dynamical runtime composition, but only by a tool called subject compositor. Subject-orientation (and subsequent approaches) address composition of system parts with extensions and multiple views, thus it helps to overcome some of the problems of (descriptive) component composition. Central runtime problems of combining components, like adapting components without interference with component internals at different granularities, component introspection/runtime traceability, or dynamic component loading/unloading are nearly not addressed.

7 Conclusion

We have addressed the issue that software architectures should be build with reusable (off-the-shelf) black-box components, but should also exploit the characteristics of white-box object-oriented frameworks regarding evolvability and flexibility. We have presented a practical approach that integrates such black-boxes, written in several languages. An object-oriented scripting language serves as a component glue with explicit export/import interfaces and as a central place to introduce changes into the hot spots of the architecture. To overcome several problems of object-orientation, the language offers interception techniques, that are valuable for component composition and adaptation. All presented techniques can be used in various other languages through explicit programming by hand. Nevertheless, (runtime) language support for introspection, dynamics, component composition/decomposition, and interception techniques is useful, since it leads to shorter, more elegant, and less error-prone solutions. Since XOTcl is itself a C library it can be embedded in any C or C++ application as a distinct component glueing language.
XOTcl can be downloaded from `http://www.xotcl.org`.

References

1. M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Object-Based Distributed Processing*, pages 152–184. LCNS 791, Springer-Verlag, 1993.
2. J. Bosch. Superimposition: A component adaptation technique. *Information and Software Technology*, 41, 1999.
3. J. O. Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley, 1998.
4. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
5. M. Goedicke, J. Cramer, W. Fey, and M. Groše-Rhode. Towards a formally based component description language a foundation for reuse. *Structured Programming*, 12(2), 1991.
6. W. Harrison and H. Ossher. Subject-oriented programming - a critique of pure objects. In *Proceedings of Conference on Object-Oriented Programming Systems, Languages (OOPSLA)*, 1993.

7. G. C. Hunt and M. L. Scott. Intercepting and insturmenting COM applications. In *Proceedings of COOTS'99, 5th Conference on Object-Oriented Technologies and Systems*, San Diego, California, USA, May 1999.
8. IONA Technologies Ltd. The orbix architecture, August 1993.
9. G. Kiczales, J. des Rivieres, and D.G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
10. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of ECOOP'97*, Finnland, June 1997. LCNS 1241, Springer-Verlag.
11. B. B. Kristensen and K. Østerbye. Roles: Conceptual abstraction theory & practical language issues. *Theory and Practice of Object Systems*, 2:143–160, 1996.
12. M.Goedicke, G. Neumann, and U. Zdun. Object system layer. Accepted for publication in EuroPlop 2000, 2000.
13. G. Neumann and U. Zdun. Enhancing object-based system composition through per-object mixins. In *Proceedings of Asia-Pacific Software Engineering Conference (APSEC)*, Takamatsu, Japan, December 1999.
14. G. Neumann and U. Zdun. Filters as a language support for design patterns in object-oriented scripting languages. In *Proceedings of COOTS'99, 5th Conference on Object-Oriented Technologies and Systems*, San Diego, California, USA, May 1999.
15. G. Neumann and U. Zdun. Implementing object-specific design patterns using per-object mixins. In *Proceedings of NOSA'99, Second Nordic Workshop on Software Architecture*, Ronneby, Sweden, August 1999.
16. G. Neumann and U. Zdun. High-level design and architecture of an http-based infrastructure for web applications. Accepted for publication in the World Wide Web Journal 3(1), 2000.
17. G. Neumann and U. Zdun. Highly flexible design and implementation of an xml and rdf parser/interpreter. to appear, 2000.
18. G. Neumann and U. Zdun. XOTcl, an object-oriented scripting language. In *Proceedings of Tcl2k: The 7th USENIX Tcl/Tk Conference*, Austin, Texas, USA, February 2000.
19. J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31, March 1998.
20. W. Pree. *Design Patterns for Object-Oriented Software Development*. ACM Press Books. Addison-Wesley, 1995.
21. D. C. Schmidt. Wrapper facade: A structural pattern for encapsulating functions within classes. *C++ Report, SIGS*, 11(2), February 1999.
22. C. Szyperski. *Component Software – Beyond Object-Oriented Programming*. ACM Press Books. Addison-Wesley, 1997.
23. P. Wegner. Learning the language. *Byte*, 14:245–253, March 1989.

Scenario-Based Analysis of Component Compositions

Hans de Bruin

Vrije Universiteit
Faculty of Sciences
Mathematics and Computer Science Department
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
hansdb@cs.vu.nl

Abstract. The behavior of a system comprised of collaborating components tend to be difficult to analyze, especially if the system consists of a large number of concurrently operating components. We propose a scenario-based approach for analyzing component compositions that is based on Use Case Maps (UCMs), but is extended with a few additional constructs for modeling component interfaces and connections. UCMs provide a high level, behavioral view on a system that is easy to comprehend by humans. However, UCMs do not have well-defined semantics. For this reason, UCMs are augmented with formal component interface specifications as used in the concurrent, object-oriented programming language BCOOPL. The combination of UCMs and BCOOPL interface specifications enables formal analysis of component compositions. This involves two steps. In the first step, UCMs and BCOOPL interface specifications are translated into a BCOOPL program. In the second step, the interactions between components are analyzed for system properties like deadlock and reachability. An important result of the combination is that the complexity, which arises when concurrently collaborating components are brought together, is tamed by considering only those usages of components that are actually specified in UCM scenarios.

1 Introduction

A high level design notation targeted to model component compositions should serve two purposes. On the one hand, the notation should be easy to understand by humans, in particular, designers. A designer should gain insight in the behavioral aspects of the system as a whole almost effortlessly. On the other hand, the notation should be precise enough to reason about compositions.

The behavior of a system can be defined with formalisms like temporal logics and traces of externally observable events, which can be represented formally and graphically with notations such as Petri nets [14] and StateCharts [12]. With suitable abstraction mechanisms we can then define the components and events of interest to gain insight in the overall behavior. However, this approach does not give the big picture. What is needed is a notation with which designers can reason about the behavior at a high abstraction level. This purpose can be fulfilled with a scenario-based technique called Use Case Maps (UCMs) [2] as will be shown in a couple of examples. One of the strong points of UCMs is that they can show multiple scenarios in one diagram and the interactions amongst them. This allows a designer to reason about a system as a whole instead of focusing on details.

G. Butler and S. Jarzabek (Eds.): GCSE 2000, LNCS 2177, pp. 129-146, 2001.

Real-size systems tend to be hard to analyze. Especially if a system is composed of concurrently running components, the number of possibilities to consider explodes exponentially. The complexity, which is caused by non-deterministic behavior, can be tamed by considering only those scenarios that can really happen. In particular, a component may provide a number of scenarios (behaviors) of which only a subset is actually used in a system. Thus, the scenarios provided by UCMs gives us a handle to prune all non-occurring behaviors of components beforehand.

UCM is an informal notation. This apparent shortcoming is precisely the reason why UCM can be used at high abstraction levels; low-level details are simply not part of the notation. By augmenting UCM with component interface specifications, it is possible to reason about component compositions formally. For instance, we will show in an example how the presence or absence of deadlock in a system can be detected. Another application area is to prove reachability properties, i.e., to verify whether a system can achieve certain goals or not.

In this paper, we combine UCMs with BCOOPL (Basic Concurrent Object-Oriented Programming Language) interface specifications, which not only detail how operations should be invoked, but also when operations may be invoked and the parties that are allowed to do so [8]. The temporal orderings of operation invocations provide a good starting point for analyzing whether a system comprised of components violates the imposed orderings or not. Apart from the aspects mentioned above, BCOOPL supports other language features that are useful for component-oriented programming, such as the built-in support for the Observer design pattern for minimizing the coupling between components, delegation for supporting black-box composition, concurrency for dealing with distributed computing transparently, and component access control for addressing security issues. These language features allow components to be specified at a higher abstraction level with respect to general purpose languages like C++ and Java. In fact, BCOOPL can be regarded as a design language with which executable designs can be specificed. The combination of UCMs and BCOOPL interface specifications can be translated unambiguously into a BCOOPL program. This BCOOPL program can then be subjected to analysis in order to prove certain system properties such as deadlock and reachability.

This paper is organized as follows. We start with an overview of UCM and BCOOPL. Next we introduce a running example to exemplify the two steps involved in analyzing a system. This is followed by a detailed explanation of the two steps: translating a specification into a BCOOPL program and analyzing the resulting program for system properties. We end this paper with a discussion and concluding remarks.

2 Use Case Maps and BCOOPL Preliminaries

The material in this paper builds on previous work in which a grey-box approach to component specification is argued [6]. In principle, a black-box approach to component deployment should be favored. In practice, however, we require information that cannot be described solely in terms of externally visible properties of components. For instance, non-functional properties (e.g., space and time requirements), environmental dependencies, and variation points (e.g., places where a component may be adapted or extended) do require insight in the internal construction of a component. The combina-

tion of UCMs and BCOOPL interfaces gives us the opportunity to document intra and inter component behavior at a high, but formal abstraction level.

2.1 Use Case Maps

A UCM is a visual notation for humans to use to understand the behavior of a system at a high level of abstraction. It is a scenario-based approach showing cause-effects by traveling over paths through a system. UCMs do not have clearly defined semantics, their strong point is to show how things work globally.

The basic UCM notation is very simple. It is comprised of three basic elements: responsibilities, paths and components. A simple UCM exemplifying the basic elements is shown in Figure 1. A path is executed as a result of the receipt of an external stimulus. Imagine that an execution pointer is now placed on the start position. Next, the pointer is moved along the path thereby entering and leaving components, and touching responsibility points. A responsibility point represents a place where the state of a system is affected or interrogated. The effect of touching a responsibility point is not defined since the concept of state is not part of UCM. Typically, the effects are described in natural language. Finally, the end position is reached and the pointer is removed from the diagram. A UCM is concurrency neutral, that is, a UCM does not prescribe the number of threads associated with a path. By the same token, nothing is said about the transfer of control or data when a pointer leaves one component and (re-)enters another one. The only thing that is guaranteed is the causal ordering of executing responsibility points along a path. However, this is not necessarily a temporal ordering, the execution of a responsibility point may overlap with the execution of subsequent responsibility points.

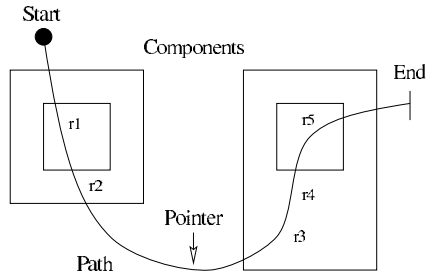


Fig. 1. UCM basic elements.

The UCM notation is quite rich, and only a small subset is used in this paper. In Figure 2, two frequently used UCMs constructs are shown. The AND-construct is used to spawn multiple activities along parallel paths. When a pointer reaches an AND-fork, this pointer is removed from the diagram and replaced by two pointers at the beginning of the parallel paths. An AND-join acts as a synchronization mechanism. When both pointers have reached the AND-join, they are replaced by a single pointer and execution is continued along the path following the join. The OR-construct should be interpreted

as a means to express multiple scenarios in a single diagram. It states that multiple scenarios are comprised of identical paths. Therefore, it is not necessary to specify conditions detailing the path to be followed at an OR-fork.

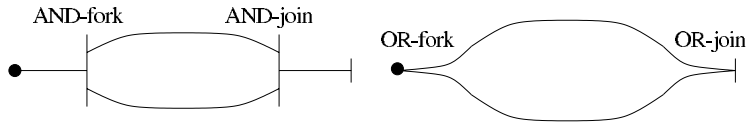


Fig. 2. Additional UCM notation.

2.2 BCOOPL

BCOOPL is a small, concurrent object-oriented programming language specifically designed to support component-oriented programming. BCOOPL has a long research history. It roots can be traced back to path expressions [4], and the concurrent object-oriented programming languages Procol [15] and Talktalk [1]. One of the strong points of BCOOPL is the built-in support of design patterns catering for component-oriented programming (see [10] for a pattern catalog). In particular, BCOOPL supports the Observer, the Mediator and the Bridge design patterns directly. The Observer and Bridge pattern are particular useful for specifying stand-alone components, while the Mediator pattern is used to mediate interactions between those components [7]. Other design patterns frequently used in components, such as the Facade and the Proxy, can be implemented relatively easily in comparison with more traditional object-oriented programming languages like Java and C++. A detailed account of BCOOPL and some of its application areas can be found in [8], which also addresses implementation issues.

2.2.1 Core Language Features BCOOPL is centered around two concepts: interfaces and patterns. An interface defines the operations that must be implemented by an object that conforms to that interface. By adhering to the principle of programming to an interface, a certain amount a flexibility is added to a system since new implementations can be provided without breaking existing code. A BCOOPL interface is specified as an augmented regular expression over operations. It not only describes how an operation can be invoked, but also when and by whom.

Class and method definition have been unified in patterns and sub-patterns. The term pattern has been borrowed from the object-oriented programming language Beta [13]. The idea is that objects are instantiated from patterns and behave according to the pattern definition. A pattern describes the allowed sequences of primitives to be executed by an object after a message has been received in a so called *inlet*, which is implicitly defined in a pattern definition. It is specified as a regular expression over primitives using the same operators as in interface specifications. A pattern may contain sub-patterns which also define inlets, and so on. A top-level pattern can be seen as a class definition, whereas sub-patterns can be seen as (sub-)method definitions.

A notification pattern is part of a pattern definition. It specifies the output behavior of a pattern in terms of notifications. An object interested in a particular notification of a publishing object can subscribe to that notification. The subscription information is comprised of, amongst others, the name of the notification, the identity of the subscriber and the pattern to be invoked in the subscriber. Notifications are issued through an *outlet* by means of a *bang-bang* (!!) primitive. As a matter of fact, notifications are not only used for implementing the Observer design pattern, but they are also used for getting a reply value as a result of sending a request to some object. The basic idea is to send a message to an object and then wait for a notification to be received in an inlet following the send primitive. The concept of notification patterns has been explored in Talktalk [1].

The type or types of a pattern are provided by interfaces. A pattern that implements an interface has the type of that interface. As in Java, multiple interface inheritance is supported in BCOOPL. That is, an interface may extend one or more sub-interfaces. In contrast to Java, BCOOPL interfaces contain sequence information.

2.2.2 Computational Model The computational model of BCOOPL is based on message passing and concurrent objects. Objects are instantiated from patterns by executing the *new* primitive. A pattern may contain sub-patterns and their corresponding objects are instantiated implicitly whenever a super-pattern is instantiated. A conceptual model of an object and its sub-objects is shown in Figure 3. Each object has an unique I.D., which is used as an address for message exchange. Objects communicate with other objects by means of a restricted from of asynchronous message passing in the sense that the partial ordering of messages sent from one object to another is preserved. An object receiving a message does not process the message right away, instead the message is placed in an unbounded message buffer. The dispatcher searches the message buffer on a *first-come-first-served* basis of acceptable messages. The acceptability of a message is determined by the state of patterns in execution. If an acceptable message is found, the dispatcher passes the message on to the corresponding (sub-)object's inlet, otherwise it waits for the arrival of new messages. Thus, the communications between objects can be summarized as synchronous message buffering, but asynchronous message processing.

A Computation in BCOOPL is achieved by sending messages. This includes control flow structures like selection (*if-then-else*) and repetition (*while-do*). A computation proceeds as follows. After a message has been received in an inlet, the sequence of primitives following the inlet are executed until one or more inlets (sub-patterns) are encountered. Regular expression operators, such as the selection (+) and the repetition (*), imply choices. Each branch resulting from such a choice must be guarded with an inlet. That is, the choice to follow a particular branch is made by sending an appropriate message. There is no such concept as non-deterministic choices.

Within an object and its sub-objects the *one-at-a-time* principle of executing primitives applies. Multiple execution threads may occur within a tree of (sub-)objects, which are introduced with the interleave operator (||). At most one thread, however, is active at any one time. Thread switching occurs at the time the active thread runs into one or more sub-patterns. Because almost every computation step is expressed in terms of message passing, thread switches occur frequently, which amounts to a semi-parallel object model. The next active thread is selected on the basis of the acceptability of the pending

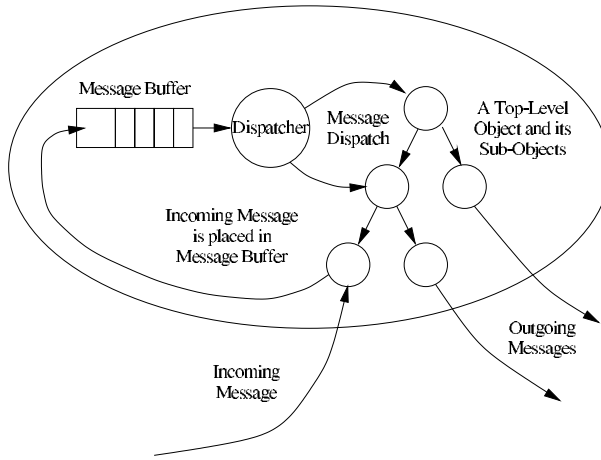


Fig. 3. Object model.

messages in the message buffer. This is a fair scheduling mechanism since the active thread cannot interfere with the scheduling of other threads, unless it claims explicitly the exclusive ownership of the object by means of the synchronize operator ($\ll expr \gg$). In contrast to intra-object concurrency, top-level objects (and their sub-objects) operate on a truly concurrent basis.

2.2.3 Interface and Pattern Specification An interface is identified by a name and may extend one or more base interfaces. It is defined by means of an *interface interaction term.*

interface *Interface Name*
  extends [*interfaces*]$_{opt}$
defines [
  interface interaction term
]$_{opt}$

An interface interaction term is specified using the following syntax:

client specifications $\mapsto$ *Pattern Name* (*input args*) $\Rightarrow$ (*notification pattern*) [
  regular expression over interface interaction terms
]$_{opt}$

An interface interaction term corresponds with a (sub-)pattern definition that implements the interface. It defines the pattern name, the formal input arguments, a notification pattern that specifies sequences of notification messages, and client specifications. An interface interaction term is recursively defined as a regular expression over interface interaction terms leading to a hierarchical interface specification. The regular expression operators used for constructing an interface and their meaning are summarized in Table 1.

Client specifications denote the parties that are allowed to invoke the corresponding pattern. They are defined by any combination of the following: by interface name, by

Expression	Operator	Meaning
$\ll$ E $\gg$	synchronize	E is executed uninterupted
E || F	interleave	E and F may occur interleaved
E + F	selection	E or F can be selected
E ; F	sequence	E is followed by F
E $*$	repetition	Zero or more times E
E $[m, n]$	bounded rep.	i times E with $m \leq i \leq n$

Table 1. Semantics of regular expression operators.

interface name set, or by object reference set The sets are used to dynamically specify the clients that are allowed to interact. A pattern implementing such an interface is responsible for the contents of a particular set.

Notifications issued by a pattern are guaranteed to be emitted according to the defined sequences specified in its notification pattern. A notification pattern is defined as a regular expression over notification terms that are specified as follows:

Notification Name (output args)

The co- and contra-variance rules apply for specifying interfaces. An interface interaction term may be redefined in a derived interface. The types of the input arguments must be the same as or generalized from the argument types of the base interface (i.e., contra-variance rule). In contrast, a notification pattern may be extended in a derived interface, both in terms of notification output arguments having derived interfaces (i.e., co-variance rule) and additional notifications.

The interface *Any* acts as a base type for every other interface. That is, every interface extends *Any* implicitly. *Any* is defined as:

interface Any

As an example of interface specification, consider the interface for a scarce resource. A resource must be acquired before it can be used, and after it has been used, it must be relinquished to allow other objects to use it again. The interface for a resource could read as follows:

```
interface Resource defines [
    Any ↦ () ⇒ () [
        (
            Any ↦ acquire () ⇒ (done()) ;
            Any ↦ use (in : SomeInArg) ⇒ (result(out : SomeOutArg)) * ;
            Any ↦ relinquish () ⇒ (done())
        ) *
    ]
]
```

This interface enforces cycles of acquiring, using, and relinquishing the resource.

Patterns and sub-patterns are defined identically. The overall structure of a pattern is the following:

```
client/server specifications ↦ pattern Pattern Name (input args) ⇒ (notification pattern)
    implements [interfaces]opt
    declares [local variables]opt
does [
    pattern implementation; a regular expression over primitives and sub-patterns
]
```

Client/server specifications are defined in patterns similar to client specifications in interfaces. Client specifications identify the kind of objects that may communicate with a pattern, whereas server specifications denote declaratively specified linkages to notifications. The syntax for server specifications is as follows.

Local Variable.Pattern Name$_1$*.Pattern Name*$_2$*.· · · .Pattern Name*$_n$*.Notification Name(formal arguments)*
Object Set.Pattern Name$_1$*.Pattern Name*$_2$*.· · · .Pattern Name*$_n$*.Notification Name(formal arguments)*

Notification linkages are established at run-time. An assignment to a variable involved in notification linkage results in first abolishing the current link, provided the variable is not *nil*, then assigning to the variable, and finally establishing a new link if the variable does not equal *nil*. In the case of an object set, a notification link is established when an object is added to the set, likewise it is abolished when the object is removed from the set.

The behavior of a pattern is defined in the *does* section. It is defined as a regular expression over primitives and sub-patterns. The supported primitives are summarized in Table 2. As an example, an implementation of the *Resource* is given below.

```
Any ↦ pattern resource () ⇒ () implements [ Resource ] does [
    (
        Any ↦ pattern acquire () ⇒ (done()) does [ !! done() ] ;
        Any ↦ pattern use (in : SomeInArg) ⇒ (result(out : SomeOutArg))
            declares [resultArg : SomeOutArg ; ]
        does [
            // use the resource
            resultArg := · · · ;
            !! result(resultArg)
        ] * ;
        Any ↦ pattern relinquish () ⇒ (done()) does [ !! done() ]
    ) *
]
```

2.3 Augmenting UCMs with BCOOPL Interface Specifications

The UCM notation has been augmented with an extension and some notational shorthands in order to have a better match with BCOOPL's language features. The augmentations are depicted in Figure 4. The extension is the more rigorously defined semantics of a scenario in progress along a path within a component. As in BCOOPL, the *one-at-a-time* regime applies, which means that only one thread of control is active at any one time, although multiple threads may be in execution on an interleaved basis. However, a scenario in execution can claim exclusive control over a component by means of enclosing a path segment in $\ll$ and $\gg$ markers. Notational shorthands have been provided for interface specifications, (asynchronous) message exchanges, and synchronization.

3 Running Example

A simplified, but realistic model of a client/server system is used as a running example. The client and the server share a resource that can only be used by one party at a time. In this example we show that a component, in this particular case the server, operates perfectly well in isolation, but it may fail when it is subjected to a composition in which another component, in this case the client, uses the same shared resource. This

Primitive	Abstract Syntax	Remarks
Assignment	*variable* := FQN*expression*	A FQN (Fully Qualified Name) denotes an object. It is specified as: *(Pseudo-)Variable.Pattern Name*$_1$.$\cdots$.*Pattern Name*$_n$
New	**new***Pattern Name*	The designated pattern is instantiated along with its subpatterns resulting in an object tree. Unreferenced objects are reclaimed by a garbage collector.
Send	FQN (*message args*)	A message is sent to the object designated with the FQN
Request	**request** FQN (*message args*)	Identical to a send with the exception that a reply (i.e., a notification) is sent only to a subobject of the object that issued the request.
Inlet (Pattern)	*CS specs* $\mapsto$ **pattern** *Name* (*in args*) $\Rightarrow$ (*notifications*) **does** [$\cdots$]	A message is received in an inlet which is implicitly defined in a pattern.
Lightweight Inlet (Pattern)	*CS specs* $\mapsto$ **pattern** *Name* (*vars*)	In contrast with an ordinary inlet, a lightweight inlet does not introduce a local scope. The received message arguments are stored in the designated variables.
Outlet	!!*Notification Name* (*message args*)	A notification is issued.
Client/Server Set Operations	*add* or *remove*	The add and the remove are currently the only supported operations.
Delegate	*not discussed further*	

Table 2. Primitives.

is a typical illustration of architectural mismatch where components make (possibly undocumented) assumptions on the environment [11]. The interface for the server is shown below, the interface and the implementation of the resource was already given in a previous section.

```
interface Server defines [
    Any ↦ (resource : Resource) ⇒ () [
        Any ↦ service (in : SomeInArg) ⇒ (result(out : SomeOutArg)) *
    ]
]
```

A server is initialized with a resource. Most likely, the server will use this resource, although it is not clear when the resource is used. This kind of behavior specification cannot be deduced from the interface specification. For this and other reasons not discussed here we have proposed a grey-box approach for component specification comprised of UCMs and BCOOPL interface specification [6]. The combination of UCMs

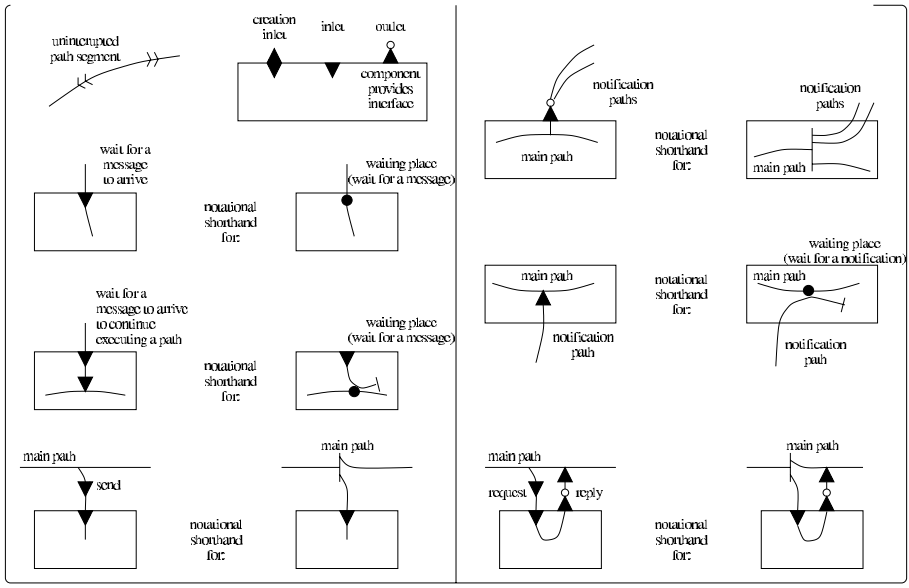


Fig. 4. UCM augmentations.

and BCOOPL interfaces gives us the opportunity to reason about the internal behavior of components and their interactions with other components.

The interactions between the server and the resource is depicted in Figure 5. A simplification is shown in the same figure in which the interactions between the two are modeled with responsibility points and the resource component is removed from the graphical representation altogether.

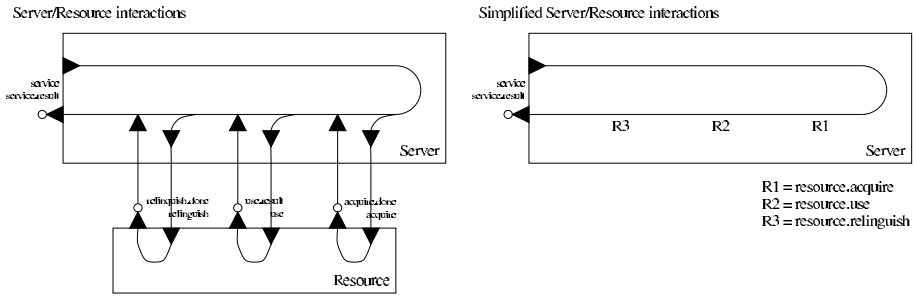


Fig. 5. Server/Resource interactions.

When the server is brought together with a client that uses the same resource in a manner depicted in Figure 6 the system might come to a halt because of deadlock. The interface for the client is given below.

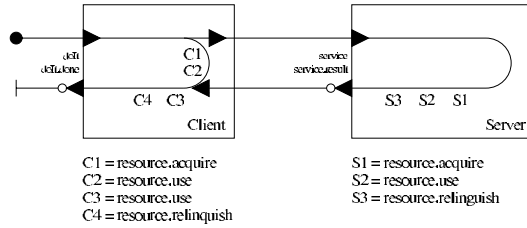


Fig. 6. Client/Server/Resource interactions.

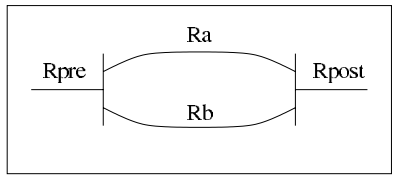
```
interface Client defines [
    Any ↦ (server : Server, resource : Resource) ⇒ () [
        Any ↦ doIt (in : SomeInArg) ⇒ (done(out : SomeOutArg)) *
    ]
]
```

Both the client and the server try to acquire the resource just after the client has sent a *service* request to the server. The system deadlocks if the client acquires the resource first. In that case, the client will never receive a reply (in the form of a *result* notification) from the server, since the server wants to acquire the resource but the resource was already acquired by the client and will be relinquished only after the server has replied. So, the client as well as the server want to proceed, but they cannot do so because they are awaiting an event that will not happen. In other words, this is a classical example of deadlock.

If, on the other hand, the server acquires the resource first, the system proceeds as intended. The server uses and then relinquishes the resource and continues with issuing a *result* notification. In the meantime, the client tries to acquire the resource and use it once before processing the result notification. Eventually, the client will succeed in doing so. After the client has processed the reply from the server, it proceeds with another use of the resource before relinquishing it.

4 Preparing UCM/BCOOPL Interface Specifications for Analysis

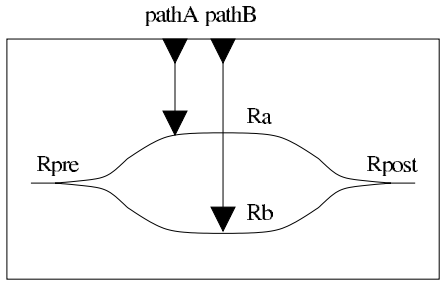
A component composition comprised of UCMs and BCOOPL interface specifications is translated into a corresponding BCOOPL program before it is analyzed. The translation can be done in an automatic and unambiguous way. We demonstrate how typical UCM constructs, like the AND/OR-fork/join and the request, translate into BCOOPL. The translation scheme for the AND-fork/join is given in Figure 7. The basic idea is to send two (asynchronous) messages to patterns defined in the object itself. The patterns that handle these messages are contained in an interleave (||) expression, so these two patterns are executed concurrently.




```
    ⋮
    ⋮
R_prc ;
self.pathA() ;
self.pathB() ;
(
    self ↦ pattern pathA () ⇒ () does [
        R_a
    ]
||
    self ↦ pattern pathB () ⇒ () does [
        R_b
    ]
) ;
R_post ;
    ⋮
    ⋮
```

Fig. 7. Translation scheme for the AND-fork/join.

The translation scheme for the OR-fork/join is along the same lines as for the AND-fork/join as is depicted in Figure 8. The forked paths are embedded in an alternative (+) expression rather than an interleave expression. In contrast to the AND-fork/join, only one path will actually be executed. The path to be executed is selected on the basis of message receipt in an inlet defined at the beginning of a forked path.




```
    ⋮
    ⋮
R_prc ;
(
    Any ↦ pattern pathA () ⇒ () does [
        R_a
    ]
+
    Any ↦ pattern pathB () ⇒ () does [
        R_b
    ]
) ;
R_post ;
    ⋮
    ⋮
```

Fig. 8. Translation scheme for the OR-fork/join.

As discussed before, a request is split into a message send followed by an inlet in which the reply notification is received. In between the message send and the receipt of a reply in an inlet, an object can perform additional computations that run concurrently with the activities of the object that services the request. The translation scheme for the request is given in Figure 9.

Given these translation schemes, we show in Figure 10 how the client/server example is translated into BCOOPL.

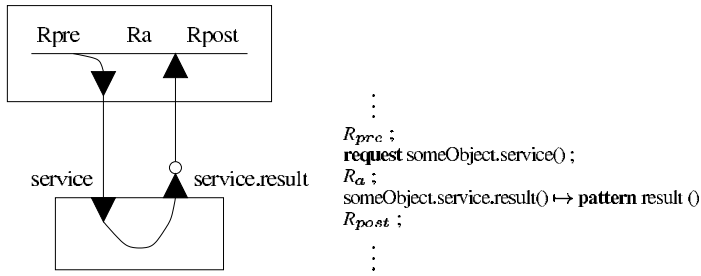


Fig. 9. Translation scheme for the request.

5 Analyzing Compositions

Finally, we show how a generated BCOOPL program is analyzed. This involves the following three steps:

1. type inferring pattern expressions;
2. building a communication graph annotated with thread information;
3. analyzing the composition with temporal orderings.

The first two steps are used to represent the program in such a way that it can be readily used in the final, analysis step.

The first step is type inferring the generated BCOOPL program in order to identify the types of pattern expressions as precisely as possible. Type information is used to deduce communication targets. In principle, communication targets can be distilled from a UCM. However, to better deal with polymorphism, type inferring a BCOOPL program yields more accurate type information. In addition, type information is also used to check the program's sanity. For instance, we can verify whether a variable that holds a target for a message send is initialized before it is used. Due to lack of space, type inferring BCOOPL programs is not discussed in this paper. A detailed discussion of type inference in Talktalk (a predecessor of BCOOPL), which is based on data flow equations, can be found in [5].

The second step is to build a communication graph detailing the *send-to* and *received-by* relations between (sub-)patterns. This graph is akin to a call graph for imperative programs. The communication graph is annotated with thread information to show the threads in which message sends and receipts occur.

In the final third step, the composition is analyzed by constructing directed graphs that show the temporal orderings between events (i.e., a message send and a message dispatch). Such graphs are constructed for each scenario specified in a UCM. Any violation of ordering constraints in a graph can be seen as a problem indicator.

For analysis purposes, BCOOPL's computational model is slightly adapted in order to simplify reasoning about component compositions without actually changing the semantics. A single message buffer is used instead of equipping each component with its own buffer. In reality, BCOOPL objects are equipped with individual buffers, which

```
// Client implementation
Any ↦ pattern client (client : Client, resource : Resource) ⇒ () implements [ Client ] does [
    Any ↦ pattern doIt (in : SomeInArg) ⇒ (done(out : SomeOutArg)))
        declares [ resultArg : SomeOutArg ; ]
    does [
        server.service(in) ;
        request resource.acquire() ; resource.acquire.done() ↦ pattern resourceAcquired () ;
        request resource.use(in) ; resource.use.result(out : SomeOutArg) ↦ pattern resourceResult (resultArg) ;
        request server.service.result(out : SomeOutArg) ↦ pattern serverIsDone (resultArg) ;
        request resource.use(in) ; resource.use.result(out : SomeOutArg) ↦ pattern resourceResult (resultArg) ;
        request resource.relinquish() ; resource.relinquish.done() ↦ pattern resourceRelinquished () ;
        !! done(resultArg)
    ] *
]

// Server implementation
Any ↦ pattern server (resource : Resource) ⇒ () implements [ Server ] does [
    Any ↦ pattern service (in : SomeInArg) ⇒ (result(out : SomeOutArg))
        declares [ resultArg : SomeOutArg ]
    does [
        request resource.acquire() ; resource.acquire.done() ↦ pattern resourceAcquired () ;
        request resource.use(in) ; resource.use.result(out : SomeOutArg) ↦ pattern resourceResult (resultArg) ;
        request resource.relinquish() ; resource.relinquish.done() ↦ pattern resourceRelinquished () ;
        !! result(resultArg)
    ] *
]

// System configuration (main start-up code)
Any ↦ pattern main () ⇒ ()
    declares [
        client : Client ;
        server : Server ;
        resource : Resource ;
        inArg : SomeInArg ;
        outArg : SomeOutArg ;
    ]
does [
    client := new client ;
    server := new server ;
    resource := new resource ;
    inArg := new someInArg ;

    resource() ;
    client(server, resource) ;
    server(resource) ;
    request client.doIt(inArg) ; client.doIt.done(out : SomeOutArg) ↦ pattern clientIsDone (outArg)
]
```

Fig. 10. Client-Server system translation into BCOOPL.

is of course a necessity for truly concurrent and distributed computing. The following rules hold:

- The temporal ordering of messages sent by one component is preserved in the message buffer.
- Message dispatching is based on a first-come-first-served basis of acceptable messages as far as an individual object (and its sub-objects) is concerned. Each object publishes its acceptable (sub-)patterns (i.e., there corresponding (sub-)objects are ready to be executed). A non-deterministic scheduling policy applies for choosing the next object to be executed.

As a consequence, the following general rule is valid: $t_{message\ buffering} < t_{message\ dispatching}$ (abbreviated as $t_{buf} < t_{disp}$). In words, a message is buffered before it is dispatched.

There is a one-to-one correspondence between regular expressions and state machines. Therefore, interface specifications, which are defined as regular expressions over interface interaction terms, imply temporal orderings. A typical example is given in the interface specification of a resource, which states that a resource cannot be used before it is acquired, and that it cannot be acquired again before it is relinquished. This can be formalized as shown below.

simplified Resource interface specification : $(acquire; use*; relinquish)*$

$t_{acquire\ disp_i} < t_{use\ disp_{ij}} < t_{use\ disp_{ij+1}} < t_{relinquish\ disp_i} < t_{acquire\ disp_{i+1}}$
with index i denoting the outer repetition cycle number, and
with index j denoting the inner repetition (use) cycle number

The temporal orderings can be shown as a directed graph in which the $<$ relation is represented as a directed edge between two nodes.

We are now in the position to analyze a composition on a scenario basis. Before discussing how all scenarios can be enumerated systematically, we show the general approach by taking the client/server system as an example. In this particular case, two scenarios can be recognized. In the first scenario, the client acquires the resource first. The temporal event orderings of this scenario are depicted in a slightly simplified form in Figure 11 (for one thing, the invocations of the resource's *use* pattern have been omitted). As can be seen in the figure, the scenario leads to a situation in which a notification (the *result* notification issued by the server) is dispatched before it is issued (i.e., buffered). This is a violation of the rule that a message just be buffered before it can be dispatched. Therefore, we conclude that the system comes to a halt as a result of deadlock. A second scenario can be devised in which the server requires the resource first. Due to space limitations we will not give a graphical representation here. In this case, however, no contradictions are found.

The enumeration of all scenarios is done by systematically considering all paths in a UCM. In particular, the OR-fork introduces two independent (sub-)scenarios, whereas the AND-fork leads to two concurrently running threads. In principle, all permutations of events in concurrent threads yield the set of scenarios to consider, which obviously leads to an explosion of scenarios. Fortunately, there are ways to prune the scenario set. First of all, we can check whether two threads are independent of each other by consulting the communication graph annotated with thread information. If components involved in communications originating from one thread do not use the components invoked from the other thread, then from an analysis point of view the execution of both threads can be regarded as a single scenario. Secondly, the temporal orderings implied by pattern expressions enforce particular message sequences (e.g., the *acquire ; use * ; relinquish* message sequence of a resource). This means that certain permutations of events in concurrent threads do not apply. For instance, in the client/server example, a resource can only be used if it has been acquired. Therefore, we do not have to consider the permutations of all resource *uses* stemming from the client and the server.

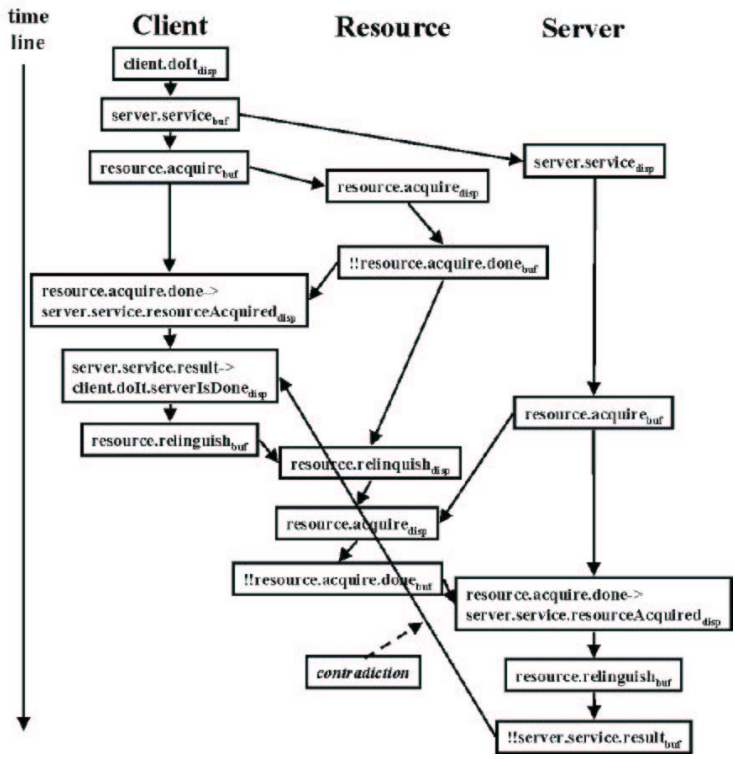


Fig. 11. Scenario leading to deadlock.

In conclusion, by analyzing temporal orderings, we might find contradictions which are an indication for errors in a specification, such as the presence of deadlock. Thus, erroneous behavior is found by considering all *possible* scenarios.

6 Discussion

We have shown how the presence or absence of deadlock in a system comprised of components can be proved. Another application area of this approach is to investigate reachability, a concept closely related to the notions of goals. By analyzing a scenario, we can verify whether the intended goals of that scenario are reached or not. This notion can be reformulated in terms of events. That is, we can analyze whether a particular event (message sending or message dispatching) or a set of events that is associated with a certain goal does happen.

As discussed before, systems comprised of compositions of concurrently running components are difficult to analyze due to the state explosions implied by nondeterministic behavior of concurrent components. Techniques have been developed to handle large number of states. For instance, a model-theoretic reasoning tool (i.e., a model checker), called the Symbolic Model Verifier (SMV), has been used to verify

models with more than 10^{20} states [3]. Although practical, but relatively small systems can be verified in this way, real-size systems can easily outdo the aforementioned number of states. A scenario-based approach like UCM reduces the number of possibilities to consider. The functionality provided by a system comprised of components is typically not equal to the sum of the functionalities provided by its parts. In general, components will be underutilized. UCM scenarios are employed to show what functionality of components is actually used by the system as a whole. This reduces the complexity involved in analyzing component systems. Furthermore, self-contained subsystems can be analyzed in isolation. An abstraction (a simplified model) of the subsystem can then be substituted in the overall system. Despite the reductions in complexity, the analysis of component compositions remains a computational expensive process.

It is interesting to explore the extent of what can be analyzed by the combination of UCM and BCOOPL interfaces. To put it differently, does the combination specify the behavior of a component composition completely? This turns out not to be the case. For one thing, pre- and post-conditions cannot be expressed formally in BCOOPL interfaces. (However, Pre- and post conditions can be specified informally by means of responsibility points in UCMs.) Although temporal orderings implied by regular expressions and client specifications can be seen as pre-conditions, there is no way to state pre- and post-conditions in terms of state variables. Thus, BCOOPL offers a static construct for specifying enabling conditions, rather than taking runtime, dynamic behavior into account. The advantage of this approach is that we do have a specifications of temporal orderings, which are hard to deduce from pre- and post-conditions alone.

There are no principal reasons not to support state variables in BCOOPL interfaces. It should be noted that the inclusion of state variables is not a violation of the principle that an interface should not commit to implementation details. For instance, size is an intrinsic property of a stack. Pre- and post-conditions for stack operations can be specified in terms of size without committing to an implementation yet. This notion is central in the component specification method Catalysis [9]. Catalysis also includes the concept of refinement. Interface (type) specifications can be gradually refined into an implementation as long as it is assured that an implementation is in conformance with its type.

Nevertheless, potential problems can be pinpointed by the combination of UCMs and BCOOPL interfaces. In some cases, however, more problems will be spotted than can actually happen at runtime due to the fact that enabling conditions cannot be specified precisely enough at present. The use of state variables in interface specifications will be explored in the near future.

7 Concluding Remarks

We have discussed an approach for modeling component compositions and analyzing them statically. UCMs and BCOOPL interfaces strike the balance between understandability (by humans) and preciseness (for reasoning about component compositions). Interacting UCM scenarios give a high level, easy to understand overview of the behavioral aspects of a system. UCM scenarios in combination with BCOOPL interface specifications allow to reason about it formally. In addition, the use of UCM scenarios helps in reducing the complexity of the analysis process.

Future work include modeling gradual refinements (from type specification to implementation) and investigating quality attributes (e.g., performance) other than behavior that are amenable for analysis purposes using the approach discussed in this paper. In addition, we want to look at tool support for analyzing real-size systems.

References

1. Peter Bouwman and Hans de Bruin. Talktalk. In Peter Wisskirchen, editor, *Object-Oriented and Mixed Programming Paradigms*, Eurographics Focus on Computer Graphics Series, chapter 9, pages 125–141. Springer-Verlag, Berlin, Germany, 1996.
2. R.J.A. Buhr. Use Case Maps as architecture entities for complex systems. *IEEE Transactions on Software Engineering*, 24(12):1131–1155, December 1998.
3. J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Fifth Conference on Logic in Computer Science*, 1990.
4. R.H. Campbell and A.N. Habermann. The specification of process synchronization by path expressions. In *Lecture Notes in Computer Science 16*, pages 89–102. Springer-Verlag, Berlin, Germany, 1974.
5. Hans de Bruin. *DIGIS: a Model Based Graphical User Interface Design Environment for Non-Programmers*. PhD thesis, Erasmus University Rotterdam, November 10, 1995.
6. Hans de Bruin. A grey-box approach to component composition. In Krzysztof Czarnecki and Ulrich W. Eisenecker, editors, *Proceedings of the First Symposium on Generative and Component-Based Software Engineering (GCSE'99), Erfurt, Germany*, volume 1799 of *Lecture Notes in Computer Science (LNCS)*, Berlin, Germany, September 28–30, 1999. Springer-Verlag.
7. Hans de Bruin. BCOOPL: A language for controlling component interactions. In H.R. Arbnia, editor, *Proceedings of the International Conference of Distributed Processing Techniques and Applications (PDPTA'2000)*, volume 2, pages 801–807, Las vegas, Nevada, USA, June 26–29, 2000. CSREA, CSREA Press.
8. Hans de Bruin. BCOOPL: Basic Concurrent Object-Oriented Programming Language. *Software Practice & Experience*, 30(8):849–894, July 2000.
9. Desmond Francis D'Souza and Alan Cameron Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Object Technology Series. Addison-Wesley, Reading, Massachusetts, 1998.
10. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, Reading, Massachusetts, 1995.
11. David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, November 1995. Carnegie Mellon University.
12. David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
13. Ole Lehrman Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, Reading, Massachusetts, 1993.
14. J.L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1981.
15. Jan van den Bos and Chris Laffra. Procol: a concurrent object language with protocols, delegation and persistence. *Acta Informatica*, 28:511–538, September 1991.

Product Instantiation in Software Product Lines: A Case Study

Jan Bosch[1]

University of Groningen
Department of Computing Science
P.O. Box 800, 9700 AV Groningen
Jan.Bosch@cs.rug.nl
http://www.cs.rug.nl/~bosch

Mattias Högström

Blekinge Institute of Technology
Department of Software Engineering and Computer Science
S-372 25 Ronneby, Sweden

Abstract. Product instantiation is one of the less frequently studied activities in the domain of software product lines. In this paper, we present the results of a case study at Axis Communication AB on product instantiation in an industrial product line, i.e. five problems and three issues. The problems are concerned the insufficiency of functional commonality, features spanning multiple components, the exclusion of unwanted features, the evolution of product line components and the handling of initialization code. The issues discuss architectural compliance versus product instantiation effort, quick-fixes versus properly engineered extensions and component instantiation support versus product instantiation effort. The identified problems and issues are based on the case study, but have been generalized to apply to a wider context.

1. Introduction

The development and evolution of software is expensive and complex, causing few software development and maintenance projects to deliver on schedule, within budget and at the required quality level. For decades, the reuse of existing software has been proposed as the most promising approach to attack these problems. In the earliest proposals, software components represented functions that could be reused [11], whereas later proposals suggested object-oriented programming, i.e. the use of classes and inheritance, to achieve reuse of software [8]. During the second half of the 1980s, it was identified that for software reuse to be beneficial, the components need to be of larger granularity than functions and classes, which lead to the introduction of object-oriented frameworks [8,6] and, later, component-oriented programming [14].
All the approaches discussed above aimed at achieving community-wide reuse of software assets. Although successful attempts exist, in general one has to conclude that the promised benefits of software reuse have not been achieved. One explanation is that the software reuse community tried to overcome two dimensions of complexity

[1] This research was performed while the first author was at the Blekinge Institute of Technology, Sweden.

G. Butler and S. Jarzabek (Eds.): GCSE 2000, LNCS 2177, pp. 147-161, 2001.

at once, i.e. reuse between software products and reuse over organizational boundaries. In response to this, the notion of software product lines, aiming at reuse between software products, but within an organization has achieved considerable amounts of attention during recent years, e.g. [9, 7, 16, 4].

In this paper we focus on the instantiation of products within a software product line. In particular, we present the problems and issues that we identified in a case study that we performed at Axis Communications AB. Since product line based development differs significantly from the traditional approach to developing software, it is reasonable to assume that new problems or new variants of known problems exist. Although we earlier have published case studies discussing software product lines in general [3], product line components [2] and product evolution [15], to the best of our knowledge, few research results are available that address product instantiation in software product lines.

The primary goal of our case study was to identify the problems and issues associated with the instantiation of software products in software product lines. The reasons for choosing Axis Communications for the case study are that we have had cooperation with this company for several years, it has a few well-established software product lines that have been operational for several years and the company employs a philosophy of openness. Finally, we believe that the company is representative for 'normal' software development organizations, e.g. development departments of 10 to 100 engineers and developing products sold to industry or consumers.

The most appropriate method to achieve our goals with this case study, we concluded, was through detailed analysis of a number of software products, discussions with system architects and studying component and product documentation. For practical reasons, we chose to narrow the scope of the study to products within the same product family. Although the analyzed products belong to the same product family, the variety of configurations concerning both software and hardware gives rise to many interesting problems that can be found in other product families as well.

The contribution of the paper, we believe, is that it identifies a set of important problems and issues associated with the instantiation of products within the context of software product lines. Awareness of these problems and issues allows practitioners to minimize the negative effects and researchers may use these topics in their research.

The remainder of this paper is organized as follows. In the next section, we present the case study company and the software product line that is the focus of our case study. Section 3 discusses the process of product instantiation within the product line. In section 4, the problems that we identified are discussed, whereas section 5 discusses the issues that we recognized. Related work and our conclusions are presented in section 6.

2. Axis Communication AB

Axis Communication was incorporated in 1984 and developed a printer server product that allowed IBM mainframes to print on non-IBM printers. The first product was a major success that established the base of the company. In 1987, the company

developed the first version of its proprietary RISC CPU that allowed for better performance and cost-efficiency than standard processors for their data-communication oriented products. Today, the company develops and introduces new products on a regular basis. Since the beginning of the '90s, object-oriented frameworks were introduced into the company, and since then, a base of reusable assets is maintained, from which most products are developed.

Axis develops IBM-specific and general printer servers, CD-ROM and storage servers, network cameras and scanner servers. Especially the latter three product types, i.e. the network devices, are developed and evolved using a common product line architecture and reusable components. In figure 1, an overview of the product-line and the product architectures is shown. The organization is more complicated than the standard case, with one product-line architecture (PLA) and several products below this product-line. In the Axis case, the software product line is hierarchically organized, i.e. there is a top product-line architecture and the product-group architectures, e.g. the storage-server architecture. Below these, there are product architectures, but since generally several product variations exist, each variation has its own adapted product architecture, because of which the product architecture could be called a product line architecture.

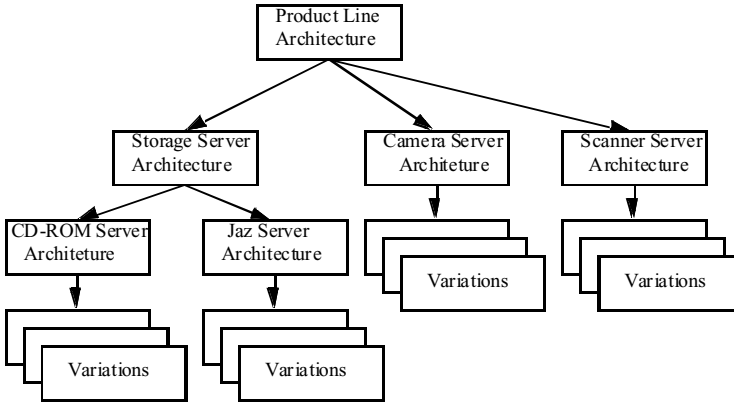


Figure 1. The Axis Software Product Line.

Orthogonal to the products, Axis maintains a product-line architecture and a set of reusable assets that are used for product construction. The main assets are a framework providing file-system functionality and a framework providing a common interface to a considerable set of network protocols, but also smaller frameworks are used such as a data chunk framework, a smart pointer framework, a 'toolkit' framework providing domain-independent classes and a kernel system for the proprietary processor providing, among others, memory management and a job scheduler. In figure 2, the organization of the main frameworks and a simplified representation of the product-line architecture is shown.

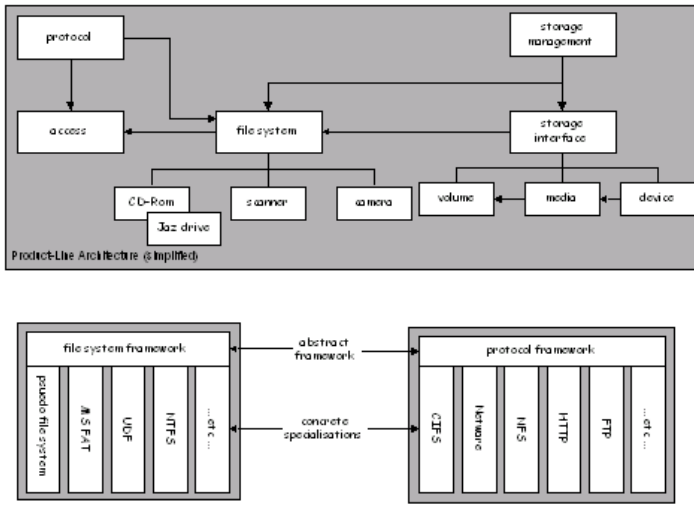


Figure 2. Overview of the Main Frameworks Used in Axis' Products.

3. Product Instantiation

In this section, we describe the software architecture and the process of product instantiation for the products in the network device product line. We focus on the StorPointCDE100 product, which is a caching CD-ROM server. It is a member of the storage product family, which consists of harddisk, cd-rom, jaz-drive servers. The process of product instantiation consists of three steps, i.e. task scheduler configuration, service configuration and component parameterization. Finally, we discuss the variability mechanisms used in the software product line.

The Architecture

The underlying hardware platform is a proprietary system-on-a-chip solution. The ETRAX 100 chip is a highly specialized and optimized chip designed to boost network performance. It provides built-in SCSI/IDE ATA-2 and 10/100Mbit ethernet interface. It is also equipped with a flash memory that allows remote updating of the software system. Like most embedded systems, the software runs on top of a micro kernel, in this particular product the OSYS real-time kernel. OSYS is an in-house developed kernel extended with a task-scheduler and a mail-communication package.

Product Template

Axis maintains a generic architecture template from which individual products are derived. The template is a skeleton of integration code connecting the kernel, the mailbox-communication package, the job scheduler, the frameworks and the components. The high degree of similarity between the product architectures allows

all product configurations to be combined in one single code base. The product specific code is attached to the reference-product through a C-function interface, thereby allowing the use of product-specific, but type-equivalent components. The 'completeness' of the reference product results in that many product-specific variation points have become distributed over the reference product, cross-cutting the architecture. In order to support all configuration subsets, preprocessor directives are used as a mechanism to attach respectively detach components at compilation. This is performed through a special header file, specific for each product configuration.
An overview of the architecture, the interaction of the mailbox-communication, tasks and the jobs is presented graphically in figure 3.

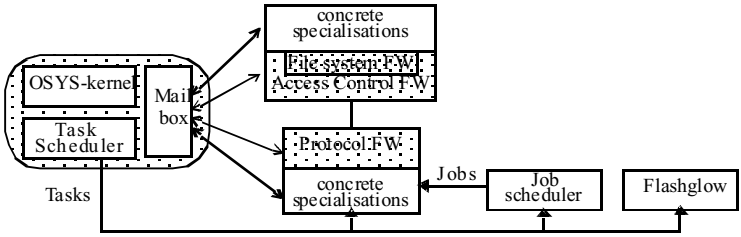


Figure 3. A Simplified View of the Core Product Architecture.

Since the reference product specifies the component connections, the remaining configuration activities include the definition of the task-list for the scheduler and the parameterization of the mailbox handler.

Task-Scheduler Configuration. The task scheduler is configured via a globally accessible C-struct. Each product variant supplies its own configuration by explicitly linking it into the final product. Each entry (task) is a function pointer referring to a startup routine in a 'active' component (typically a framework loop), which is scheduled using a round-robin algorithm.

```
struct tasks [ ] = {
          idle_main,
          job_scheduler,
          flashglow,
          protocol_main
};
```

The 'flashglow' handles run-time updates of the executable-binary, and the 'protocol_main' handles incoming network requests (jobs) and inserts them into the queue of the 'job_scheduler'.

Service Configuration. Services are typically component plug-ins of a framework. Inside each component resides a start-up routine used for initialization and registration in the framework. When the system boots up, the scheduler posts 'public_create' and 'public_init' messages via the mailbox. The mailbox itself contains hard-coded function calls referring to each start-up routine for all available components. When the messages arrive, first the frameworks are initialized.

Subsequently, the components are initialized and registers themselves at the frameworks. The initialization via the mailbox and component start-up routines provides a uniform way to add and remove services. At compilation a specific configuration can choose whether or not to include a service. From the reference product's perspective the components and frameworks are detached from each other, but in reality implicit dependencies exist. This is discussed in the next section.
Below, a small code fragment is shown illustrating the use of precompiler directives in the mailbox.

```
MailBox::async_message(<init message>) {
   switch (<message>) {
      #if SMB_INCLUDED
         case SMB_PROG:
            smb_main ( <init> );
            break;
      #endif
      #if APPLETALK_INCLUDED
         case APPLETALK_PROG:
            appletalk_main ( <init> );
            break;
      #endif
      ..etc..
   }
}
```

Component Parameterization. Although the frameworks and the framework plug-ins seem independent, in reality there is a hard coupling between the components and the frameworks. The components are not reused at binary level, but at source code level. The source code is a mixture of common code and product-specific code guarded by precompiler directives. Consequently, each product variant needs to specify preprocessor directives that include respectively exclude pieces of code before the product source is compiled into a binary. Both the reference product and the components need to recompiled, when a service is added. Enabling or disabling a feature frequently requires recompilation of the complete source code tree, since features often have relations to multiple components.
Below, a code fragment from the file system component is shown in which the ISO9660 standard implementation is conditionally included, depending on the F_ISO9660 and FS_9660 precompiler flags.

```
void CDMediumFSPrototypeMap::init( ) {
   #if defined(F_ISO9660) && FS_9660
      this->registerPrototype( new ISO9660Volume( ) );
         //Registers filesystem in the factory
   #endif
   ...etc...
}
```

Summarizing Discussion

Axis' product line consists of frameworks, framework plug-ins (components) and a configurable skeleton of integration code. Inheritance and C++ templates are used as means to obtain variability of the core assets. The architecture is characterized by its pervasive use of design patterns, among others Abstract factory, Composite, Observer and Singleton. Axis has taken reuse beyond the availability of core assets by providing a skeleton of generic integration code that, once configured, represents a complete, functional product. This approach has allowed for taking advantage of the similarities within the product family to its full extent.

The products in the product family are functionally similar but technically dissimilar. The hardware differences between products complicate the successful abstraction of common features. Nevertheless, through the extensive use of preprocessing, Axis has managed to do it in practice. Typical types of usage are: definition of constants and parameters, configuration of orthogonal features that span over multiple components, configuration of low-level details hidden behind several abstraction levels, removing calls/references/dependencies between assets and enabling/disabling hard-coded sections of source code. The downside of the focus on the current requirements is that the product line assets are hard to evolve, with consequent effects: increased complexity of the source code, non-trivial impact analysis, decreased reuse in testing and frequent recompilation of the complete source tree. The solution does have several advantages: it maximizes reuse, much more assets can be made "generic", assets can be large without loosing configurability and the size of the executable can be kept small since unnecessary features can be left out. In the next section we have abstracted some of the problems that we found in our case study into more general problems specific to product instantiation in software lines.

4. Problems

Based on the analysis of the products in the software product line, interviews with software architects and engineers and documentation collected at the organizations, we have identified a number of problems associated with product instantiation in software product line based development that we believe have relevance in a wider context than just Axis Communication AB. Some problems are applicable for product-line development in general, while other problems are specific the embedded systems domain.

The problems presented here share two main underlying causes, i.e. variability and evolution. The variability required from the reusable assets to meet the needs from products in the family creates difficulties because separating general and product specific functionality is generally not trivial. In addition, evolution of the products and the reusable components that define most of their functionality causes solutions and design decisions that were correct at the time they were made to become invalid. The problem is that removing the effects of these solutions and design decisions may be prohibitively expensive.

Functional Commonality Is not Enough

Problem. The reusable components represent the core of software product lines since these capture the commonalities between the products. The more commonalties that can be extracted from the products and captured into reusable components, the smaller the amount of code that has to be maintained. In addition, larger components can be reused more efficient [Szyperski 97].
The problem is that commonalities between products cannot always be captured in generic reusable components, or at least not without considerable effort. A number of instances of this problem exist. First, products may have conflicting quality requirements, which require component implementations that are incompatible with each other. Second, the products may operate on different hardware platforms and it may prove difficult to avoid the effects of these differences in the component. Finally, although all or most products in the family may require a particular domain of functionality represented by a component, different products or subsets of products may require features that cross-cut the component functionality. Since these features need to be optional, including or excluding a feature may have effects at several locations in the component.

Example. The product line of Axis Communications comprises a variety of different products. The products do not share a common hardware platform. They differ in configuration of the hardware chip, among the differences are network connectivity hardware, cache, flash memory and bus architecture (SCSI/ATA-2). The products can however also differ in peripheral configuration, e.g. data media devices (hd/cd/jaz). A product is often released in several configurations focussed on particular market segments. Variants aimed for the high-end market are equipped with expensive hardware peripherals, whereas low-budget variants often lack these. For example, Axis StorPointCDE100 is a caching 100 Mbit cd-rom server while StorPointCD10 is a 10 Mbit non-caching cd-rom server. The two products are functionally almost identical, except for the difference in hardware and the presence of a cache. The commonalties between the product variants do not allow for the development of large reusable assets, because the implementation of the additional features of the high-end product cross-cuts the component functionality and cannot be handled easily by traditional variability mechanisms.

Causes. One may argue that this problem is caused by a poor decomposition of the system into its main components, i.e. a badly designed software architecture. We investigated this and found no evidence of this being the case. Instead, we became convinced that even an otherwise optimal decomposition may cause such problems. The system does simply not allow for a decomposition that supports the variability requirements for all aspects, e.g. hardware, features, and domain. As we identified in [2], functionality related to the different aspects becomes intertwined early in the development process and varying one of these aspects independent of the others once developed code exists is very difficult.

Single Feature Spans over Multiple Components

Problem. In the previous section, the problem was that multiple features affected a single component. However, the opposite situation, i.e. an optional feature affecting multiple components, creates difficulties as well. Configuring a single component is manageable, but some features require changes in the interaction between components, affecting the software architecture and consequently, the initialization of the components. In addition, an orthogonal feature may affect the implementation and the variability mechanism of individual components. Features that span over multiple components may require architectural variability to the extent that the benefits associated with software product lines are compromised.

Example. As mentioned earlier, the StorPointCDE100 product is a caching cd-rom server whereas the StorPointCD100 product is a non-caching cd-rom server. The collaborating components behave and interact differently depending on the presence of a cache. If a cache is present, a number of configurational issues needs to be dealt with. First, the system must be configured for handling a harddisk (which caches the content on the CDs). Second, the harddisk requires a read-write file system to be added to the system, in addition to the ISO9660 read-only file system used for the CD-ROMs. Third, the cache module needs to be configured (size of cache, block size, prefetching, etc.). Finally, the cache must be embedded in the software architecture of the product, i.e. the new components have to be integrated with the existing architecture and the connections of the existing components need to be reconfigured to take benefit from the presence of a cache. The presence of a cache affects the parameterization and interaction of several components, and hence sections of several initialization routines must be maintained to support a single feature.

Causes. One can identify two main causes for this problem. First, the reusable assets of a software product line are not optimized for a specific product, but for a family of products. This means that despite the difficulty of integrating a particular feature in the product software, the current decomposition and implementation of the software product line may still represent the optimal solution. The second cause is that loose coupling is usually favored as a means to decrease dependencies between components and achieve reuse on a higher level, but not all features can effectively be handled late in the configuration process.

Excluding Component Features

Problem. In our experience, components in software product lines are typically relatively large entities, e.g. 100 KLOC. Several of the companies that we cooperate with make use of object-oriented frameworks to implement the component functionality. Several of these frameworks, typically black-box, implement the superset of the functionality required by the products in the family. However, since the features implemented in the framework typically have relations to other parts within the framework and to other frameworks, it is normally hard to exclude functionality. Especially in the domain of embedded systems, this conflicts directly with, among others, the resource efficiency quality requirement. Variability can, in several cases, not be supported at the component level without including unused

features. Large reusable components have a complex internal architecture with dependencies between its parts, which complicate the exclusion of parts of the component.
An additional complicating factor is the fact that components, during evolution, often have a tendency to incorporate implicit dependencies to other components it uses, see also [Bosch 99b]. These implicit dependencies make it even harder to exclude unused functionality from components.

Example. One of the components in the software product line studied in this paper is a network protocol framework that is reused by all products. It supports a plethora of different protocols. Most network protocols are implemented on top of lower level protocols. The protocols implement different abstraction levels, ranging from details about bit transmission to high-level application logic. For, among others, performance reasons, the layers in a protocol stack are rather tightly coupled, which make it hard to exclude unused protocols. The tight coupling results in relatively large components. The problem is that it generally is very hard to exclude code of a protocol due to the hard coupling in the implementation, whereas especially in embedded systems it is important to keep the system 'footprint' as small as possible.

Causes. The main cause for this problem is evolution. Often, the first version of a component supports the variability requirements on the product line at the time of design. During evolution, typically two types of change scenarios cause the discussed problem. First, functionality that up to now has been included in all products is required to become optional because some new product or a new version of some product demands this. Second, new features are added to the component because a relevant subset of the product family requires this. During implementation, it is discovered that the feature affect the component in multiple locations, complicating the implementation of the optionality of the new feature. A final cause is discussed in the next section.

Managing Evolution of Product Line Components

Problem. The core components represent common functionality that is shared by the products. For each product, these components are instantiated and configured to support the product specific requirements. The number of contexts in which a component is used may be substantial. During evolution of a component, it may therefore be difficult to verify compatibility of the new version of the component with all contexts in which the component may be used. This results in the situation, especially during development, where products that successfully compiled and ran one day, suddenly exhibit failures on the next. The problem is a software configuration management problem, i.e. how to handle impact analysis on a practical level. Too much analysis slows down production and delays the introduction of new features in products, whereas too little analysis may require additional work to correct new versions of components that caused errors in products that incorporate it.

Example. Axis Communications has a product release schedule that updates the product hardware once or twice per year and the product software three to four times

per year. Customers that have obtained Axis' products can download new versions of the software and install it in their product, typically overwriting a flash memory. Axis constantly enhances its hardware to support more and more functionality. Initially, a SCSI interface was the only interface provided for connecting harddisks and cd-roms. But, the low budget market demanded the ATA interface, since the devices are considerably less expensive than equivalent SCSI devices. There are fundamental differences between the ATA and SCSI interfaces. ATA is synchronous and non-multitasking, whereas SCSI is asynchronous and multitasking. Initially, the software architecture and the components were designed for SCSI, but later versions of these assets had to be adapted to support the ATA interface standard as well, though not necessarily at the same time. It turned out to be challenging to incorporate modifications in core components that are used in all product configurations, especially when the components go through extensive product-specific configuration or are used in several contexts, i.e. the component should be able to cooperate with many different components. Maintenance turned out to be complex, tedious and time consuming, since it was hard to do a thorough up-front impact analysis that covers all contexts in all configurations. Incidental dependencies might generate overlap in functionality or other conflicts. Occasionally, implemented modifications had to be removed and reimplemented differently to solve configuration conflicts.

Causes. The first cause is that the core components are shared entities and all products that incorporate these components are affected by modifications. Whenever the core assets evolve, all products evolve simultaneously. Common features get ideally inserted in the core components, and hence it is desirable that a change is compatible with all affected products. The second cause is the high frequency of incorporating modifications in these core assets. This makes it practically impossible to verify compatibility for each single modification.

Initialisation Code Is Scattered and Hidden

Problem. Although the reusable components are shared by all or most products, the components are instantiated and configured for each context in which they are used. One problem that we have identified is the location of initialization code. A reusable component typically contains two types of initialization code. First, generic initialization code that instantiates the internal entities and binds them appropriately. Second, context specific initialization code that configures the product-specific aspects of the component instantiation. Typically, even generic component functionality needs to be configured for product specific details. The ambition of software product lines is that products in the same product family to share as much common code as possible, but at the same time are not forced to duplicate similar code for all variants. The problem is that, due to this structure, initialization code is spread out throughout the system. First, this causes potential ordering problems during system start-up. Because the code is spread out through the system, it is hard to predict whether all components will have all required information, e.g. references to other components, available at the time of initialization. Second, it is hard to determine whether the initialization code is complete. If it is not, uninitialized references and similar problems may cause unpredictable system behavior.

Example. The products in the Axis product line are built around a micro-kernel, a mailbox and a task handler. These parts are relatively tight coupled with the product, since those parts manage most of the initialization for all products. Initialization can be viewed as part of the reusable components, since it is shared between all products. The problem is that the initialization code in a layered architecture of components is usually hard to access and modify. As a consequence, the product specific details are actually configured by code from inside the components. One disadvantage is that it limits the ways in which a product can be tailored and new products can be incorporated, and it is not obvious what code actually is used in a specific product or when the code is obsolete and can be removed.

Causes. The primary cause for this problem is there exist two forces that affect the location of initialization code. On the one hand, one can argue that initialization code ought to be associated with the instantiated component because it is a logical part of it. For instance, classes in object-oriented languages have their own constructor method. On the other hand, locating the initialization code for all components in a single component allows for more control over the ordering of instantiation and binding of components. In addition, product-specific functionality and features that affect multiple components can be implemented and instantiated more easily. The optimal balance between these forces is, in our opinion, specific to each software product line. However, typically, generic component initialization code should be associated with the component and most product-specific initialization code should be centralized. This consequently leads to initialization code being spread out through the system.

5. Issues

The problems discussed in the previous section present issues that are plain problematic and need to be addressed in software product lines. In this section, we discuss three issues. Different from problems, issues discuss the balancing between two conflicting forces.

Architectural Compliance versus Product Instantiation Effort

One of the primary advantages of software product lines is the fact that newly incorporated features automatically become available to all products in the product line. Since the cost of incorporating a new feature is shared among several products, the maintenance cost per product can be decreased drastically.
Software products are developed based on the product line assets by deriving a product architecture from the product line architecture and, subsequently, selecting, instantiating and configuring product line components. When the product requirements force the product architecture to deviate considerably from the product line architecture, fewer product line components can be used for the product and, consequently, more product specific components need to be developed. However, this may still be the most cost-effective development strategy because achieving the

product requirements by adding product-specific code to product line components may require even more effort.
However, if product line members are less compliant to the product line architecture, the advantages associated with product line evolution are diminished because new features can only partially be implemented at the product line level and need to be implemented per product for the remaining part.
Thus, a conflict exists between the ease with which a product can be instantiated and the cost of evolution in the software product line. The software architecture team for the product line has to decide where product architectures may deviate from the product line architecture and where the product line concepts need to be enforced.

Determining Value of an Investment

New requirements on the software product line assets that have architectural impact can generally be implemented in two ways, i.e. using a quick-fix avoiding architecture reorganization or by a properly engineered, but expensive implementation. It is generally difficult to decide whether to accept the drawbacks of a quick-fix or invest in a proper but expensive upgrade. The software product line approach is an architecture-centric approach. That means that common features are implemented once, but may be reused by any product in the product family. This means that the cost of a feature can be divided over multiple products. Unfortunately, architectural changes may cause ripple effects onto existing products, forcing these product to evolve with the architecture. Proper implementation of a requirement with architectural impact is potentially very expensive.
A quick-fix avoids architecture reorganization and is consequently less effort demanding, but degrades the structure and conceptual integrity of the software product line. Multiple quick-fixes will cause architecture erosion and decrease the economical value of product line assets.
In order to make a well-founded decision, an investment analysis of the different alternatives may be necessary. A large up-front investment may prove useless due to unexpected technical issues or market events. A quick-fix requires less effort and allows for short time-to-market, but are harmful for maintainability of the product line. Since typically no investment analysis is done, quick-fixes are often not replaced after the deadline, for instance because few understand the actual implementation and because it requires effort that otherwise could be spent on adding new features or products to the product line.
The question is which approach provides the highest return on investment. It is easy to criticize quick-fixes, but in some cases it is the preferable approach. The lack of economical models make it hard to show the long-term benefits of a software product line and how it is affected by various maintenance alternatives.

Component Instantiation Support versus Product Instantiation Effort

Components in a software product line are generally large and relatively complex. Instantiating and configuring a component for a new member of the product line (or a new version of an existing member) may therefore be difficult and time consuming. Since this has to be repeated for all product line components used in the product, the effort required for the instantiation of the product may be substantial.

Although a software component may just provide the source code and the documented provided, required and configuration interface, the developers of the component are able to provide more advanced instantiation and configuration support. For instance, in [13] the use of visual programming tools and domain specific languages is discussed to instantiate and configure object-oriented frameworks. The availability of such tools can decrease the instantiation effort considerably.

Although tools supporting component instantiation provide important support for product instantiation, there are two important issues that should be considered. First, the development of component instantiation support is effort demanding and increases the total component development cost substantially. Second, since more software is associated with a component, i.e. the component code and the instantiation support, the incorporation of new features in the component will become more expensive because the instantiation support must evolve as well.

The balance between providing component instantiation support and product instantiation effort is influenced by the number of new features that need to be incorporated and by the number of product instantiations that incorporate the product. Consequently, component instantiation support is more suitable for stable domains where relatively few new features are incorporated.

6. Conclusions

The notion of software product lines currently provides the most promising approach to increasing reuse of software and, consequently, decreased development cost and time-to-market. Several authors have discussed software product lines. For instance, in [7], the authors discuss an approach to software product lines consisting of application family engineering, component engineering and application engineering phases. Although product instantiation is discussed, the problems and issues discussed in this paper are not addressed. [16] discuss the FAST method, which is a systematic approach creating a software product line with a primary focus on the use of domain-specific languages and code generators for these languages. Other work in the area of software product lines includes [1, 5, 10]. However, none of these publications discusses product instantiation in software product lines in detail. In [4], we discuss the development, deployment and evolution phases of software product lines, but do not discuss the problems and issues presented in this paper.

In this article, we have presented the results of a case study identifying the problems and issues associated with product instantiation. The case study was performed at Axis Communications AB and focussed on their set of CD-ROM products in the networked devices product line. We identified five problems and three issues. The problems we discussed are concerned with the insufficiency of functional commonality, features spanning multiple components, the exclusion of unwanted features, the evolution of product line components and the handling of initialization code. The issues discuss architectural compliance versus product instantiation effort, quick-fixes versus properly engineered extensions and component instantiation support versus product instantiation effort.

The contribution of the paper, we believe, is that it identifies a set of important problems and issues associated with the instantiation of products within the context of software product lines. Awareness of these problems and issues allows practitioners to minimize the negative effects and researchers may use these topics in their research.

Acknowledgments
We would like to thank Axis Communication AB; in particular Torbjörn Söderberg, Hendrik Ruijter and Mikael Starvik for their valuable comments and encouragements.

References

1. L. Bass, P. Clements, R. Kazman, *Software Architecture In Practice*, Addison Wesley, 1998.
2. Jan Bosch, 'Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study', *Proceedings of the First Working IFIP Conference on Software Architecture*, February 1999.
3. Bosch, J., 'Product-Line Architecture in Industry: A Case Study', *Proceedings of the 21st International Conference on Software Engineering*, May 1999.
4. J. Bosch, *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*, Addison-Wesley, ISBN0-201-67494-7, 2000.
5. D. Dikel, D. Kane, S. Ornburn, W. Loftus, J. Wilson, 'Applying Software Product-Line Architecture,' *IEEE Computer*, pp. 49-55, August 1997.
6. M. Fayad, D. Schmidt, R. Johnson, *Building Application Frameworks - Object-Oriented Foundations of Framework Design*, ISBN 0-471-24875-4, Wiley, 1999.
7. L. Jacobson, M. Griss, P. Jonsson, *Software Reuse: Architecture, Process, and Organisation for Business Success*, Addison-Wesley-Longman, May 1997.
8. R. Johnson, B. Foote, 'Designing Reusable Classes', *Journal of Object-Oriented Programming*, vol. 1, no. 2, pp. 22-35, 1988.
9. F. van der Linden (Editor), 'Development and Evolution of Software Architectures for Product Families', *Proceedings of the Second International ESPRIT ARES Workshop*, Las Palmas de Gran Canaria, Spain, LNCS 1429, Springer Verlag, February 1998.
10. R.R. Macala, L.D. Stuckey, D.C. Gross, 'Managing Domain-Specific Product-Line Development,' *IEEE Software*, pp. 57-67, 1996.
11. M. D. McIlroy, 'Mass Produced Software Components,' in 'Software Engineering,' *Report on A Conference Sponsored by the NATO Science Committee*, P. Naur, B. Randell (eds.), Garmisch, Germany, 7th to 11th October, 1968, NATO Science Committee, 1969.
12. Parnas, D., 'On the Criteria to be Used in Decomposing Systems into Modules', *Communications of the ACM*, vol. 15, no. 12, pp. 1053-1058, 1972.
13. D. Roberts, R. Johnson, 'Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks,' *Proceedings of the Third Conference on Pattern Languages and Programming*, Montecillio, Illinois, 1996.
14. C. Szyperski, *Component Software - Beyond Object-Oriented Programming*, Addison-Wesley, 1997.
15. M. Svahnberg, J. Bosch, 'Evolution in Software Product Lines: Two Cases', *Journal of Software Maintenance*, Vol. 11, No. 6, pp. 391-422, 1999.
16. D. Weiss, Robert C. Lai, *Software Product-Line Engineering: A Family-Based Software Development Process*, Addison-Wesley-Longman, ISBN 0-201-69438-7, 1999.

Mixin-Based Programming in C++[1)]

Yannis Smaragdakis
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
yannis@cc.gatech.edu

Don Batory
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
batory@cs.utexas.edu

Abstract. Combinations of C++ features, like inheritance, templates, and class nesting, allow for the expression of powerful component patterns. In particular, research has demonstrated that, using C++ *mixin classes*, one can express layered component-based designs concisely with efficient implementations. In this paper, we discuss pragmatic issues related to component-based programming using C++ mixins. We explain surprising interactions of C++ features and policies that sometimes complicate mixin implementations, while other times enable additional functionality without extra effort.

1 Introduction

Large software artifacts are arguably among the most complex products of human intellect. The complexity of software has led to implementation methodologies that divide a problem into manageable parts and compose the parts to form the final product. Several research efforts have argued that C++ templates (a powerful parameterization mechanism) can be used to perform this division elegantly.

In particular, the work of VanHilst and Notkin [29][30][31] showed how one can implement *collaboration-based* (or *role-based*) designs using a certain templatized class pattern, known as a *mixin class* (or just *mixin*). Compared to other techniques (e.g., a straightforward use of *application frameworks* [17]) the VanHilst and Notkin method yields less redundancy and reusable components that reflect the structure of the design. At the same time, unnecessary dynamic binding can be eliminated, resulting into more efficient implementations. Unfortunately, this method resulted in very complex parameterizations, causing its inventors to question its scalability.

The *mixin layers* technique was invented to address these concerns. Mixin layers are mixin classes nested in a pattern such that the parameter (superclass) of the outer mixin determines the parameters (superclasses) of inner mixins. In previous work [4][24][25], we showed how mixin layers solve the scalability problems of the VanHilst and Notkin method and result into elegant implementations of collaboration-based designs.

[1)] We gratefully acknowledge the sponsorship of Microsoft Research, the Defense Advanced Research Projects Agency (Cooperative Agreement F30602-96-2-0226), and the University of Texas at Austin Applied Research Laboratories.

G. Butler and S. Jarzabek (Eds.): GCSE 2000, LNCS 2177, pp. 163-177, 2001.

This paper discusses practical issues related to mixin-based programming. We adopt a viewpoint oriented towards C++ implementations, but our discussion is not geared towards the C++ expert. Instead, we aim to document common problems and solutions in C++ mixin writing for the casual programmer. Additionally, we highlight issues that pertain to language design in general (e.g., to Java parameterization or to the design of future languages). Most of the issues clearly arise from the interaction of C++ features with the constructs under study. The discussion mainly stems from actual experience with C++ mixin-based implementations but a few points are a result of close examination of the C++ standard, since they refer to features that no compiler we have encountered implements. Even though we present an introduction to mixins, mixin layers, and their uses, the primary purpose of this paper is *not* to convince readers of the value of these constructs. (The reader should consult [4], [24], [25], [26], or [29] for that.)

We believe that the information presented here represents a valuable step towards moving some powerful programming techniques into the mainstream. We found that the mixin programming style is quite practical, as long as one is aware of the possible interactions with C++ idiosyncrasies. As C++ compilers move closer to sophisticated template support (e.g., some compilers already support separate template compilation) the utility of such techniques will increase rapidly.

2 Background (Mixins and Mixin Layers)

The term *mixin class* (or just *mixin*) has been overloaded in several occasions. Mixins were originally explored in the Lisp language with object systems like Flavors [20] and CLOS [18]. In these systems, mixins are an idiom for specifying a class and allowing its superclass to be determined by *linearization* of multiple inheritance. In C++, the term has been used to describe classes in a particular (multiple) inheritance arrangement: as superclasses of a single class that themselves have a common *virtual base class* (see [28], p.402). (This is ***not*** the meaning that we will use in this paper.) Both of these mechanisms are approximations of a general concept described by Bracha and Cook [6]. The idea is simple: we would like to specify an extension without pre-determining what exactly it can extend. This is equivalent to specifying a subclass while leaving its superclass as a parameter to be determined later. The benefit is that a single class can be used to express an incremental extension, valid for a variety of classes.

Mixins can be implemented using parameterized inheritance. The superclass of a class is left as a parameter to be specified at instantiation time. In C++ we can write this as:

```
template <class Super>
class Mixin : public Super {
  ... /* mixin body */
};
```

To give an example, consider a mixin implementing *operation counting* for a graph. Operation counting means keeping track of how many nodes and edges have been

visited during the execution of a graph algorithm. (This simple example is one of the non-algorithmic refinements to algorithm functionality discussed in [33]). The mixin could have the form:

```
template <class Graph>
class Counting: public Graph {
  int nodes_visited, edges_visited;
public:
  Counting() : nodes_visited(0), edges_visited(0), Graph() { }
  node succ_node (node v) {
    nodes_visited++;
    return Graph::succ_node(v);
  }
  edge succ_edge (edge e) {
    edges_visited++;
    return Graph::succ_edge(e);
  }
  ...
};
```

By expressing operation counting as a mixin we ensure that it is applicable to many classes that have the same interface (i.e., many different kinds of graphs). We can have, for instance, two different compositions:

```
  Counting< Ugraph > counted_ugraph;
```

and

```
  Counting< Dgraph > counted_dgraph;
```

for undirected and directed graphs. (We omit parameters to the graph classes for simplicity.) Note that the behavior of the composition is exactly what one would expect: any methods not affecting the counting process are exported (inherited from the graph classes). The methods that do need to increase the counts are "wrapped" in the mixin.

VanHilst and Notkin demonstrated that mixins are beneficial for a general class of object-oriented designs [29]. They used a mixin-based approach to implement *collaboration-based* (a.k.a. *role-based*) designs [5][15][16][21][29]. These designs are based on the view that objects are composed of different roles that they play in their interaction with other objects. The fundamental unit of functionality is a protocol for this interaction, called a *collaboration*. The mixin-based approach of VanHilst and Notkin results in efficient implementations of role-based designs with no redundancy. Sometimes, however, the resulting parameterization code is quite complicated—many mixins need to be composed with others in a complex fashion. This introduces scalability problems (namely, extensions that instantiate template parameters can be of length exponential to the number of mixins composed—see [24]). To make the approach more practical by reducing its complexity, *mixin layers* were introduced. Because mixin layers are an incremental improvement of the VanHilst and Notkin method, we only discuss implementing collaboration-based designs using mixin layers.

Mixin layers [24][25][26] are a particular form of mixins. They are designed with the purpose of encapsulating refinements for multiple classes. Mixin layers are nested mixins such that the parameter of an outer mixin determines the parameters of inner mixins. The general form of a mixin layer in C++ is:

```
template <class NextLayer>
class ThisLayer : public NextLayer {
public:
   class Mixin1 : public NextLayer::Mixin1 { ... };
   class Mixin2 : public NextLayer::Mixin2 { ... };
   ...
};
```

Mixin layers are a result of the observation that a conceptual unit of functionality is usually neither one object nor parts of an object—a unit of functionality may span several different objects and specify refinements (extensions) to all of them. All such refinements can be encapsulated in a single mixin layer and the standard inheritance mechanism can be used for composing extensions.

This property of mixin layers makes them particularly attractive for implementing collaboration-based designs. Each layer captures a single collaboration. Roles for all classes participating in a collaboration are represented by inner classes of the layer. Inheritance works at two different levels. First, a layer can inherit entire classes from its superclass (i.e., the parameter of the layer). Second, inner classes inherit members (variables, methods, or even other classes) from the corresponding inner classes in the superclass layer. This dual application of inheritance simplifies the implementation of collaboration-based designs, while preserving the benefits of the VanHilst and Notkin method. An important source of simplifications is that inner classes of a mixin layer can refer unambiguously to other inner classes—the layer acts as a namespace.

We illustrate our point with an example (presented in detail in [24]) of a collaboration-based design and its mixin layers implementation. (Full source code is available, upon request.) This example presents a graph traversal application and was examined initially by Holland [16] and subsequently by VanHilst and Notkin [29]. This application defines three different algorithms on an undirected graph, all implemented using a depth-first traversal: *Vertex Numbering* numbers all nodes in the graph in depth-first order, *Cycle Checking* examines whether the graph is cyclic, and *Connected Regions* classifies graph nodes into connected graph regions. The application has three distinct classes: *Graph*, *Vertex*, and *Workspace*. The *Graph* class describes a container of nodes with the usual graph properties. Each node is an instance of the *Vertex* class. Finally, the *Workspace* class includes the application part that is specific to each graph operation. For the *VertexNumbering* operation, for instance, a *Workspace* object holds the value of the last number assigned to a vertex as well as the methods to update this number.

As shown in Fig 1, we can decompose this application into five independent collaborations—one encompassing the functionality of an undirected graph, another encod-

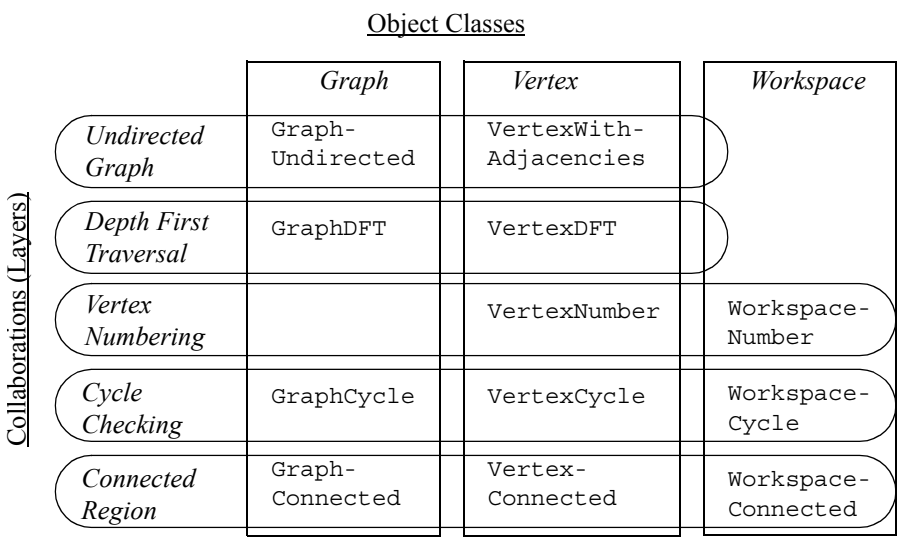


Fig 1: Collaboration decomposition of the example application: A depth-first traversal of an undirected graph is specialized to yield three different graph operations. Ovals represent collaborations, rectangles represent classes, their intersections represent roles.

ing depth-first traversals, and three containing the specifics of each graph algorithm (vertex numbering, cycle checking, and connected regions). Note that each collaboration captures a distinct aspect of the application and each object may participate in several aspects. That is to say, each object may play several roles. For instance, the role of a *Graph* object in the "*Undirected Graph*" collaboration supports storing and retrieving a set of vertices. The role of the same object in the "*Depth First Traversal*" collaboration implements a part of the actual depth-first traversal algorithm.

By implementing collaborations as mixin layers, the modular design of Fig 1 can be maintained at the implementation level. For instance, the "*Vertex Numbering*" collaboration can be implemented using a layer of the general form:

```
template <class Next>
class NUMBER : public Next {
public:
  class Workspace : public Next::Workspace {
    ... // Workspace role members
  };
  class Vertex : public Next::Vertex {
    ... // Vertex role members
  };
};
```

Note that no role (nested class) is prescribed for `Graph`. A `Graph` class is inherited from the superclass of `Number` (the class denoted by parameter `Next`).

As shown in [24], such components are flexible and can be reused and interchanged. For instance, the following composition builds `Graph`, `Vertex`, and `WorkSpace` classes nested inside class `CycleC` that implement vertex numbering of undirected graphs using a depth-first traversal:[2)]

```
typedef DFT < NUMBER < DEFAULTW < UGRAPH > > > CycleC;
```

By replacing `NUMBER` with other mixin layers we get the other two graph algorithms discussed. Many more combinations are possible. We can use the templates to create classes that implement more than one algorithm. For instance, we can have an application supporting both vertex numbering *and* cycle checking on the same graph by refining two depth-first traversals in order:

```
typedef DFT < NUMBER < DEFAULTW < UGRAPH > > > NumberC;
typedef DFT < CYCLE < NumberC > > CycleC;
```

Furthermore, all the characteristics of an undirected graph are captured by the `UGRAPH` mixin layer. Hence, it is straightforward to apply the same algorithms to a directed graph (mixin layer `DGRAPH` interchanged for `UGRAPH`):[3)]

```
typedef DFT < NUMBER < DEFAULTW < DGRAPH > > > NumberC;
```

This technique (of composing source components in a large number of combinations) underlies the *scalable libraries* [3] design approach for source code plug-and-play components.

3 Programming with C++ Mixins: Pragmatic Considerations

Since little has been written about the pragmatics of doing component programming using C++ mixins (mixin classes or mixin layers), we feel it is necessary to discuss some pertinent issues. Most of the points raised below concern fine interactions between the mixin approach and C++ idiosyncrasies. Others are implementation suggestions. They are all useful knowledge before one embarks into a development effort using C++ mixins and could serve to guide design choices for future parameterization mechanisms in programming languages. The C++ aspects we discuss are well-documented and other C++ programmers have probably also made some of our observations. Nevertheless, we believe that most of them are non-obvious and many only arise in the context of component programming—that is, when a mixin is designed and used in complete isolation from other components of the system.

Lack of Template Type-Checking. Templates do not correspond to types in the C++ language. Thus, they are not type-checked until instantiation time (that is, composi-

---

[2)] The `DEFAULTW` mixin layer is an implementation detail, borrowed from the VanHilst and Notkin implementation [29]. It contains an empty `WorkSpace` class and its purpose is to avoid dynamic binding by changing the order of composition.

[3)] This is under the assumption that the algorithms are still valid for directed graphs as is the case with the original code for this example [16].

tion time for mixins). Furthermore, methods of templatized classes are themselves considered function templates.[4] Function templates in C++ are instantiated automatically and only when needed. Thus, even after mixins are composed, not all their methods will be type-checked (code will only be produced for methods actually referenced in the object code). This means that certain errors (including type mismatches and references to undeclared methods) can only be detected with the right template instantiations and method calls. Consider the following example:

```
template <class Super>
class ErrorMixin : public Super {
public:
  ...
  void sort(FOO foo) {
    Super::srot(foo);   // misspelled
  }
};
```

If client code never calls method `sort`, the compiler will *not* catch the misspelled identifier above. This is true even if the `ErrorMixin` template is used to create classes, and methods other than `sort` are invoked on objects of those classes.

Delaying the instantiation of methods in template classes can be used to advantage, as we will see later. Nevertheless, many common designs are such that all member methods of a template class should be valid for all instantiations. It is not straightforward to enforce the latter part ("for *all* instantiations") but for most practical purposes checking all methods for a single instantiation is enough. This can be done by explicit instantiation of the template class, which forces the instantiation of all its members. The idiom for explicit instantiation applied to our above example is:

```
template class ErrorMixin<SomeFoo>;
```

When "Subtype of" Does not Mean "Substitutable for". There are two instances where inheritance may not behave the way one might expect in C++. First, constructor methods are not inherited. Ellis and Stroustrup ([13], p.264) present valid reasons for this design choice: the constructor of a superclass does not suffice for initializing data members added by a subclass. Often, however, a mixin class may be used only to enrich or adapt the method interface of its superclasses *without* adding data members. In this case it would be quite reasonable to inherit a constructor, which, unfortunately, is not possible. The practical consequence of this policy is that the only constructors that are visible in the result of a mixin composition are the ones present in the outer-most mixin (bottom-most class in the resulting inheritance hierarchy). To make matters worse, constructor initialization lists (e.g.,

```
constr() : init1(1,2), init2(3) {}  )
```

can only be used to initialize direct parent classes. In other words, all classes need to know the interface for the constructor of their direct superclass (if they are to use

[4] This wording, although used by the father of C++—see [28], p.330—is not absolutely accurate since there is no automatic type inference.

constructor initialization lists). Recall, however, that a desirable property for mixins is that they be able to act as components: a mixin should be implementable in isolation from other parts of the system in which it is used. Thus a single mixin class should be usable with several distinct superclasses and should have as few dependencies as possible. In practice, several mixin components are just thin wrappers adapting their superclass's interface.

A possible workaround for this problem is to use a standardized construction interface. A way to do this is by creating a construction class encoding the union of all possible arguments to constructors in a hierarchy. Then a mixin "knows" little about its direct superclass, but has dependencies on the union of the construction interfaces for all its possible parent classes. (Of course, another workaround is to circumvent constructors altogether by having separate initialization methods. This, however, requires a disciplined coding style to ensure that methods are always called after object construction.) As a side-note, destructors for base classes are called automatically so they should not be replicated.

Synonyms for Compositions. In the past sections we have used `typedefs` to introduce synonyms for complicated mixin compositions—e.g.,

```
typedef A < B < C > > Synonym;
```

Another reasonable approach would be to introduce an empty subclass:

```
class Synonym : public A < B < C > > { };
```

The first form has the advantage of preserving constructors of component `A` in the synonym. The second idiom is cleanly integrated into the language (e.g., can be templatized, compilers create short link names for the synonym, etc.). Additionally, it can solve a common problem with C++ template-based programming: generated names (template instantiations) can be extremely long, causing compiler messages to be incomprehensible.

Designating Virtual Methods. Sometimes C++ policies have pleasant side-effects when used in conjunction with mixins. An interesting case is that of a mixin used to create classes where a certain method can be virtual or not, depending on the concrete class used to instantiate the mixin. This is due to the C++ policy of letting a superclass declare whether a method is virtual, while the subclass does not need to specify this explicitly. Consider a regular mixin and two concrete classes instantiating it (a C++ `struct` is a class whose members are public by default):

```
template <class Super>
struct MixinA : public Super {
  void virtual_or_not(FOO foo) { ... }
};

struct Base1 {
  virtual void virtual_or_not(FOO foo) { ... }
  ... // methods using "virtual_or_not"
};
```

```
struct Base2 {
  void virtual_or_not(FOO foo) { ... }
};
```

The composition `MixinA<Base1>` designates a class in which the method `virtual_or_not` is virtual. Conversely, the same method is not virtual in the composition `MixinA<Base2>`. Hence, calls to `virtual_or_not` in `Base1` will call the method supplied by the mixin in the former case but not in the latter.

In the general case, this phenomenon allows for interesting mixin configurations. Classes at an intermediate layer may specify methods and let the inner-most layer decide whether they are virtual or not.

As we recently found out, this technique was described first in [12].

Single Mixin for Multiple Uses. The lack of template type-checking in C++ can actually be beneficial in some cases. Consider two classes `Base1` and `Base2` with very similar interfaces (except for a few methods):

```
struct Base1 {
  void regular() { ... }
  ...
};
struct Base2 {
  void weird() { ... }
  ... // otherwise same interface as Base1
};
```

Because of the similarities between `Base1` and `Base2`, it makes sense to use a single mixin to adapt both. Such a mixin may need to have methods calling either of the methods specific to one of the two base classes. This is perfectly feasible. A mixin can be specified so that it calls either `regular` or `weird`:

```
template <class Super>
class Mixin : public Super {
  ...
public:
  void meth1() { Super::regular(); }
  void meth2() { Super::weird(); }
};
```

This is a correct definition and it will do the right thing for both composition `Mixin<Base1>` and `Mixin<Base2>`! What is remarkable is that part of `Mixin` seems invalid (calls an undefined method), no matter which composition we decide to perform. But, since methods of class templates are treated as function templates, no error will be signalled unless the program actually uses the wrong method (which may be `meth1` or `meth2` depending on the composition). That is, an error will be signalled only if the program is indeed wrong. We have used this technique to provide uniform, componentized extensions to data structures supporting slightly dif-

ferent interfaces (in particular, the red-black tree and hash table of the SGI implementation of the Standard Template Library [22]).

Propagating Type Information. An interesting practical technique (also applicable to languages other than C++) can be used to propagate type information from a subclass to a superclass, when both are created from instantiating mixins. This is a common problem in object-oriented programming. It was, for instance, identified in the design of the P++ language [23] (an extension of C++ with constructs for component-based programming) and solved with the addition of the `forward` keyword. The same problem is addressed in other programming languages (e.g., Beta [19]) with the concept of *virtual types*.

Consider a mixin layer encapsulating the functionality of an allocator. This component needs to have type information propagated to it from its subclasses (literally, the subclasses of the class it will create when instantiated) so that it knows what kind of data to allocate. (We also discussed this example in detail in [25] but we believe that the solution presented here is the most practical way to address the problem.) The reason this propagation is necessary is that subclasses may need to add data members to a class used by the allocator. One can solve the problem by adding an extra parameter to the mixin that will be instantiated with the final product of the composition itself. In essence, we are reducing a conceptual cycle in the parameterization to a single self-reference (which is well-supported in C++). This is shown in the following code fragment:

```
template <class EleType, class FINAL>
class ALLOC {
public:
  class Node {
    EleType element;  // stored data type
  public:
    ...  // methods using stored data
  };

  class Container {
  protected:
    FINAL::Node* node_alloc() {
      return new FINAL::Node();
    }
    ... // Other allocation methods
  };
};

template <class Super>
class BINTREE : public Super {
public:
  class Node : public Super::Node {
    Node* parent_link, left_link, right_link ;
  public:
    ...   // Node interface
  };
```

```
  class Container : public Super::Container {
    Node* header; // Container data members
  public:
    ...           // Interface methods
  };
};

class Comp : public BINTREE < ALLOC <int, Comp> > {/* empty */
};
```

Note what is happening in this code fragment (which is abbreviated but preserves the structure of actual code that we have used). A binary tree data structure is created by composing a `BINTREE` mixin layer with an `ALLOC` mixin layer. The data structure stores integer (`int`) elements. Nevertheless, the actual type of the element stored is *not* `int` but a type describing the node of a binary tree (i.e., an integer together with three pointers for the parent, and the two children of the node). This is the type of element that the allocator should reserve memory for.

The problem is solved by passing the final product of the composition as a parameter to the allocator mixin. This is done through the self-referential (or *recursive*) declaration of class `Comp`. (Theoretically-inclined readers will recognize this as a *fixpoint* construction.) Note that `Comp` is just a synonym for the composition and it has to use the synonym pattern introducing a class (i.e., the `typedef` synonym idiom discussed earlier would not work as it does not support recursion).

It should be noted that the above recursive construction has been often used in the literature. In the C++ world, the technique was introduced by Barton and Nackman [2] and popularized by Coplien [9]. Nevertheless, the technique is not mixin-specific or even C++-specific. For instance, it was used by Wadler, Odersky and the first author [32] in Generic Java [7] (an extension of Java with parametric polymorphism). The origins of the technique reach back at least to the development of F-bounded polymorphism [8].

Hygienic Templates in the C++ Standard. The C++ standard ([1], section 14.6) imposes several rules for name resolution of identifiers that occur inside templates. The extent to which current compilers implement these rules varies, but full conformance is the best approach to future compatibility for user code.

Although the exact rules are complicated, one can summarize them (at loss of some detail) as “templates cannot contain code that refers to ‘nonlocal’ variables or methods”. Intuitively, “nonlocal” denotes variables or methods that do not depend on a template parameter and are not in scope at the global point closest to the template definition. This rule prevents template instantiations from capturing arbitrary names from their instantiation context, which could lead to behavior not predicted by the template author.

A specific rule applies to mixin-based programming. To quote the C++ standard (14.6.2), "if a base class is a dependent type, a member of that class cannot hide a name declared within a template, or a name from the templates enclosing scopes". Consider the example of a mixin calling a method defined in its parameter (i.e., the superclass of the class it will create when instantiated):

```
struct Base {
  void foo() { ... }
};

void foo() { }

template <class Super>
struct Mixin : public Super {
  void which_one()   { foo(); } // ::foo
};

Mixin < Base > test;
```

That is, the call to `foo` from method `which_one` will refer to the global `foo`, not the `foo` method of the `Base` superclass.

The main implication of these name resolution rules is on the way template-based programs should be developed. In particular, imagine changing a *correct* class definition into a mixin definition (by turning the superclass into a template parameter). Even if the mixin is instantiated with its superclass in the original code, the new program is *not* guaranteed to work identically to the original, because symbols may now be resolved differently. This may surprise programmers who work by creating concrete classes and turning them into templates when the need for abstraction arises. To avoid the potential for insidious bugs, it is a good practice to explicitly qualify references to superclass methods (e.g., `Super::foo` instead of just `foo`).

Compiler Support. Most C++ compilers now have good support for parameterized inheritance (the technique we used for mixins) and nested classes. We have encountered few problems and mostly with older compilers when programming with C++ mixins. In fact, most of the compiler dependencies are not particular to mixin-based programming but concern all template-based C++ programs. These include limitations on the debugging support, error checking, etc. We will not discuss such issues as they are time-dependent and have been presented before (e.g., [10]). Note, however, that mixin-based programming is not more complex than regular template instantiation and typically does not exercise any of the "advanced" features of C++ templates (type inference, higher-order templates, etc.). Overall, the compiler support issues involved in mixin-based programming are about the same as those arising in implementing the C++ Standard Template Library [27].

4 Related Work

Various pieces of related work have been presented in the previous sections. We cannot exhaustively reference all C++ template-based programming techniques, but we will discuss two approaches that are distinct from ours but seem to follow parallel courses.

The most prominent example is the various implementations of the STL. Such implementations often exercise the limits of template support and reveal interactions of C++ policies with template-based programming. Nevertheless, parameterized inheritance is not a part of STL implementations. Hence, the observations of this paper are mostly distinct from the conclusions drawn from STL implementation efforts.

Czarnecki and Eisenecker's generative programming techniques [10][11] were used in the Generative Matrix Computation Library (GMCL). Their approach is a representative of techniques using C++ templates as a programming language (that is, to perform arbitrary computation at template instantiation time). What sets their method apart from other template meta-programming techniques is that it has similar goals to mixin-based programming. In particular, Czarnecki and Eisenecker try to develop components which can be composed in multiple ways to yield a variety of implementations. Several of the remarks in this paper are applicable to their method, even though their use of mixins is different (for instance, they do not use mixin layers).

5 Conclusions

We presented some pragmatic issues pertaining to mixin-based programming in C++. We believe that mixin-based techniques are valuable and will become much more widespread in the future. Mixin-based programming promises to provide reusable software components that result into flexible and efficient implementations.

Previous papers have argued for the value of mixin-based software components and their advantages compared to application frameworks. In this paper we tried to make explicit the engineering considerations specific to mixin-based programming in C++. Our purpose is to inform programmers of the issues involved in order to help move mixin-based programming into the mainstream.

References

[1] ANSI / ISO Standard: *Programming Languages—C++*, ISO/IEC 14882, 1998.

[2] J. Barton and L.R. Nackman, *Scientific and Engineering C++: An Introduction with Advanced Techniques and Applications*, Addison-Wesley, 1994.

[3] D. Batory, V. Singhal, M. Sirkin, and J. Thomas, "Scalable Software Libraries", *ACM SIGSOFT* 1993.

[4] D. Batory, R. Cardone, and Y. Smaragdakis, "Object-Oriented Frameworks and Product-Lines", *1st Software Product-Line Conference*, Denver, Colorado, August 1999.

[5] K. Beck and W. Cunningham, "A Laboratory for Teaching Object-Oriented Thinking", *OOPSLA 1989*, 1-6.

[6] G. Bracha and W. Cook, "Mixin-Based Inheritance", *ECOOP/OOPSLA 90*, 303-311.

[7] G. Bracha, M. Odersky, D. Stoutamire and P. Wadler, "Making the future safe for the past: Adding Genericity to the Java Programming Language", *OOPSLA 98*.

[8] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell, "F-bounded Polymorphism for Object-Oriented Programming", in *Proc. Conf. on Functional Programming Languages and Computer Architecture*, 1989, 273-280.

[9] J. Coplien, "Curiously Recurring Template Patterns", *C++ Report*, 7(2):24-27, Feb. 1995.

[10] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[11] K. Czarnecki and U. Eisenecker, "Synthesizing Objects", *ECOOP 1999*, 18-42.

[12] U. Eisenecker, "Generative Programming in C++", in *Proc. Joint Modular Languages Conference (JMLC'97)*, LNCS 1204, Springer, 1997, 351-365.

[13] M.A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[15] R. Helm, I. Holland, and D. Gangopadhyay, "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems". *OOPSLA 1990*, 169-180.

[16] I. Holland, "Specifying Reusable Components Using Contracts", *ECOOP 1992*, 287-308.

[17] R. Johnson and B. Foote, "Designing Reusable Classes", *J. of Object-Oriented Programming*, 1(2): June/July 1988, 22-35.

[18] G. Kiczales, J. des Rivieres, and D. G. Bobrow, *The Art of the Metaobject Protocol*. MIT Press, 1991.

[19] O.L. Madsen, B. Møller-Pedersen, and K. Nygaard, *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

[20] D.A. Moon, "Object-Oriented Programming with Flavors", *OOPSLA 1986*.

[21] T. Reenskaug, E. Anderson, A. Berre, A. Hurlen, A. Landmark, O. Lehne, E. Nordhagen, E. Ness-Ulseth, G. Oftedal, A. Skaar, and P. Stenslet, "OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems", *J. of Object-Oriented Programming*, 5(6): October 1992, 27-41.

[22] Silicon Graphics Computer Systems Inc., *STL Programmer's Guide*. See: `http://www.sgi.com/Technology/STL/` .

[23] V. Singhal, *A Programming Language for Writing Domain-Specific Software System Generators,* Ph.D. Dissertation, Dep. of Computer Sciences, University of Texas at Austin, August 1996.

[24] Y. Smaragdakis and D. Batory, "Implementing Reusable Object-Oriented Components". In the *5th Int. Conf. on Software Reuse (ICSR 98)*.

[25] Y. Smaragdakis and D. Batory, “Implementing Layered Designs with Mixin Layers”. In *ECOOP 98*.

[26] Y. Smaragdakis, “Implementing Large-Scale Object-Oriented Components”, Ph.D. Dissertation, Department of Computer Sciences, University of Texas at Austin, December 1999.

[27] A. Stepanov and M. Lee, “The Standard Template Library”. Incorporated in ANSI/ISO Committee C++ Standard.

[28] B. Stroustrup, *The C++ Programming Language, 3rd Ed.*, Addison-Wesley, 1997.

[29] M. VanHilst and D. Notkin, “Using C++ Templates to Implement Role-Based Designs”. *JSSST International Symposium on Object Technologies for Advanced Software*, Springer-Verlag, 1996, 22-37.

[30] M. VanHilst and D. Notkin, “Using Role Components to Implement Collaboration-Based Designs”. *OOPSLA 1996*.

[31] M. VanHilst and D. Notkin, “Decoupling Change From Design”, *SIGSOFT* 96.

[32] P. Wadler, M. Odersky and Y. Smaragdakis, “Do Parametric Types Beat Virtual Types?”, unpublished manuscript, posted in October 1998 in the Java Genericity mailing list `(java-genericity@cs.rice.edu)`.

[33] K. Weihe, “A Software Engineering Perspective on Algorithmics”, available at `http://www.informatik.uni-konstanz.de/Preprints/` .

Metaprogramming in the Large

Andreas Ludwig and Dirk Heuzeroth

Institut für Programmstrukturen und Datenorganisation
Universität Karlsruhe
{ludwig,heuzer}@ipd.info.uni-karlsruhe.de

Abstract. Software evolution demands continuous adaptation of software systems to continuously changing requirements. Our goal is to cope with software evolution by automating program transformation and system reconfiguration. We show that this can be achieved with a static metaprogramming facility and a library of suitable metaprograms. We show that former approaches of program transformations are not sufficient for large object oriented systems and outline two base transformations that fill the gap.

1 Introduction

Many software systems have grown enormously large in the last decades. Requirements for these systems have changed continuously, and still do. Consequently, the software needs to be adapted all the time: software evolves. However, changing software is a complex and error prone task and is likely to produce high costs. One reason for this are lots of dependencies in typical programs, which make them hard to understand. Moreover, the effects of changes to a system are hard to predict. This is especially true for large changes like system restructuring, addition, removal and exchange of components and subsystems as well as composition of components. But even small modifications like changing the signature of a single method may show effects throughout the whole system: for instance, a changed method parameter type can not only influence all references to that method but also all references to overloaded methods which in turn might be defined in any subclass. To deal with those effects, we require global information about the program and must hence operate in the large. We provide further examples in sections 2 and 3.

Software evolution means transforming programs. Our group developed a static metaprogramming facility called COMPOST that provides a library of useful analyses and transformations. We found metaprogramming to be a natural and very promising strategy to carry out changes consistently and thus support both software evolution and software construction.

To cope with software evolution and construction, software must be easy to change. We therefore focus on software changes and start with a concrete scenario, the evolution from JDK 1.0 to JDK 1.3 (section 2). After that, we identify some program transformations that are needed for our purposes, especially to deal with imperative object oriented programs (section 3). We then outline the

G. Butler and S. Jarzabek (Eds.): GCSE 2000, LNCS 2177, pp. 178–187, 2001.

infrastructure needed to actually perform those changes (section 4). Finally, we discuss related work and possible uses of metaprogramming in these areas, which we cannot cover in detail here (section 5). We conclude with our experiences and suggestions for future work (section 6).

2 Software Evolution Scenario

Assume that a new version of a component offers improved behaviour, but also comes with a changed interface. In order to use the revised version, all using contexts of this component have to be updated. Now assume that the programs that have been using the old component are also maintained. Then, binary compatibility is not sufficient. Instead, the programs must be brought to the state of the art at source level.

Example 1 (JDK Update). There have been about 300 changes in the API of the standard Java libraries up to now — a complete list is available at `http://java.sun.com/j2se/1.3/docs/api/deprecated-list.html`. These interface changes affect all pieces of Java software that rely on the now "deprecated" features; we classify these changes as follows and will study them in detail later on:

① Local changes of signatures (name or parameters), e. g. in java.awt.Component:
- move(int, int) → setLocation(int, int)
- disable() → setEnabled(false)
- add(String, Component) → add(Component, Object)

② Change of the location of features, e.g. in java.awt:
- Frame.CROSSHAIR_CURSOR → Cursor.CROSSHAIR_CURSOR
- Frame.getCursorType() → Component.getCursor().getType()

③ Removal or replacement of features, e.g.
- java.awt.Component.getPeer() → ∅ (deleted)
- java.io.StringBufferInputStream → java.io.StringReader

④ Change of an entire framework, using multiple design patterns, such as the new event handling framework that replaces a template method based solution by an observer pattern, which concerns some dozen classes in several packages.

How can we automate an update for a large system? The solution is at hands: Let the programmer of the component write a "smart patch" metaprogram that is able to update arbitrary programs.

The scenario illustrates what we consider metaprogramming in the large. In this setting, we have a big advantage, as some of the semantics of the system to change is known. Assuming that the old versions of the components have been used correctly allows to assert many predicates that would have been very hard to find out by analyses, such as object lifetime dependencies. After some basic transformations have been detailed, we illustrate how this knowledge can be used in the second part of the example.

3 New Program Transformations

Program transformations have been investigated for years (for a good overview see [CE00]). We discuss how goals and languages have changed since then and now require further transformations. We then sketch an abstract program model to deal with object oriented languages, and finally discuss two fundamental transformations based upon that model.

3.1 Classical Program Transformations Improved

The primary research focus of classical program transformations was program refinement and optimisation, as well as the creation of extensible languages. The most important achievements are a set of base transformations such as *add definition*, *fold*, *unfold*, *function generalisation* and *specialisation*, and techniques to resolve recursions. These transformations were based upon a functional programming model and local contexts only, to avoid the problems introduced by *states* and *scopes*. However, states and scopes are inherent parts of imperative object oriented languages. While the known base transformations reappear in more specific forms to suit the rules of the concrete language, further *base transformations* are required to cover states and scopes.

Transformations come with a set of *constraints* restricting their use, e. g. prohibition of ambiguous names. Single transformations often lack constructive strategies how to fulfil these conditions; usually, only a combination of transformations is applicable. Thus, the goal is to collect sufficient information about the context to derive applicable chains of transformations to achieve the desired effects.

3.2 Nested Feature Model for Object Oriented Languages

The notions of component interfaces and scopes are somewhat dependent on the concrete language. Therefore, we introduce an abstract program model consisting of hierarchically nested *features* such as classes, variables, and methods, which may define a scope (see Figure 1). We may regard packages, constructors, attributes or parameters as special cases of classes, methods, or variables. To complete this simplistic model, we distinguish *declarations* of features and *references* to features which are contained within a scope, and assume that each feature has at most one definition, but arbitrarily many references. The exact rules for feature declarations, feature nesting, and feature reference resolution (including feature visibilities, overloading, and inheritance) are part of the language specification. In our experiments, we used Java as a representative object oriented language.

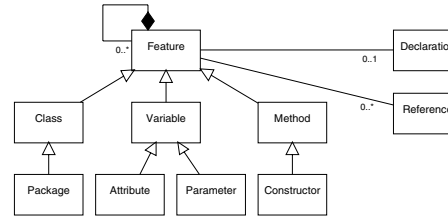


Fig. 1. Simple Nested Feature Model.

3.3 A Transformation to Deal with States

The feature model has no visible states to manipulate directly. However, the state transitions defined by variable and method accesses can be used to bring in new states. These accesses may be embedded into arbitrary expression terms. To avoid side effects in the partial expressions that are evaluated prior to the feature access, we need a transformations that shifts these parts and *flattens* the syntax tree of the expression term. The following example illustrates this.

Example 2 (Expression Flattening). Consider the expression `x = f(g(y), h(k(z))` and assume we want to insert some code before the very moment when `f` is accessed. The transformation should produce: `int i = g(y); boolean b = h(k(z)); /* some new code */ x = f(i, b);` — now the reference to `f` is the first expression that is evaluated in the statement. Note that the types of `g` and `h` cannot be derived from the original expression, and that it was not necessary to flatten `k`.

The transformation takes a reference to a variable or method as argument and flattens the top-level expression term this reference is contained within:

1. Collect all expressions that become evaluated before the relevant feature access and that may have side effects.
2. For each largest expression `e` in the order of evaluation:
   (a) Calculate the type τ of `e`
   (b) Create a new unambiguous temporary variable `v` of type τ and insert the declaration of `v` in front of the the top-level expression.
   (c) Insert `e` as initialisation code of the variable.
   (d) Substitute the original occurrence of `e` with a new reference to `v`.

This code transformation allows to modify arbitrary accesses to features by simple insertion of new code. It is easy to see how it can be used for instrumentalisation or to exchange method calls by other means of communication.

Note that this transformation requires context sensitive information to derive the types of expressions, and to choose an unambiguous variable name.

3.4 A Transformation to Deal with Scopes

The most general and basic transformation that is able to reconfigure interfaces is *move feature declarations*.[1] The transformation consists of two subtasks, which are also base transformations: creating new feature declarations in the new scope, and deletions of old declarations. Note that we explicitly allow inheritance, feature hiding, overloading, and combinations thereof.

1. Insert feature declarations in the new scope
   (a) Check if a declaration is forbidden due to ambiguous names in the new scope, or incompatible modifiers or signatures while attempting to overwrite a feature of identical name in a superclass. If so, rename the feature, or optionally change modifiers in the superclass.

[1] Special cases of this transformation are already depicted in [FBB+99] as refactorings. The version presented here is more general and more detailed.

   (b) Check if a definition hides less general definitions that are overloaded or overwritten. If so, rename the new features, or strengthen qualification of references to the hidden feature (add type casts or access qualifiers).
2. Remove feature declarations from the old scope
   (a) Check if there are references from the moved features to other features in the old scope. If so, redirect them to the old scope using a proper access path. If no predefined access path is available, create a new one by adding an additional parameter to each feature. If necessary, adapt visibilities of the used features to allow access from the new scope. Alternatively, compute a self-dependent *feature group* and ask the user if this group shall be moved instead.[2]
   (b) Check if there are external references to the features in the old scope. If so, redirect them to the new scope, by using a proper access path.

This transformation allows to reconfigure system interfaces arbitrarily. If one reads design patterns as instructions how to change a system rather than creating one from scratch, then many of them are built upon this transformation.

Finding access paths is a key issue in this transformation when the scopes are classes. In many cases, access paths are predefined, either by the caller of the transformation, or if features are global, or the scopes are nested or inherited (see Figure 2) — then, the underlying objects are either accessible from anywhere, identical, or share the same life time.

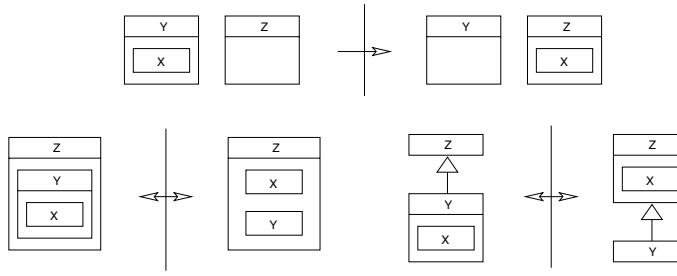


Fig. 2. Feature Movement Schemes – General Case and Two Specialisations (below) — Note that the admissible feature types of x, y and z may vary.

In the general case, no predefined access path is available. Sometimes, it is possible to identify a unique path candidate that is free of possible side effects. If there are several solutions, the choice is a genuine design decision and should be left to the user. If there is not even a possible candidate, the transformation can attach a new attribute and insert some initialisation code for the target object, if the initialisation is not ambiguous.

We learn two lessons from this example: First, user interaction is required when analyses fail and design alternatives must be chosen. Second, a non-trivial transformation requires a lot of subtasks which have to be addressed. Figure 3 illustrates some auxiliary functions.

[2] This is the reason why moving a set of declarations can be easier than a single one.

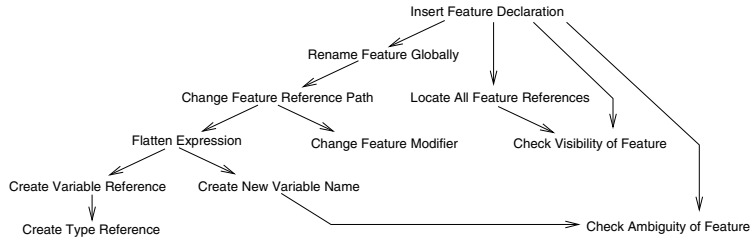


Fig. 3. Transformation Subtasks.

Example 3 (JDK Update Revisited). Equipped with the transformations just presented and additional knowledge about concrete components that we have to deal with, we can now try to find a metaprogram that would have saved enormous amounts of programming time...

① Local changes of signatures: Renaming is a task similar to addition of a new feature, while a change of parameters requires a change of arguments on the caller side — flattening the expressions that might have side effects is the most difficult part.

② Change of the location of features: The examples given are easy to handle with feature moves as we have to deal with global constants, or already know a proper replacement access path.

③ Removal or replacement of features: In the case of the deprecated Peer classes, there is no obvious update. However, the only proper use of the peers has been to check for consistent initialisation state. We can attempt to pattern match for c.getPeer() != null and replace this expression by c.isDisplayable(). The second example concerning the replacement of an improper Unicode stream, is also hard to resolve, as different data types (byte vs. char) are involved. But again, all meaningful uses should not rely on concrete bit counts, so a simple reassignment of methods to their counterparts is sufficient. Only one method features a different signature: read(char[],int,int) replaces read(byte[],int,int). Typical uses of the old version will have translated the byte array into a string or a character array anyway, but to ensure compatibility, we would have to insert some translation code. We have to attach some glue code to calls of the form in.read(arr, off, len) in order to obtain:

```
char[] tmp = new char[arr.length];
in.read(tmp, off, len);
System.arraycopy(new String(tmp, off, len).getBytes(), 0, arr, off, len);
```

There are some details to get right, such as adding a new statement block when necessary and getting rid of side effect expressions in access paths.

④ Change of an entire framework: Beginning with JDK 1.1, event handling changed to an observer pattern replacing redefined methods of the corresponding components. The following code snippets show a simple solution obtained by some feature moves. The former action method in class C

```
public boolean action(Event e, Object what) {
    // action code here
    return super.action(e, what); // or other expression
}
```

is transformed into a new attribute declaration in C

```
ActionListener actions = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        // user code from super.action goes here
        // action code here; 'what' redirected to e.getActionCommand()
        // no return value here - no need to consume events
    }
};
```

plus an additional call addActionListener(actions); in each constructor of C.

Example 3 shows that an automatic update is actually possible while human interaction is necessary only in few cases.

In the following section, we discuss the specific requirements and challenges for a technical infrastructure supporting the construction of metaprograms.

4 Object Oriented Metaprogramming Realized

The general procedure of source to source transformations is illustrated in Figure 4: A compiler frontend produces a metamodel from the sources, which includes all syntactic data necessary for the pretty printer to reproduce the sources later on. Transformations can use any derived information to change the syntactic parts.

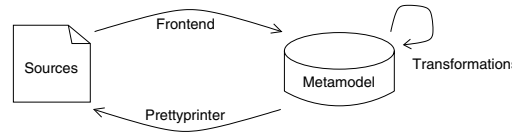


Fig. 4. Source Level Transformations.

4.1 Requirements for Metaprogramming Systems

Metaprograms must cover some important requirements, and these must be considered when supplying tools to facilitate the construction.

Readability: A metaprogramming system changes source codes. The result should obey coding conventions and should not destroy formatting. In this setting, blank spaces become important and must be protected. In general, "code bloat" should be kept to a minimum.

Scalability: In contrast to compilers, metaprogramming systems change the syntactic base of the metamodel during runtime. Hence, when used in an interactive and incremental environment, information derived from the syntactic model must also be updated incrementally. To retain scalability, it is not feasible to invalidate the whole model after a change. Instead, it is necessary to carefully analyse *change impacts* to prevent transitive propagation throughout the model. For instance, a change of a feature declaration may change the assignment of all references to the feature and all hidden ones.

Note that the smart patch scenario is not necessarily incremental — usually it is possible to do all necessary analyses in advance and delay the transformations.

Extensibility: There are lots of possible transformations motivated by high level design problems. A library of metaprograms is therefore unlikely to be complete, but it should be extensible. A proper system design should facilitate the creation of metaprograms. For instance, the metamodel interface should hide the process of information retrieval, offer navigational links, and provide meaningful abstractions over the concrete representations. Auxiliary transformations, analyses and code snippet generators should become part of the library.

4.2 Limitations of Metaprogramming Systems

Source level metaprogramming suffers from some general problems.

Source code hygiene and especially comment assignment are an issue. Comments will not always be assigned properly and may move with the wrong element. Also, it will not always be possible to avoid some redundant code.

In general, *comments and documentation* do not obey a formal semantics and hence cannot be maintained automatically. However, it is possible to automate changes of designated, formal parts. For instance, the Java documentation conventions require to tag feature references with `@see` or `@link` making them accessible for analysis and transformations.

Analyses are conservative and may fail to derive enough information to do certain decisions automatically. Then, interaction with the human programmer is needed. Sometimes an alternative approach can be found, which might be far from perfect, but functional — e. g. to insert a new attribute and ignore an uncertain candidate. Possible improvements could then be offered in an interactive session after the main transformations have taken place.

4.3 COMPOST - An Implementation

We have been constructing a metaprogramming system for the Java programming language called COMPOST (COMPOsition SysTem). The system is available from `http://i44www.info.uni-karlsruhe.de/~compost` and consists of nearly 600 classes. It contains a Java frontend providing facilities to resolve feature references and for cross reference information. The pretty printer is configurable to fit specific conventions and is able to reproduce (or improve upon) the original input and to insert new code seamlessly. We have built a library of auxiliary functions and transformations also containing the auxiliaries from Figure 3.

Transformations are responsible for the maintenance of the abstract syntax model, while the pretty printer handles the concrete syntax, and lookup services update the derived semantic information. Transformations report the syntactic effects of their changes, and this information is interpreted by a responsible service as soon as new semantic information is required.

4.4 Experiences

Memory consumption of a metaprogramming system is high, but does not exceed that of an optimising compiler, and seems to scale with the size of the program. We were able to analyse 3000 classes on a state of the art workstation.

Naive incremental model updates are sufficient for a handful of classes only, but more sophisticated algorithms should be able to handle a big number of classes, as some sparse experiments in literature have indicated [BS86] [Wag98].

We have chosen Java as our primary target language, but metaprogramming should be applicable for all programming languages. In general, languages providing many static predicates allow to do more decisions automatically based on the additional knowledge represented by these predicates.

5 Related Work

We already compared metaprogramming with program transformation systems in section 3. Now we discuss related work in the area of software evolution.

Extreme programming (XP) [Bec99] is a development process to support evolutionary programming. XP relies on testing to assure the correctness of changes. Verified metaprograms would no longer require additional tests.

The base mechanics of XP are refactorings [FBB+99]. Refactorings are program transformations that optimise for readability. To create sophisticated tooling, the strategies given in the catalogues are still too vague and heavily rely on the programmer. Compiler construction research can provide solutions for many problems, but we need a more formal background to do so. The feature movement transformation is an attempt to generalise many special cases.

While refactorings are too local to have a meaningful motivation by themselves, design patterns [GHJV95] are complete development processes leading from a single design problem down to implementations. In many cases, the pattern can be regarded as an instruction to change software, using a series of refactorings and similar transformations [Zim97]. If this is possible, then we can derive a metaprogram instantiating the pattern. For instance, moving features is fundamental to create new levels of indirection widely used in patterns.

Architectural systems [SDZ95] allow to exchange *connectors* representing a communication association. Similarly, composition languages [NM95] focus on the wiring of software components. Metaprograms can easily exchange connectors or compose programs avoiding special languages.

Generic programming allows to bind parameters embedded in the code with concrete syntactic content. Metaprogramming can simulate the instantiation of a parameter and even attempt to identify the replacement locations even if they were not tagged as parametric.

Aspect oriented programming [KIL+97] or multidimensional separation of concerns [OT99] intend to offer multiple views on the (abstract) program that might not be identical to the primary functional modularisation. Each aspect, or concern, can be changed separately, and must then be merged, or woven, with the other aspects to derive the final model. This weaving process can be implemented by a metaprogram.

In this work we concentrated on the support of transformations for refactoring software systems rather than the configuration of system families. The latter is subject of the generative programming paradigm [CE00].

6 Conclusion

We showed that program transformations realized by metaprograms significantly facilitate consistent and correct changes and that this approach is a well suited technique for software evolution. Furthermore, we demonstrated that metaprogramming fits seamlessly into emerging design techniques such as design patterns, separation of concerns, and refactorings.

We identified two important basic transformations that are necessary for reconfiguring large imperative object oriented systems. We also showed that metaprogramming is feasible for real world languages with the right tool support. We described the infrastructure necessary for source to source metaprogramming. COMPOST is an implementation of such an infrastructure.

Future work comprises the identification of further base transformations for object oriented programs as well as fast incremental model updates since there is still little experience in this area.

References

Bec99. Kent Beck. *extreme Programming explained*. Addison Wesley, Reading, MA, 1999.

BS86. Rolf Bahlke and Gregor Snelting. The PSG System: From Formal Language Definitions to Interactive Programming Environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, October 1986.

CE00. Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming*. Addison Wesley, 2000.

FBB+99. M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley, 1999.

GHJV95. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1995.

KIL+97. Gregor Kiczales, John Irwin, John Lamping, Jean-Marc Loingtier, Cristina Videira Lopes, Chris Maeda, and Anurag Mendhekar. Aspect-oriented Programming. In *ECOOP'97*, pages 220–242. Springer-Verlag, 1997.

NM95. Oscar Nierstrasz and Theo Dirk Meijler. Requirements for a composition language. In O. Nierstrasz P. Ciancarini and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, LNCS 924, pages 147–161. Springer-Verlag, 1995.

OT99. Harold Ossher and Peri Tarr. Multi-dimensional separation of concerns in hyperspace. Technical report, IBM T.J. Watson Research Center, 1999.

SDZ95. Mary Shaw, Robert DeLine, and Gregory Zelinski. Abstraction and Implementation for Architectural Connections. Technical report, CMU, November 1995.

Wag98. Tim A. Wagner. *Practical Algorithms for Incremental Software Development Environments*. Ph.D. thesis, Computer Science Division, EECS Department, University of California, March 1998.

Zim97. Walter Zimmer. *Frameworks und Entwurfsmuster*. PhD thesis, Universität Karlsruhe, February 1997.

Just When You Thought Your Little Language Was Safe: "Expression Templates" in Java

Todd L. Veldhuizen

Extreme Computing Laboratory
Indiana University Computer Science Department
Bloomington Indiana 47405, USA
tveldhui@acm.org

Abstract. Template techniques in C++ allow a modest degree of *generative programming*: creating specialized code for specialized problems. This use of templates has been controversial; indeed, one of the oft-cited reasons for migrating to Java is that it provides a simpler language, free of complexities such as templates. The essence of generative programming in C++ is not templates – the language feature – but rather the underlying algorithms in the compiler (template instantiation) which unintentionally resemble an optimization called *partial evaluation* [12,18]. By devising a partial evaluator for Java, we reproduce some of the generative programming aspects of C++ templates, without extending the Java language. The prototype compiler, called *Lunar*, is capable of doing "expression templates" in Java to optimize numerical array objects.

1 Introduction

Good numeric performance for Java is mostly limited to code written in what might be called *JavaTran* (Java/FORTRAN) style: Fortran 77-like code surrounded by class declarations. A typical example is:

```
public class DoArrayStuff {
  public static apply(float[] w, float[] x,
    float[] y, float[] z)
  {
    for (int i=0; i < w.length; ++i)
      w[i] = x[i] + y[i] * z[i];
  }
}
```

This is not much of an improvement over Fortran 77. It would be nice if we could use the object-oriented features of Java in performance-critical code. Then we could write high-level code using objects representing arrays, matrices, tensors, and the like. Suppose we had a package which provided efficient numerical array objects; with operator overloading as proposed by the Java Grande committee [6], we could have Fortran-90 style array notation. Using such a package, the `DoArrayStuff` code above could be written as:

G. Butler and S. Jarzabek (Eds.): GCSE 2000, LNCS 2177, pp. 188–200, 2001.

```
public static apply(Array w, Array x, Array y,
  Array z)
{
  w = x + y * z;
}
```

Without operator overloading, one would write:

```
w.assign(x.plus(y.times(z)))
```

This is an improvement over *JavaTran* (Java in Fortran 77 style). However, can this Array class be implemented efficiently? This is an old question, with four well-known implementation choices:

1. **Compiler extension**: Write a compiler or preprocessor which recognizes a particular array class, and optimizes it using special semantics. This is the intent of the `valarray` class in the C++ standard, and of the ROSE preprocessor for C++ [1]. In Java, this approach is used by the Ninja compiler [11], which recognizes a special set of array classes and performs optimizations for them using "semantic inlining".
2. **Pairwise evaluation**: Each subexpression such as `y.times(z)` returns an intermediate Array result. The extra loops, memory allocation, and memory movement make this inefficient. Automatically fusing these loops is more difficult than the equivalent Fortran or C loop fusion problem, because the array pointers and loop bounds are fields of array objects. So far, no commercial compiler has been able to eliminate the temporary arrays.
3. **Deferred evaluation**: In this approach, expressions such as `y.times(z)` construct a parse tree of an expression as a data structure. When a parse tree is assigned to an array, a match can be sought in a library of commonly used array expressions. This requires considerable run-time overhead, and if the expression is not found, one must revert to temporary arrays.
4. **Expression templates** [16]: This C++ technique is similar to deferred evaluation, in that a parse tree of the expression is created. The parsing is done at compile time using C++ templates, by encoding the parse tree as a template type.[1] When a parse tree is assigned to an array, a function like this is called:

   ```
   template<class T>
   void assign(Array w, Expr<T> expr)
   {
     for (int i=0; i < n; ++i)
       w.data[i] = expr.eval(i);
   }
   ```

   For each array position `i`, `expr.eval(i)` traverses the parse tree, evaluating the expression. C++ semantics require that this traversal be done at compile

[1] For example, `x+y*z` might be parsed as the template type `Expr<Array,Plus,Expr<Array,Times,Array> >`

time, so the resulting loop is efficient. This technique is the basis of C++ libraries such as Blitz++ [17] and POOMA [8].

Of these approaches, expression template-based array libraries have been popular for C++, since they deliver efficiency without compiler extensions.

It turns out that templates in C++ are strikingly like *partial evaluation*. A partial evaluator takes a program, performs the operations which depend only on known values, and outputs a specialized program [7]. The standard example is shown in Figure 1.

```
float pow(float x, int n)
{
  if (n == 0)
    return 1.0;
  else
    return x * pow(x,n-1);
}

float a, b;
a = pow(b,3);
```

(a) Some code

```
// pow has been specialized for n=3
float pow_3(float x)
{
  return x * x * x;
}

float a, b;
a = pow_3(b);
```

(b) After partial evaluation

Fig. 1. Partial Evaluation Example.

C++ requires that all template parameters be known at compile time. When a template parameter is given by an expression (for example, `Vector<3+8>`), that expression *must* be evaluated at compile time. C++ templates effectively require that a partial evaluator be built into the compiler to evaluate template expressions [12,18]. It is this partial evaluation which makes possible expression templates and other template-based optimizations.

So here is a thought. Perhaps if we implemented a partial evaluator for Java, we could get the same performance benefits one gets from templates in C++, without templates.

This turns out to be at least partly true. We demonstrate a partial evaluator for Java that can be used to implement "expression templates" and similar performance-enhancing techniques from C++. We do this without any language extensions; we do not use GJ or other genericity proposals for Java.

1.1 Structure of this Paper

We start by considering an analogue of the C++ "expression templates" technique for Java (Section 2). This gives users array notation, at the cost of very poor performance. When this code is partially evaluated, we obtain performance similar to code written in *JavaTran* style (Section 3). In Section 4, we overview the Lunar compiler and summarize the optimizations performed. We point out some shortcomings of our compiler and some related work in Section 5.

2 A Java Version of "Expression Templates"

Our goal is to write a class `Array` which provides a 1-D array of floats. Users of this class will write code such as:

```
int n = 1000;
Array w = new Array(n);
Array x = new Array(n);
Array y = new Array(n);
Array z = new Array(n);

w = x + y * z;     // an array expression
```

Since we do not yet have overloaded operators in Java, we will write `w=x+y*z` like this:

```
w.assign(x.plus(y.times(z)));
```

Array expressions will be evaluated for every element in the array; the equivalent *JavaTran* version of `w=x+y*z` is:

```
float[] w, x, y, z;         // ...
for (int i=0; i < n; ++i)
  w[i] = x[i] + y[i] * z[i];
```

We start by creating a highly inefficient mechanism for evaluating array expressions which resembles the "expression templates" technique of C++. Then we will see that a suitable optimizing compiler can automatically turn this into the equivalent *JavaTran* implementation. More generally, such a optimizer has the potential to duplicate many of the generative programming features of C++ templates.

Fig. 2. Inheritance Diagram for the `Array` Classes.

For our pseudo-"expression templates" implementation, we will need objects representing array expressions. We will use the class hierarchy in Figure 2: `Expr` is a common base class of array expressions, and `Array` is itself an expression. To represent an expression such as `y*z` (or `y.times(z)`), we use an instance of a class `BinaryOpExpr` (short for Binary Operator Expression). `BinaryOpExpr` contains two expressions and a pointer to a binary operator object; to create an object representing `y*z`, we could write

```
Expr expr = new BinaryOpExpr(y, z, new Times());
```

where `Times` is an object representing multiplication. The definitions of these classes are shown in Figures 3 and 4.

The expression base class `Expr` defines an abstract method `float eval(int i)` which evaluates an array expression at a single array index `i`. Hence we can assign an expression to array using this method of class Array:

```
public class Array extends Expr {
 ...
 public void assign(Expr e)
 {
    for (int i=0; i < length; ++i)
      data[i] = e.eval(i);
 }
}
```

```
public class Array extends Expr {
  float data[];
  int length;

  public Array(int n)
  {
    data = new float[n];
    length = n;
  }

  public float eval(int i)
  {
    return data[i];
  }

  public void set(int i, float value)
  {
    data[i] = value;
  }

  public void assign(Expr e)
  {
    int t = length;
    for (int i=0; i < t; i=i+1)
      data[i] = e.eval(i);
  }
}
```

```
public abstract class Expr {
  public abstract float eval(int i);

  public Expr plus(Expr b)
  {
    BinaryOperator plus = new Plus();
    return new BinaryOpExpr(this,b,plus);
  }

  public Expr times(Expr b)
  {
    BinaryOperator times = new Times();
    return new BinaryOpExpr(this,b,times);
  }
}

public class BinaryOpExpr extends Expr {
  Expr a, b;
  BinaryOperator op;

  public BinaryOpExpr(Expr _a, Expr _b,
      BinaryOperator _op)
  {
    a = _a;
    b = _b;
    op = _op;
  }

  public float eval(int i)
  {
    return op.apply(a.eval(i),b.eval(i));
  }
}
```

Fig. 3. The Array Class, and Parse Tree Classes `Expr` and `BinaryOpExpr`.

Figure 5 shows a test program which exercises the Array class. It initializes some arrays, then evaluates the expression `w=x+y*z`. The method which assigns the expression to `w` is `Array.assign(Expr)`:

```
    public abstract class BinaryOperator {
        public abstract float apply(float a, float b);
    }

    public class Plus extends BinaryOperator {
        public float apply(float a, float b)
        {
            return a+b;
        }
    }

    public class Times extends BinaryOperator {
        public float apply(float a, float b)
        {
            return a*b;
        }
    }
```

Fig. 4. The `BinaryOperator` Base Class and Two Subclasses.

```
public class Test {
    public static void main(java.lang.String[] args)
    {
        // Create some arrays
        int n = 12345;
        Array w = new Array(n);
        Array x = new Array(n);
        Array y = new Array(n);
        Array z = new Array(n);

        // Initialize with data
        for (int i=0; i < n; i=i+1)
        {
            x.set(i,i*0.33f);
            y.set(i,10.0f+i);
            z.set(i,100.0f*i);
        }

        // With operator overloading, this would be
        // w = x + y * z
        w.assign(x.plus(y.times(z)));
    }
}
```

Fig. 5. Test Code for the `Array` Class.

```
public void assign(Expr e)
{
    for (int i=0; i < length; i=i+1)
        data[i] = e.eval(i);
}
```

This method loops through the array, evaluating the array expression for each `i`, and storing the result in `w.data[i]`. To evaluate each `e.eval(i)`, six virtual function calls, three bound checks, and numerous pointer indirections are required. Not surprisingly, performance is quite poor with typical Java compilers. This loop has been benchmarked at 0.7 Mflops using the Kaffe JIT compiler on a 300 MHz sparcv9; that is roughly 1 flop every 428 clock cycles.

With partial evaluation, all of this inefficiency can be removed. We have implemented a prototype compiler called *Lunar* that compiles Java to an intermediate form, partially evaluates it, and emits C code. This is then compiled to machine code using a C compiler. When applied to the example program (Figure 5), Lunar performs these optimizations:

- Constant and copy propagation through the heap, resolving (when possible) virtual method calls and pointer indirection at compile time.
- Inlining virtual methods.
- Elimination of bound checks when they are provably unnecessary.
- Elimination of unnecessary temporary objects.

```
int i = 0;
for (; (i < 12345);)
{
    int __a75 = (i * 4);
    int __a71 = (i * 4);
    float __a31 = *((float *) (array1d__96 + __a71));
    int __a184 = (i * 4);
    float __a171 = *((float *) (array1d__107 + __a184));
    int __a163 = (i * 4);
    float __a176 = *((float *) (array1d__118 + __a163));
    float __a35 = (__a171 * __a176);
    float __a76 = (__a31 + __a35);
    *((float *) (array1d + __a75)) = __a76;
    i = (i + 1);
}
```

Fig. 6. C Code Generated by Lunar for the Expression `w=x+y+z` of Figure 5

The C code generated by Lunar for the Java expression `w.assign(x.plus(-yexpr.times(zexpr)))` is shown in Figure 6. Lunar is able to eliminate all the virtual functions, remove all bound checks, and even get rid of the temporary `BinaryOpExpr`, `Plus`, and `Times` objects.

3 Benchmark Results

We present preliminary benchmark results on a sparcv9 processor at 300 MHz, using `gcc -O3` as the back end compiler for Lunar. Figure 7 compares the performance of the Lunar compiler to hand-coded C for the array expression $w = x + y * z$. The JavaTran series shows the performance of Lunar on the loop

```
float[] w, x, y, z;   // ...
for (int i=0; i < n; i=i+1)
  w[i] = x[i] + y[i]*z[i];
```

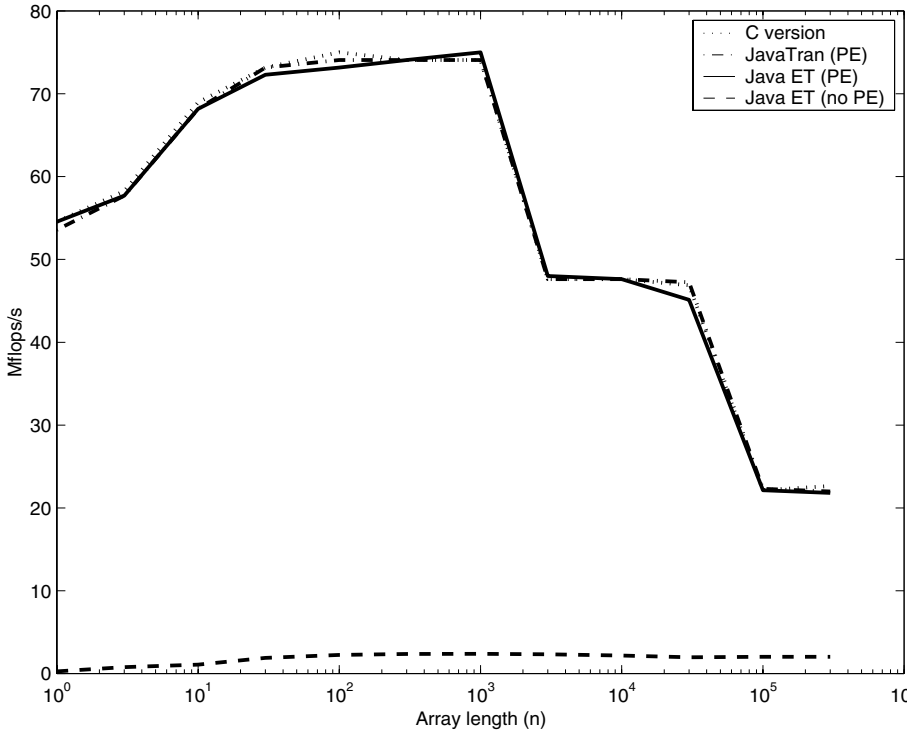


Fig. 7. Benchmark results for the array expression `w=x+y*z`, comparing (from top to bottom) hand-coded C, *JavaTran*-style code compiled with Lunar, Java "Expression-templates" version compiled with Lunar (with Partial Evaluation), Java "Expression-templates" compiled with Lunar (no Partial Evaluation). The three plateaus correspond to (left to right): L1 cache, L2 cache, and out-of-cache performance.

The Java ET series show the performance of `w.assign(x.plus(y.times(z)))` using `Array` objects. When the partial evaluator is disabled (the "no PE" series),

Table 1. Mflops/s for the array expression $w = x + y * z$, $n = 1000$, single precision. PE=Partial Evaluation.

JVM or compiler	"JavaTran"	"Expression Templates"
Lunar (no PE); gcc -O2 backend	33.1	2.4
Lunar (with PE); gcc -O2 backend	74.6	74.6
Sun Hotspot 1.3beta (JIT)	1.4	0.4
Transvirtual Kaffe (JIT)	4.3	0.7

performance takes a drastic hit – indicating that Lunar, not `gcc`, is doing the important optimizations.

Table 1 compares performance of Lunar to two JIT compilers (Sun Hotspot and Transvirtual Kaffe).

The partial evaluator gives a modest improvement over the *JavaTran*-style code, but the biggest improvement is for the Java version of "expression templates".

Table 2. Time to compile the "expression templates in Java" example of Figure 5, excluding C compilation time.

Phase	Elapsed time (ms)
Parsing	916
Conversion to Lunar IL_2	303
Prepasses	91
Partial evaluation	1569
Unparsing to C	678
Total	3557

Table 2 shows a summary of the time taken by Lunar to compile the "expression templates in Java" example with partial evaluation. The Lunar compiler is written in Java, and was run using Sun Hotspot 1.3beta. Once the compiler is able to compile itself, these times should decrease.

4 Compiler Overview

Figure 4 shows an overview of the Lunar compiler. The front end takes Java code and translates it into an intermediate language, called Lunar Intermediate Language 2 (IL_2). Prior to partial evaluation, a prepass puts the code into a form which makes partial evaluation simpler. This code is then passed to the partial evaluator. The output from the partial evaluator is then translated into C and compiled using a C compiler.

Table 3 summarizes the analyses and transformations currently implemented in Lunar. Many of these are similar to those implemented in conventional compilers. However, there are substantial differences between partial evaluation and

typical optimizers. Partial evaluators tend to have a flavour of symbolically executing a program, whereas typical imperative compilers are more flow-graph based.

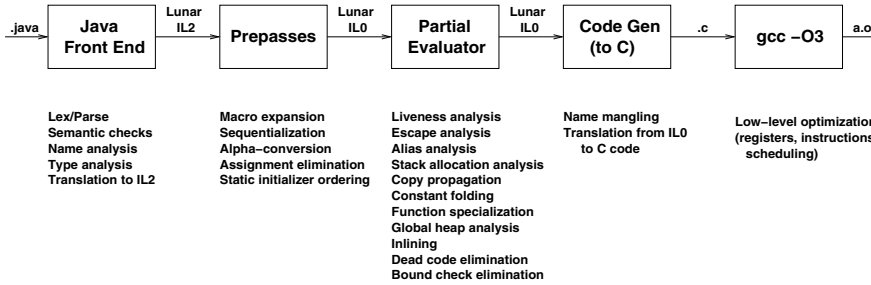


Fig. 8. Overview of the Lunar Compiler.

Table 3. Analyses and Transformations in the Lunar Compiler.

Prepass transforms	**Purpose**
Macro expansion	Expand macros used to simplify the Java front end
Sequentialization	Name intermediate values, simplify language structure
Assignment elimination	Remove variable assignments; ensure that variable names map uniquely to values within a scope.
α-conversion	Rename variables to ensure unique names
Static initializer ordering	Find initialization order for global variables; delete unused functions and global variables.
Partial evaluator	**Purpose**
Liveness analysis	Decide how function parameters are used.
Escape analysis	Decide if values might escape into the heap.
Alias analysis	Decide if heap objects are alias-free.
Stack allocation analysis	Allocate objects on the stack instead of the heap when possible.
Copy propagation	Find variables that refer to the same value.
Constant folding	Fold constants, eliminate primitive operations
Function specialization	Specialize functions according to known argument values
Global heap analysis	Constant and copy propagation through the heap.
Inlining	Selectively inline functions to improve optimization.
Dead Code Elimination	Eliminate dead variables, code, functions, and global variables.
Bounds check elimination	Remove bounds checks that are provably unnecessary.

5 Summary

5.1 Caveats

Lunar's Java front end handles a modest subset of Java 1.1. It does not yet support threads or inner classes. It is possible that use of some Java language features (for example, synchronization) will make it harder to perform the optimizations described in this paper.

Lunar emits code for a "loose numerics" version of Java. It uses the native floating point hardware, without concern for whether this correctly implements Java numeric semantics. This is not a requirement, but a shortcut. Strict numerics could be implemented, although they would likely have performance implications on some platforms.

Lunar does not yet perform null pointer checking. The plan is to follow the example of `gcj`, and trap SIGSEGV signals. This does not require changing any of the translation or code generation; trapping is done by the hardware and handled in a runtime library. Hence, the benchmark results reported here are not compromised by our omission of null pointer checking.

Our Java runtime does not do any garbage collection. In most of the benchmark results presented, the partial evaluator eliminates all memory allocation inside loops. Hence the absence of a garbage collector does not affect our performance in a major way. The exception is the benchmarks of Table 1 which show Lunar results without PE. For these results, not doing garbage collection gives Lunar an unfair advantage over the JIT compilers.

Some of the algorithms in Lunar are $O(n \lg n)$, and a few are quadratic in certain (possibly unlikely) scenarios. Lunar has not yet been tested on large Java programs, so it is possible we will uncover scaling problems. The plan is to provide a set of optimization switches `-O1,-O2,..,-O5` which progressively enable more expensive and expansive forms of partial evaluation.

The algorithms in Lunar do not require closed-program analysis. However, the more of the program Lunar can see, the better it can optimize.

Unlike C++ templates, Lunar offers no guarantee of compile-time evaluation. There is no analogy to type template parameters in C++, although Lunar can specialize functions based on types related by an inheritance hierarchy.

5.2 Related Work

There is a wealth of literature about compilers and partial evaluation, both separately and their intersection. The idea of resolving virtual functions through specialization was detailed by Dean et al [4]. Lunar achieves a similar effect "for free" by relying on the heap analyzer to propagate function names through dispatch tables. Khoo [9] used partial evaluation to compile inheritance, doing what would (analogously) in Java be dispatch table (or vtable) layout, but he did not use partial evaluation to resolve and inline virtual functions.

Many of the C++-template-like optimizations performed by Lunar are driven by a heap analyzer, which does constant and copy propagation through the heap.

This builds on a tradition of partly-static data structures in the partial evaluation community (e.g. [3]), and of store analyzers in the imperative world (e.g. [13,14]).

Volanschi et al [19] describe a partial evaluator for Java which relies on user annotations ("specialization classes") to guide specialization of Java programs. This provides finer control of specialization than Lunar, at the cost of requiring language extensions.

The benefits of partial evaluation for scientific programs are well known; see for example [2,5,10]. Lunar is not designed to optimize scientific programs per se, but rather to provide reliable partial evaluation semantics which can be used as a driving mechanism for performing domain-specific optimizations (for example, "expression templates" for array classes). In this regard it has similar goals as two- or multi-stage languages (e.g. [15]) which seek to provide a natural framework for writing "program generators".

Acknowledgments

Lunar uses JavaCC for lexing and parsing, and the GCC compiler for its back end. The term *JavaTran* derives from the similar term *C++Tran* coined by Scott Haney. I am grateful to Kent Dybvig, Ken Chiuk, Jeremy Siek, and Steve Karmesin for comments and discussion.

References

1. Bassetti, F., Davis, K., and Quinlan, D. Optimizing transformations of stencil operations for parallel object-oriented scientific frameworks on cache-based architectures. *Lecture Notes in Computer Science 1505* (1998), 107.
2. Berlin, A., and Weise, D. Compiling scientific code using partial evaluation. *Computer 23*, 12 (Dec 1990), 25–37.
3. Consel, C. Binding time analysis for higher order untyped functional languages. In *1990 ACM Conference on Lisp and Functional Programming* (June 1990), ACM, ACM Press, pp. 264–272.
4. Dean, J., Chambers, C., and Grove, D. Selective specialization for object-oriented languages. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)* (La Jolla, California, 18–21 June 1995), pp. 93–102.
5. Glück, R., Nakashige, R., and Zöchling, R. Binding-time analysis applied to mathematical algorithms. In *System Modelling and Optimization* (1995), J. Doležal and J. Fidler, Eds., Chapman & Hall, pp. 137–146.
6. Interim Java Grande report. Tech. Rep. JGF-TR-4, Java Grande Committee, 1999.
7. Jones, N.D. An introduction to partial evaluation. *ACM Computing Surveys 28*, 3 (Sept. 1996), 480–503.
8. Karmesin, S., Crotinger, J., Cummings, J., Haney, S., Humphrey, W., Reynders, J., Smith, S., and Williams, T. Array design and expression evaluation in POOMA II. In *ISCOPE'98* (1998), vol. 1505, Springer-Verlag. Lecture Notes in Computer Science.
9. Khoo, S.C., and Sundaresh, R. S. Compiling inheritance using partial evaluation. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (New Haven, CN, June 1991), vol. 26(9), pp. 211–222.

10. Kleinrubatscher, P., Kriegshaber, A., Zöchling, R., and Glück, R. Fortran program specialization. *SIGPLAN Notices 30*, 4 (1995), 61–70.
11. Moreira, J.E., Midkiff, S.P., Gupta, M., Artigas, P.V., Snir, M., and Lawrence, R.D. Java programming for high-performance numerical computing. *IBM Systems Journal 39*, 1 (2000), 21–56.
12. Salomon, D.J. Using partial evaluation in support of portability, reusability, and maintainability. In *Compiler Construction '96* (Linköping, Sweden, 24–26 Apr. 1996), pp. 208–222.
13. Sarkar, V., and Knobe, K. Enabling sparse constant propagation of array elements via array SSA form. *Lecture Notes in Computer Science 1503* (1998), 33.
14. Steensgaard, B. Sparse functional stores for imperative programs. In *ACM SIGPLAN Workshop on Intermediate Representations (IR'95)* (Jan. 1995), vol. 30 (3) of *SIGPLAN Notices*, ACM Press, pp. 62–70.
15. Taha, W., and Sheard, T. Multi-stage programming with explicit annotations. *ACM SIGPLAN Notices 32*, 12 (1997), 203–217.
16. Veldhuizen, T.L. Expression templates. *C++ Report 7*, 5 (June 1995), 26–31. Reprinted in C++ Gems, ed. Stanley Lippman.
17. Veldhuizen, T.L. Arrays in Blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)* (1998), Lecture Notes in Computer Science, Springer-Verlag.
18. Veldhuizen, T.L. C++ templates as partial evaluation. In *Proceedings of PEPM'99, The ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, ed. O. Danvy, San Antonio, January 1999.* (Jan. 1999), University of Aarhus, Dept. of Computer Science, pp. 13–18.
19. Volanschi, E.-N., Consel, C., Muller, G., and Cowan, C. Declarative specialization of object-oriented programs. In *ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'97)* (October 1997), pp. 286–300.

Author Index