Report Number: WUCS-2007-29

2007-05-01

Unwoven Aspect Analysis

Morgan G. Deters

Various languages and tools supporting advanced separation of concerns (such as aspect-oriented programming) provide a software developer with the ability to separate functional and non-functional programmatic intentions. Once these separate pieces of the software have been specified, the tools automatically handle interaction points between separate modules, relieving the developer of this chore and permitting more understandable, maintainable code. Many approaches have left traditional compiler analysis and optimization until after the composition has been performed; unfortunately, analyses performed after composition cannot make use of the logical separation present in the original program. Further, for modular systems that can be configured... Read complete abstract on page 2.

Unwoven Aspect Analysis

Morgan G. Deters

Complete Abstract:

Various languages and tools supporting advanced separation of concerns (such as aspect-oriented programming) provide a software developer with the ability to separate functional and non-functional programmatic intentions. Once these separate pieces of the software have been specified, the tools automatically handle interaction points between separate modules, relieving the developer of this chore and permitting more understandable, maintainable code. Many approaches have left traditional compiler analysis and optimization until after the composition has been performed; unfortunately, analyses performed after composition cannot make use of the logical separation present in the original program. Further, for modular systems that can be configured with different sets of features, testing under every possible combination of features may be necessary and time-consuming to avoid bugs in production software. To solve this testing problem, we investigate a feature-aware compiler analysis that runs during composition and discovers features strongly independent of each other. When the their independence can be judged, the number of feature combinations that must be separately tested can be reduced. We develop this approach and discuss our implementation. We look forward to future programming languages in two ways: we implement solutions to problems that are conceptually aspect-oriented but for which current aspect languages and tools fail. We study these cases and consider what language designs might provide even more information to a compiler. We describe some features that such a future language might have, based on our observations of current language deficiencies and our experience with compilers for these languages.

2007-29

Unwoven Aspect Analysis, Doctoral Dissertation, May 2007

Authors: Morgan G. Deters

Corresponding Author: mdeters@cse.wustl.edu

Abstract: Various languages and tools supporting advanced separation of concerns (such as aspect-oriented programming) provide a software developer with the ability to separate functional and non-functional programmatic intentions. Once these separate pieces of the software have been specified, the tools automatically handle interaction points between separate modules, relieving the developer of this chore and permitting more understandable, maintainable code.

Many approaches have left traditional compiler analysis and optimization until after the composition has been performed; unfortunately, analyses performed after composition cannot make use of the logical separation present in the original program. Further, for modular systems that can be configured with different sets of features, testing under every possible combination of features may be necessary and time-consuming to avoid bugs in production software.

To solve this testing problem, we investigate a feature-aware compiler analysis that runs during composition and discovers features strongly independent of each other. When the their independence can be judged, the

Type of Report: PhD Dissertation

WASHINGTON UNIVERSITY

Sever Institute
School of Engineering and Applied Science

Department of Computer Science and Engineering

Dissertation Examination Committee:
Ron K. Cytron, Chair
Chris Gill
Ronald Indeck
Chenyang Lu
Aaron Stump

UNWOVEN ASPECT ANALYSIS

by

Morgan G. Deters

A dissertation presented to the
Graduate School of Arts and Sciences
of Washington University in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

May 2007

Saint Louis, Missouri

Acknowledgments

I would first like to thank my advisor Dr. Ron Cytron for his guidance and continued patience on this research, this dissertation, and for my graduate education more broadly; Drs. Aaron Stump and Chris Gill for collaborations and insightful discussions over the years; and all five members of my doctoral committee, Drs. Ron Cytron, Chris Gill, Aaron Stump, Chenyang Lu, and Ron Indeck, from whom I was very fortunate to receive invaluable suggestions in performing this work. Without them, among other things, I would not graduate. So the dean tells me.

Charles Comstock assisted in the editing of this dissertation; for his reading of drafts and his interest I am grateful.

The reference counting work of Chapter 5 was joint work with Nick Leidenfrost, Matt Hampton, James Brodman, and Ron Cytron, originally published at Real-Time and Embedded Technology and Applications Symposium (RTAS) in 2004, and the rate-monotonic template metaprogramming of Chapter 4 was joint work with Chris Gill and Ron Cytron, originally published at the Model-Driven Embedded Systems (MDES) workshop at RTAS in 2003. I further wish to thank Dante Cannarozzi and Scott Friedman for their help in the conceptualization and implementation of the work in this chapter, and Martin Linenweber for his tireless effort implementing and improving the Clazzer tool, a dataflow-aware bytecode-manipulation and optimization framework I've used for the work of Chapter 5 and countless other projects over the years. My friend and colleague Angelo Corsaro was the original designer and implementer of jRate, a real-time Java compiler and runtime platform; in recent years I've collaborated with him on maintaining and extending jRate as a SourceForge.net project. It was this compiler and runtime platform I modified for Chapter 6, and the work would not have been possible but for Angelo's vision and determination in his work on jRate.

Off-campus, I kept my sanity (but not always my sobriety) only through the Herculean and coordinated efforts of Shanna Carpenter, Daron Dierkes, Justin Levine, Scott McGinnis, Sam Moyerman, Anna Olsson, John Reskusich, Angela Woike, and Bob Zimmermann especially; I could not have made it without these supportive and wonderful friends. Many friendships were critical and important to me, including those of Don Allgeier, Dawn Childress, Devin Childress, Lisa Clancy, Nicci Cobb, Charles Comstock, Matthew Cunningham,

Ravi Das, Ben Floyd-Clapman, Ava Hegedus, Jake LaBombarbe, Brent Larson, Annie McCutchen, Derrick Mosley, Ryan Ritter, William Smith, Ben West, Eddy Westbrook, and innumerable others.

I should thank all past and present members, affiliates, and conspirators of the Distributed Object Computing Group at Washington University for their friendship and stimulating discussions and/or arguments. Aside from those already mentioned, several in particular deserve special, alphabetically-ordered mention: Kitty Balasubramanian, Sharath Cholleti, Delvin Defoe, Steve Donahue, Lucas Fox, Scott Friedman, Matt Hampton, Mike Henrichs, Chris Hill, Richard Hough, Frank Hunleth, Victor Lai, Rob LeGrand, David Levine, Tobias Mann, Balachandran Natarajan, Kirthika Parameswaran, Michael Plezbert, Ravi Pratap Maddimsetty, Irfan Pyarali, Doug Schmidt, Venkita Subramonian, Justin Thiel, Stephen Torri, and Nanbor Wang. I must also acknowledge Peggy Fuller, Jean Grothe, Myrna Harbison, Sharon Matlock, Madeline Straatmann, and Stella Sung, without whom the department would not function, and the staff of Computing Technology and Services, without whom the department would not compute.

I never expected to be in touch with so many friends from my undergraduate days; these friendships have stood the test of time. For daily affirmations, laughs, rants, and more, I thank Jeremy Axe, Brandt Fundak, Mike Groeniger, Brian Ingold, Dan Isaacs, Lou Lomasky, Colleen Myers, Keith Myers, Tom Printy, Justin Schlenker, Andy Staats, Kasia Trapzso, and Jeremy Zawodny. Collectively, the *Goon Squad*.

I thank the faculty of Bowling Green State University's Department of Computer Science for their guidance; in particular, Drs. Ron Lancaster, Walter Maner, Sub Ramakrishnan, and Guy Zimmerman provided me with much to think about during a critical point in my academic life.

I thank Linda Stacy for teaching me years ago to cook under pressure and demanding that I be a decent human being. It meant a lot to me.

And I would especially like to thank my parents, Dr. Donald and Lynn Deters, for their lifetime of support and encouragement.

The research represented in this thesis was sponsored by the Defense Advanced Research Projects Agency (DARPA) under the PCES program (contract F33615–00–C–1697); DARPA and Raytheon Integrated Defense Systems under ARMS II; and the Air Force Research Laboratory (AFRL) and the Boeing Company under the ARTEC program.

`E323 23C4 BE50 29C2 9FBD  77F8 4E87 8464 2325 9CBD`

Morgan G. Deters

Washington University in Saint Louis

May 2007

For Ace and Zoë,
whose midnight meows
are even better than caffeine

Contents

List of Figures

ABSTRACT OF THE DISSERTATION

Unwoven Aspect Analysis

by

Morgan G. Deters

Doctor of Philosophy in Computer Science

Washington University in St. Louis, 2007

Ron K. Cytron, Chairperson

Various languages and tools supporting *advanced separation of concerns* (such as aspect-oriented programming) provide a software developer with the ability to separate functional and non-functional programmatic intentions. Once these separate pieces of the software have been specified, the tools automatically handle interaction points between separate modules, relieving the developer of this chore and permitting more understandable, maintainable code.

Many approaches have left traditional compiler analysis and optimization until after the composition has been performed; unfortunately, analyses performed after composition cannot make use of the logical separation present in the original program. Further, for modular systems that can be configured with different sets of features, testing under every possible combination of features may be necessary and time-consuming to avoid bugs in production software.

To solve this testing problem, we investigate a feature-aware compiler analysis that runs during composition and discovers features strongly independent of each other. When the their independence can be judged, the number of feature combinations that must be separately tested can be reduced. We develop this approach and discuss our implementation.

We look forward to future programming languages in two ways: we implement solutions to problems that are conceptually aspect-oriented but for which current aspect languages and tools fail. We study these cases and consider what language designs might provide even more information to a compiler. We describe some features that such a future language might have, based on our observations of current language deficiencies and our experience with compilers for these languages.

Preface

When faced with increasingly complex software requirements, software developers are motivated to find more advanced development tools and languages that allow them to reduce their labor. Ideally, such tools allow them more clearly to express their algorithmic and compositional intents and to reduce the repetitive and often redundant nature of many software development tasks. Proposed solutions to fit this need constitute a considerable slice of computer science research spanning several decades; the complete body of work is too considerable even to list. Among the notable recent work that is most relevant to this text are the development of cross-platform middleware, domain-specific languages, modelling tools, automatic programming, design patterns, advanced type systems, and new approaches to programming coupled with new languages to support these new approaches.

Such new devices drive upwards the level of abstraction afforded the programmer. When properly designed, new abstractions—whether new features of a language, functionality provided by middleware, or something else—are highly useful to developers; this is one large motivation behind the invention of new abstractions. Often this increased level of abstraction comes at a price: automated language translation and analysis can be more expensive or difficult, requiring additional compilation, execution, and testing time. This stretches the development cycle, countering, in part, some of the programming efficiency introduced by the abstraction.

This is not always the case, however. Sometimes, a new abstraction comes "for free" or even with additional unintended benefits. The use of higher-level abstractions can be highly useful to compilers when the *intent* of the developer becomes clearer as a result of their proper application. This dissertation argues that a certain modern abstraction, the *aspect*, has this property.

Aspect-Oriented Programming (AOP) boasts more modularity than typically capable of popular, non-AOP software systems. Through *aspects*, a programmer can modularly

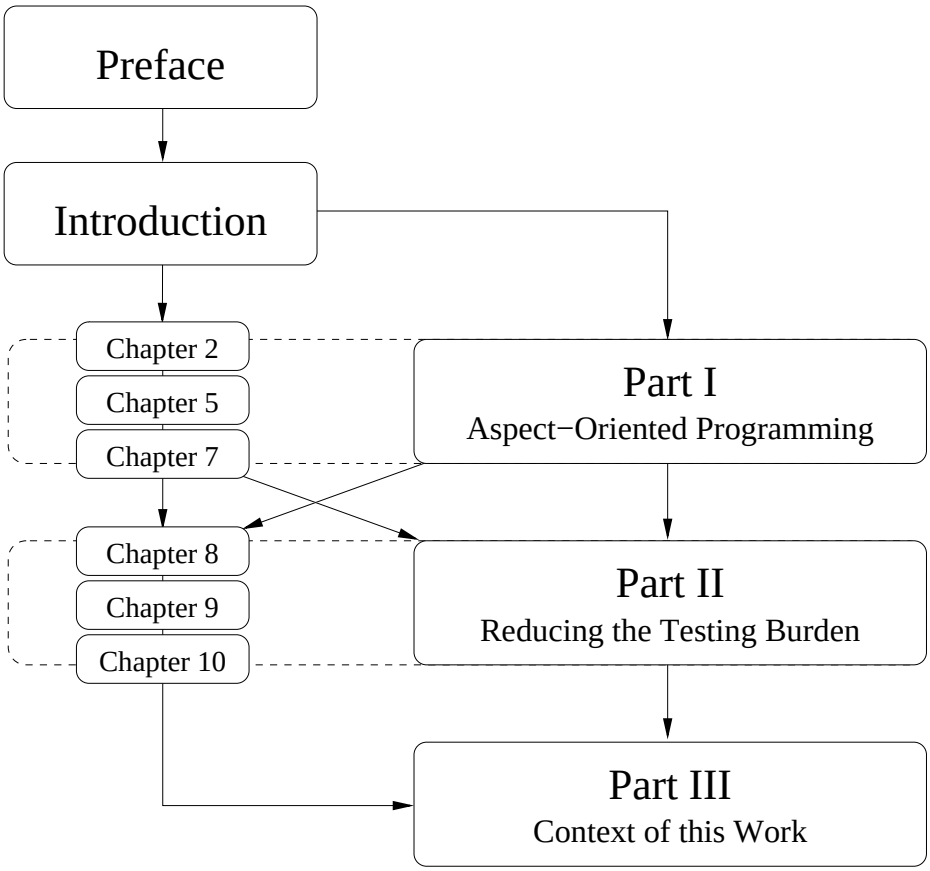
express not only algorithms (procedures) and constituents (objects), but emergent behavior (aspects) of his program. Reduced to a proper form, this has the potential to provide a wealth of information to a translation engine seeking to implement efficiently the specified software system. Further, information about how the system is *supposed* to act is partly specified in a manner not available in popular non-AOP languages.

This is the information exploited for the analysis herein developed. This dissertation is concerned primarily with investigating aspects and determining their effect on compiled code: it develops analysis techniques that can be applied to a collection of aspects to inform compilation, and it identifies deficiencies in current aspect technology.

⋆ ⋆ ⋆

This dissertation has been organized into three parts. After Chapter 1's introduction, Part I sets the stage by introducing aspect-oriented programming (Chapter 2) and providing some illustrative "systemic" examples of aspects (Chapter 3 describes system aspects and Chapters 4, 5, and 6 walk through nontrivial examples); it concludes with observations about these aspects and about aspect-oriented languages in general (Chapter 7). Chapters 4, 5, and 6 also stand on their own as novel solutions to the problems discussed in each. Part II lays out some problems of testing modern, feature-oriented software (Chapter 8), describes our notion of aspect dependence and our analysis (Chapter 9), provides case studies (Chapter 10), and outlines our implementation (Chapter 11). Part III surveys related work not noted elsewhere (Chapter 12) and concludes (Chapter 13). An index with topical and bibliographic entries is provided.

Following the introductory chapter, this dissertation can be read in alternate ways. Parts I and II can be skipped in part, especially for the reader familiar with aspect-oriented programming or dependence analysis; however, Chapters 2, 5, 7, 8, 9, and 10 are recommended reading in any case. Part III notes conclusions relevant to the work as a whole. With a nod to Dr. Knuth, the diagram on the next page is provided and makes explicit the set of recommended paths through this text.



Alternate paths through this text.

Chapter 1

Introduction

This dissertation intends to achieve two goals. The first is to argue for a specific direction for future programming languages. Part I of this work primarily concerns itself with the review of certain "system" aspects that can *conceptually* be written in an aspect-oriented programming style, though current languages and tools for that purpose are largely inadequate. This inadequacy is, at times, due to an inability to express the solution in a "natural" and type-safe way; at others, it is due to a problem with the performance of state-of-the-art tools and the executables they produce. In some instances, well-studied language features could be added to the host language to fix the problem; generic programming features would be very useful for some of these system aspect implementations, for example. Other cases require more work to add sufficient flexibility to the host language. Chapter 7 draws these conclusions from observations made of the material in Part I.

The second goal of this work is to demonstrate that aspects are not only highly useful to developers but to compilers and analysis tools as well. Programs written in aspect languages provide certain hints to the compiler that are often lost in non-aspect languages. Part II describes compiler mechanisms for discovering these hints and applies aspect independence to reduce the amount of testing required.

1.1 Preview of technical contributions

This work makes the following eleven contributions.

1. It argues for a shift in view of aspects and the types of applications for which they are useful.
2. It provides three examples of "system aspects" (Chapters 4, 5, and 6), each of which stand on their own as novel work:
   - a compile-time metaprogram is designed and implemented to statically configure a rate-monotonic schedule (Chapter 4);
   - an aspectual system for reusing heap storage in Java programs based on a reference-counting scheme is developed (Chapter 5); and
   - software support for a hardware garbage collector is motivated and its implementation described (Chapter 6).
3. It observes where these system aspects don't have an aspectual formulation with today's languages and tools (Chapter 7).
4. It indicates possible future directions for aspect languages to bring these aspects, and others like them, under the aspect purview (Chapters 7 and 13).
5. It introduces a taxonomy of dependence and independence relations for aspects; these terms are used inconsistently in the literature (Chapter 9).
6. It proposes an algorithm for determining valid configurations based on a static *feature set specification* (Chapter 8).
7. It investigates a method of analysis to determine strong independence of aspects (Chapter 10).
8. It provides an implementation of that analysis in a production compiler (Chapter 11).
9. It determines what benefits can be gained by aspect independence in terms of programmer effort (Chapter 10).
10. It considers possible future directions for aspect languages that could maximize this gain (Chapter 13).

11. It compares the future aspect language directions hinted at by contributions 4 and 10 above (Chapter 13).

For a concluding discussion of all of these contributions, see Chapter 13.

To our knowledge these contributions are not previously explored in the literature, except where noted; in particular, Chapters 4 and 5 are based on previous work published by the author and his colleagues, and Chapter 12 points to some related work not investigated elsewhere.

1.2 General terms and conventions used in this text

The terms *developer, software developer, programmer, application programmer, coder,* and sometimes *author* are used throughout and are synonymous. Except where explicitly stated or clear from context, no semantic distinction between these terms is intended. This being a text on compilers and languages, the term *user* is occasionally employed for the same purpose where its meaning should be clear.

Code listings and references to specific code attributes in the text are typeset in a `monospace typeface`. Conceptual programming entities, when not intended to refer to a specific code artifact in a specific implementation, are represented in sans serif. Traditionally uppercase acronyms of more than three letters are sometimes set in SMALL CAPITALS.

Part I

Aspect-Oriented Programs

New ideas go through stages of acceptance, both from within and without. From within, the sequence moves from "barely seeing" a pattern several times, then noting it but not perceiving its "cosmic" significance, then using it operationally in several areas, then comes a "grand rotation" in which the pattern becomes the center of a new way of thinking, and finally, it turns into the same kind of inflexible religion that it originally broke away from. From without, as Schopenhauer noted, the new idea is first denounced as the work of the insane, in a few years it is considered obvious and mundane, and finally the original denouncers will claim to have invented it.

– *Alan Kay, The Early History of Smalltalk [63]*

Chapter 2

Aspects and Aspect Weaving

Aspect-Oriented Programming (AOP) [67]—and, more generally, Advanced Separation of Concerns (ASoC) technology—is a broad and active area of research. Here we only introduce the subject. For reviews of several approaches to ASoC, see [44] and also Chapter 3 of [36].

2.1 Terms and definitions

We adopt the terminology of the programming language AspectJ [6]. AspectJ is a general-purpose aspect-oriented extension to Java [5].

2.1.1 Aspect

Conceptually, an *aspect* is the implementation of a particular view of program state and behavior related to a particular *concern* of the program. In aspect-oriented languages, aspects are technically an extension to the *class* concept. The body of an aspect may, like classes, contain method, field, constant, and type definitions. It may also contain *advice* and *pointcut* definitions, and *intertype declarations* (historically called field and method *introductions*). These terms are defined below.

From an implementation perspective, the main behavioral difference between a class and an aspect is that an aspect is not instantiated explicitly by the user. Instances of aspects are implicitly created by the runtime system in accordance with programmer specification.

Aspects may be *singletons* [49] or may be instantiated in certain *join point* contexts (below) or type contexts. Aspects typically also include advice and intertype declarations, described below, that have significant effects on program behavior.

2.1.2 Join points

Join points are points in the execution of a program. That definition is very general, and typically the join point model of a particular aspect-oriented language makes severe restrictions on what constitutes a valid *join point*.[1] AspectJ defines several kinds of join points:

- executions of methods and constructors
- calls to methods and constructors
- access to the values of fields
- assignments to the values of fields
- "pre"-initializations of object instances (before superclass constructor invocation)
- initializations of object instances (after superclass constructor invocation)
- the one-time initializations of classes (static initializations)
- the execution of exception handlers
- the execution of *advice* (see below)

Sets of join points can be flexibly constructed using *pointcuts*, and the composition of these sets is allowed using a natural syntax. For instance, the specification

```
execution(void Foo.run()) || execution(void Foo.main(String[]))
```

selects executions of the `Foo.run()` method and also executions of the `Foo.main()` method (for some class `Foo`). Pointcuts are further discussed below.

[1] Still, the general definition of *join point* is intentionally limiting; it is based on current widely-used aspect-oriented languages. Join point models need not be restricted to points in the *execution* of a program, but could represent points in the flow of data through the execution of a program, the connectedness of a program's data storage, or communication between two or more programs. These ideas are explored further in Chapter 13.

2.1.3 Pointcuts

Pointcuts, simply, are sets of join points.

Some *pointcut primitives* are defined by the AspectJ language to select join points matching certain criteria. First, each of the join point types of Section 2.1.2 has a corresponding primitive:

- `execution(`*pattern*`)` – executions of methods or constructors matching *pattern*
- `call(`*pattern*`)` – calls to methods or constructors matching *pattern*
- `get(`*pattern*`)` – accesses of fields matching *pattern*
- `set(`*pattern*`)` – assignments to fields matching *pattern*
- `preinitialization(`*pattern*`)` – pre-initializations of instances of classes matching *pattern*
- `initialization(`*pattern*`)` – initializations of instances of classes matching *pattern*
- `staticinitialization(`*pattern*`)` – the one-time initializations of classes matching *pattern*
- `handler(`*pattern*`)` – the execution of exception handlers for exception types matching *pattern*
- `adviceexecution()` – the execution of *advice* (see below)

Further, it is possible to select join points which are lexically *within* a type, constructor, or method:

- `within(`*pattern*`)` – join points lexically within a type matching *pattern*
- `withincode(`*pattern*`)` – join points lexically within a constructor or method matching *pattern*

The various application sites of these pointcut specifiers and of the join points described in Section 2.1.2 can be discovered statically for a program at compile time. However, pointcuts can select on dynamic data as well:

- `cflow(`*pointcut*`)` – selects join points in the control flow of join points in *pointcut*; for example, if *pointcut* includes a method call join point, `cflow(`*pointcut*`)` includes all join points that occur (dynamically) during the call, including the call join point itself
- `cflowbelow(`*pointcut*`)` – selects join points in the control flow of join points in *pointcut*, excluding the top-level join points themselves
- `if(`*expr*`)` – selects join points at which the Java boolean expression *expr* is true

Obviously the application sites of these dynamic pointcut primitives cannot be statically determined in general. In practice, candidate application sites can be statically determined, and runtime checks must be inserted into the program to determine whether the `cflow()` or `if()` applies in a particular dynamic context.

Additional pointcut specifiers select based on the types of data associated with the join point:

- `this(`*pattern*`)` – selects join points where the special `this` reference is of a type matching *pattern*
- `target(`*pattern*`)` – selects join points where the *target* of the join point—the object containing the field referenced or receiving a method call—is of a type matching *pattern*
- `args(`*pattern-list*`)` – selects join points where the associated arguments match all patterns in *pattern-list*

Finally, pointcuts can be negated (with `!`), and their union with other pointcuts can be computed (with `||`) as well as their intersection (with `&&`). They can be explicitly named and these names used in other pointcuts. Such named pointcuts can have formal parameters, and these formal parameters can be bound to the data associated with the join point using an extended version of the `this`, `target`, and `args` constructs:

- `this(`*identifier*`)` – selects join points where the special `this` reference has the type of formal parameter *identifier*; *identifier* is bound to this reference
- `target(`*identifier*`)` – selects join points where the *target* of the join point—the object containing the field referenced or receiving a method call—matches the type of formal parameter *identifier*; *identifier* is bound to this reference

- `args(`*pattern-or-identifier-list*`)` – selects join points where the associated arguments match the types of formal parameters and type patterns in *pattern-or-identifier-list*; named formal parameters are bound to the actual parameters at the join point

For a full account of pointcuts and matching and binding semantics in AspectJ, see the AspectJ programming guide [8] and the AspectJ 5 developer's notebook [7].

2.1.4 Advice

Procedures are passive: they are *invoked* to perform a *service* for the caller. A piece of *advice* is an inverted procedure: it *pushes* its behavior out to a particular set of join points that is specified by a named or anonymous pointcut.

Advice behavior is specified as arbitrary Java code that can be run *before*, *after*, or *around* (instead of) the join points in the pointcut. Advice running *after* join points can be restricted to those that return successfully, or restricted to those that return exceptionally (by throwing an exception), or can run unconditionally. Advice running *around* join points can choose to invoke the underlying join point or not. If *around* advice does invoke the underlying join point, it can do so with different parameters than the advice was itself provided.

This is a very general feature that has far-reaching implications for program design, as seen in the following chapters.

2.1.5 Intertype declarations

Intertype declarations are declarations inside of aspects that apply to another type.[2] They may introduce fields, constants, methods, and constructors to other types. These introduced members can have any access mode: if declared *public* they change the public interface of the type; if declared *private*, they are private to the aspect that introduced them (rather than to the type into which they are introduced, which remains oblivious to their existence).

[2]Intertype declarations used to be referred to as *introductions* and sometimes are still. The term *intertype declarations* now appears to be preferred by the AspectJ team and some members of the research community.

This facility is useful for many purposes. Public interfaces can be changed when appropriate, and *private* intertype declarations can be used to associate additional state with objects that would have to be kept in a separate dictionary data structure otherwise. This allows aspects to maintain good performance and footprint in some cases where it is difficult otherwise to do so without breaking modularity by implementing behavior in a class that is conceptually external to it.

2.1.6 Declare statements

AspectJ has a facility to change a program structurally in other ways as well. The *declare* facility can force a class to implement new interfaces or extend a different type, force a checked Java exception to be unchecked, and other things. This facility is beyond the scope of this chapter. The reader is referred to [8] and [7] for more details.

2.1.7 Summary

To summarize, *join points* are well-defined locations in a program corresponding to language features, such as method calls and exception handling. *Aspects* are bundles of state and behavior that push their behavior into matching sites of the program. They typically contain specified sets of join points called *pointcuts*, pieces of *advice* that operate on program code matching these pointcuts, and *intertype declarations* that structurally modify the program (by adding fields or methods, for example).

The process of *weaving* implements the advice and intertype declarations of an aspect-oriented program. The weaving process is described in the next section.

2.2 Aspect weaving

A compiler for an aspect-oriented programming language like AspectJ contains an additional step not present in other compilers (see Figure 2.1). It parses the program in a similar way, keeping track of advice, intertype declarations, and join points matching pointcuts of interest. Then a *weaving* pass executes, implementing intertype declarations and pushing advice behavior out to join points.

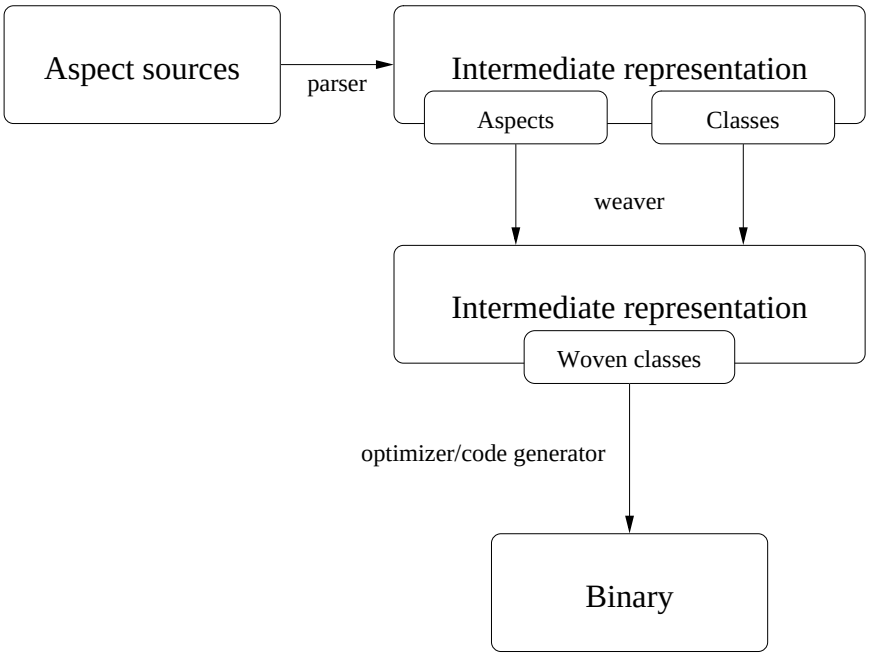


Figure 2.1: Compiler infrastructure for an aspect-oriented language. Here, the entire body of program code is fed into the parser, including all aspects and classes to be used in the program.

Because aspect weavers push advice out to all parts of a program, they need to operate on the *entire* program. This is unlike compilers for Java, C, or C++, which can compile source files separately. Modular compilation is difficult for aspect-oriented tools, but recent work has suggested an approach [43]. Modular compilation is supported in recent AspectJ releases [8]; it is not supported by the AspectBench compiler [11, 90], an alternative compiler for the AspectJ language. Further discussion of modular compilation mechanisms is beyond the scope of this work.

There are many possible approaches to aspect weaving. We present two that differ in how aspects affect other aspects in the system. Advice application (inside a weaver) can be seen as code transformation; indeed, this is often done in theoretical work on aspects [111]. We specify these two weaving mechanisms using a code transformation model.

It is not the purpose of this chapter to give a full theoretical treatment of aspects and aspect weaving, some of which exists in the aspect literature [111]. Rather, the formal treatment here is intended as a descriptive aid, explaining some important differences between aspect weaver designs and set the stage for discussions of later chapters.

2.2.1 A nominal weaver

Given n pieces of advice $A_1 \dots A_n$ and a base program P, each A_i is translated into a code transformation φ_i that causes the advice to be implemented at applicable join points. The composed program P' is given by:

$$P' = \varphi_n(\varphi_{n-1}(\cdots \varphi_2(\varphi_1(P))))$$

It is instructive to map these elements back to Figure 2.1. P is the set of classes and aspects in intermediate representation in the figure. $A_1 \dots A_n$ are the intermediate representations of the advice in the aspects, and P' is the woven intermediate representation.

Clearly, in this weaver, a piece of advice A_{i+1} can affect advice A_i, since the code is transformed with φ_{i+1} before being transformed with φ_i. However, advice A_i *cannot* affect advice A_{i+1} since it transforms the program first. This behavior is an important quality of this weaver, and is the primary difference between the nominal weaver and AspectJ's weaver, described next.

2.2.2 AspectJ's weaver

Given n pieces of advice $A_1 \dots A_n$, their owning aspects $C_1 \dots C_m$ and a base program P, each A_i is translated into a method M_i of its owning aspect class and a code transformation φ_i that causes M_i to be called at applicable join points. The composed program P' is given by:

$$P' = \varphi_n(\varphi_{n-1}(\cdots \varphi_2(\varphi_1(\textit{insert-methods-}A_1(\cdots \textit{insert-methods-}A_n(P))))))$$

This weaver design works by first inserting the advice-implementing methods M_i into the aspect classes *before* matching join points and implementing any advice. This has the consequence that advice can have self-referential and, together with other advice, circular effects.

However, the order of application of code transformation is still significant; where two pieces of advice A_i and A_j apply to the same join point, φ_i and φ_j will both rewrite part of the program at that join point. If $i < j$ and A_i and A_j are both *before* advice, then the effect of A_j will be *closer* to the join point after both transformations are applied;

Figure 2.2 shows this for a call join point. *After* and *around* advice implementation order is significant for similar reasons.

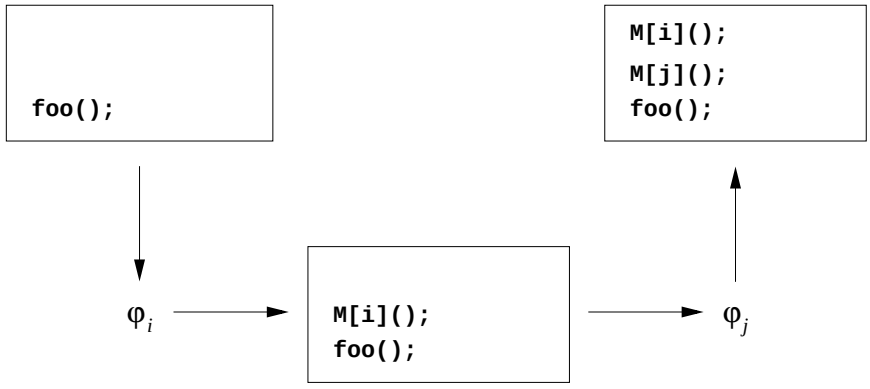


Figure 2.2: Two pieces of *before* advice implemented at a call join point, an illustration of AspectJ's weaving mechanism in Section 2.2.2. Note that order is significant here. Since $i < j$, $\varphi(i)$ transforms the code snippet before $\varphi(j)$ does; the advice implemented by $\varphi(j)$ is *closer* to the join point in the woven program.

2.3 Chapter summary

We have given a short technical introduction to aspect-oriented programming and provided definitions for the terms *aspect*, *advice*, *join point*, *pointcut* and *intertype declaration*. We have described the process of *aspect weaving* and demonstrated two different designs for weavers with differing behavior.

Chapter 3

System Aspects

Aspect-Oriented Programming is a powerful mechanism that allows decoupling different concerns in programs. It is especially useful for removing pervasive, systemic concerns from the main program logic entirely and stating them separately in declarative form. Conceptually, many tasks can be expressed in terms of aspects; however, popular aspect languages and tools are often not sufficient to perform these tasks.

This is especially true of aspects that attempt to modify the behavior of the base language and runtime library—a highly pervasive and systemic concern—as opposed to modifying application logic behavior. Modifying language behavior is a natural task to perform with aspects—whether for performance reasons, to increase the usability of the language, or for another reason. Throughout this text these are termed *system aspects* because of their *systemic* and *aspectual* nature: they are properties of the *system* underlying the software. They can be described (at least conceptually) as *aspects*, but they are primarily intended to extend the functionality of the language and runtime system instead of that of the application.

These system aspects would be most easily realized in a system where the language and runtime system is implemented in the same language that it implements. One example of such a system is Jikes RVM [58], which is a Java virtual machine itself implemented in Java. If Jikes RVM were extended with aspect-oriented features and re-engineered to allow a high level of dynamicity, aspects conceivably could directly apply to the just-in-time (JIT) compiler and all runtime libraries.

This chapter provides some background for Chapters 4, 5, and 6, each of which describes a systemic problem and an aspect-like solution to solve it. Importantly, these solutions are not implemented as aspects directly: for differing reasons, they cannot be without serious limitations, and this is made clear in each discussion. These chapters represent novel research contributions on their own; they are self-contained within this dissertation. Chapter 7 concludes this part by tying together these system aspects, and it draws conclusions about aspect language design. Later chapters probe this topic more deeply.

The remainder of this chapter previews existing language mechanisms that can be used *in lieu* of aspects to perform tasks that cannot be adequately handled by current aspect languages and tools. These mechanisms are employed in the following chapters.

3.1 Aspect-similar mechanisms

Chapters 4, 5, and 6 describe three projects where the aims are compatible with the aims of aspect-oriented programming systems. However, current aspect languages and tools are not sufficient for achieving the desired outcome. Each chapter provides background and motivation, but this section provides a brief introduction to the chief mechanisms used *instead* of aspects in these chapters.

3.1.1 Template metaprogramming

The term *template metaprogramming* [108] is typically limited to C++, though other programming languages have analogous and similar features, among them compile-time metaobject protocols [31, 103] and some advanced macro systems [23]. Template metaprogramming is described more fully in Chapter 4. Briefly, C++'s generic types mechanism is a Turing-complete sublanguage of C++ evaluated at compile time, a fact realized after the language was designed [108]. Almost as an accident, C++ supports dependent types and (with `typedef`) arbitrary type mappings.[1] This permits a natural approach to compile-time flexibility directly within the C++ language itself: a few constants can be declared manually, and other constants can be configured by the compiler based on them. Further, the

[1] It must be mentioned, though, that as they weren't initial design goals, for certain purposes these features aren't as well-suited as their analogues in other languages.

types, data structures, and algorithms used in the compiled program can be automatically selected from a set of options without *any* runtime overhead. Alexandrescu [1] provides a highly useful reference to practical applications of the technique.

Chapter 4 applies C++ template metaprogramming to implement an entirely compile-time configuration of runtime artifacts. When compiled with an optimizing compiler there is zero runtime cost associated with the approach of Chapter 4, but also zero runtime flexibility: the design flexibility is entirely at the source level.

3.1.2 Bytecode transformations

Bytecode transformations are simply post-compilation transformations of Java programs. Once compiled into architecture-independent *bytecode,* Java programs are transformed further to effect desired goals. Java analysis, optimization, and obfuscation engines typically target bytecode rather than Java source. Because bytecode transformations operate only on the structured bytecode format, they are portable across Java source compilers and also across virtual machines and bytecode JIT compilers.

Chapter 5 attempts an aspect solution to a problem and compares its performance to a targetted transformation of Java bytecode (which includes some aspects). The transformations are certainly within the reach of an aspect language, and features of such a language are discussed. This can be seen almost as a *partial evaluation* approach, where part of the code can be partially evaluated at compile time, resulting in a hybrid static/dynamic solution: where possible, flexibility at the source level is optimized away. There is a runtime performance hit in places where such optimization cannot be performed.

3.1.3 Compiler modifications

Given source availability, compiler modifications are also possible. These are generally nonportable,[2] but can perform certain program transformations impossible with today's aspects and bytecode transformations, due to limitations of aspect languages and the restrictive Java bytecode format. Their design can provide insight into what features are lacking in source languages.

[2]The (non)portability of compiled programs depends on *which* compiler is modified—a Java source-to-bytecode compiler, a Java "ahead-of-time" compiler (source-to-native), or a Java JIT compiler.

Chapter 6 describes a project in which a production compiler was modified to support a hardware device. An aspect approach works *conceptually* but is impractical using current languages and tools. This approach is almost entirely dynamic, with little work occurring at compile time except the identification of join points. There is a large runtime penalty, but it is required given the design requirements of the project.

3.2 Chapter summary

This chapter introduced the notion of a *system aspect*, an aspect that operates on the language and runtime system itself and has pervasive, systemic reach. We also described alternative, existing mechanisms that can be used to perform these systemic transformations. Chapters 4, 5, and 6 will describe problems for which a natural aspect solution exists. However, for varying reasons, the aspectual formulation using current tools is problematic, and one of the alternatives will be used instead.

Each of these chapters describes a self-contained contribution independent of their aspect-oriented formulation. To tie them together and to forward the main thrust of this dissertation, Chapter 7 concludes this part of the dissertation by making observations about the solutions in this part and highlights some lessons for aspect language and tool design.

Chapter 4

Rate-Monotonic Metaprogramming

We describe an implementation of Rate-Monotonic Analysis (RMA) within the C++ parametric type system that provides C++ real-time software developers a good way to reason with types at the source level about recurrent tasks and deadlines. Using our approach, a program can be considered *incorrect*, raising type errors at compile time, if a given set of tasks is not statically schedulable. Similarly, this compile-time *metaprogram* can adjust a task set so as to become feasible; we perform this analysis inside the C++ type system, which allows a very natural integration into C++ programs. We discuss our approaches and the applicability of our work to the model-driven development of real-time embedded systems.

This compile-time metaprogram is similar to an aspect in an aspect-oriented language. However, current aspect tools like AspectJ, as discussed further in Chapter 5, don't support user-directed, compile-time computation. This often leads to a runtime performance overhead even when the computation can be performed statically.

This chapter studies an aspect-like but non-aspectual formulation of task scheduling. It is a contribution apart from its aspectual nature; we make observations about its aspectual nature in Chapter 7.

Acknowledgments

This chapter expands upon work performed jointly with Chris Gill and Ron Cytron. That work was originally published in 2003 [39].

4.1 Introduction

Real-time embedded systems have specific timeliness requirements that result in the necessity of *scheduling* tasks' access to scarce resources. Rate-Monotonic Scheduling (RMS) is a well-known static scheduling technique in which periodic tasks are assigned priorities in accordance with their period: more frequent tasks receive a higher priority. A runtime schedule honoring RMS-assigned priorities is known to be an optimal schedule for the fixed-priority scheduling problem [76]; that is, if any assignment of fixed priorities yields a feasible schedule, the RMS assignment will.[1] RMA refers to the computation performed on a set of periodic tasks to determine whether they may be statically assigned fixed priorities with RMS (or indeed with any such scheme, since RMS leads to an optimal schedule with respect to feasibility) and meet all deadlines.

As originally stated by Liu and Layland [76], a set of m periodic tasks has utilization:

$$U = \sum_{i=1}^{m} \frac{C_i}{T_i}$$

where C_i is the execution time budget, or *cost*, of task i on some machine and T_i is the execution *period* of task i. A task set is feasibly schedulable with RMS *if*

$$U = \sum_{i=1}^{m} \frac{C_i}{T_i} \leq m\left(2^{1/m} - 1\right) . \tag{4.1}$$

This is a computationally simple test, and can easily be performed (even manually) for a given set of tasks. However, this test is pessimistic, disqualifying task sets that are, in fact, feasible. Lehoczky, Sha, and Ding offer a stricter test [71, 95].[2] A set of m periodic tasks is feasibly schedulable *if and only if*

[1] In this chapter, we intend "feasible" to mean that all tasks are guaranteed to meet all deadlines, over all possible task phasings. The deadline of a task in classical RMS is the start of its next execution period.

[2] The proof is found in [71]; a useful discussion appears in [95].

$$\forall\, i,\ 1 \le i \le m,$$

$$\exists\, t \in \left\{ l \cdot T_k \;\middle|\; 1 \le k \le i,\ 1 \le l \le \left\lfloor \frac{T_i}{T_k} \right\rfloor \right\}$$

$$s.t. \quad \sum_{j=1}^{i} C_j \left\lceil \frac{t}{T_j} \right\rceil \;\le\; t \ . \tag{4.2}$$

When engineering a real-time system that makes use of static scheduling, such tests are typically performed on a set of proposed tasks ahead of time, often long before compilation—even in the design phase, *e.g.*, through model-integrated computing tools—to secure a guarantee that they will meet their deadlines. This may be acceptable if the task set is known in advance and does not change through the software development process. However, for purposes of debugging and design flexibility, a solution that integrates compilation with RMA task set verification is desired so that the task set can easily be modified. Further, for retargettable, reconfigurable real-time systems, software development teams often wish to provide similar systems meeting slightly different design requirements and manage all such configurations using a modeling tool. Clearly, this goal is unnecessarily complicated if the software is designed in a rigid manner for a specific set of tasks.

One solution to this problem would be to compute feasibility of the task set at runtime. Indeed, this approach is taken by some systems [21]. However, the main benefit of static scheduling over dynamic scheduling is its simplicity and low overhead. At worst, the only computation required at runtime for a fixed-priority periodic scheduling mechanism is the comparison of eligible tasks' priorities; at best, the processor is scheduled in a sequential fashion and scheduling and context switches are free.[3]

Because runtime feasibility checks are not required for many real-time systems, we do not seek to require them in a new system for real-time software development. At the same time, we wish to ease the development process by allowing the task set to change with each compilation, yet require that compiled programs are indeed feasible. In Section 4.3 we propose a system that uses the C++ compiler to perform feasibility testing as part of program translation. We extend the basic idea in Section 4.4 to show that our technique can be used to enforce that every correct program is feasible—that is, a semantic error is flagged by a standard-compliant C++ compiler when infeasible task sets are specified by the program—and to search a parameter space of different task rates for feasibility.

[3]Task sets are scheduled most easily when the task rates are harmonic; such task sets also have the benefit of achieving 100% utilization.

This chapter is organized as follows. Section 4.2 gives a brief primer on the specific technologies we employ, Section 4.3 explains our approach, Section 4.4 discusses some useful extensions to our base technique, Section 4.5 points to some related work, and Section 4.6 offers some conclusions and our thoughts on future research directions in this area.

4.2 Principles

In our approach, described in Section 4.3, we use advanced C++ [99] features and freely-available libraries, and we apply idioms from generic programming. We briefly document these language constructs and design ideas here.

4.2.1 Templates

Templates provide the C++ *generic types* facility. Each *template type*[4] does not, in itself, define a data structure or executable code; it does, however, define a number of *template parameters* that may be filled in by concrete types or templates. When the compiler detects the use of a template with all parameters fully specified, the template is said to be *instantiated* with those parameters; there is only one instantiation for each distinct set of parameters, and each template instantiation is an actual type—it can be directly constructed—although it is incompatible with other instantiations of the same template. As an example, consider a *linked list* template:[5]

```
template <class T>
struct list {
  T value;
  list<T> *next;
};
```

as well as the use of this template for integer types:

[4] For brevity, we ignore *template functions* in this section.

[5] *Note:* All C++ code examples in this chapter have been tested and compile properly on the GNU C++ compiler version 3.2.

```
list<int> *l = new list<int>;
l->value = 5; // l->value is an 'int'
l->next = new list<int>;
l->next->value = 2;
l->next->next = 0; // etc...
```

This mechanism provides a simple mechanism for reconfigurable software modules; templates provide a convenient way to write code that works for both integral and floating-point types, to implement the *Strategy* design pattern, and myriad other idioms. Because templates are instantiated and resolved at compile-time and guaranteed well-formed by the C++ typechecker, template instantiations like the above generally have minimal runtime execution-time impact; templates can, however, lead to highly redundant code segments in the produced binary—and, thus, much higher storage costs—which is critical for embedded systems. For further information on C++ templates, see [99].

4.2.2 Template metaprogramming

By manipulating constant values and template instantiations, you can perform computation at runtime. This is termed *template metaprogramming* [1] and is an incredibly powerful concept that allows a C++ programmer to write nicely modularized code without runtime overhead—because all of the necessary computation is performed at runtime. The best explanation here is an example, and the commonly-used example is the compile-time computation of primes, originally proposed by Unruh while serving on the C++ standardization committee in 1994 [107, 108]:[6]

[6]This listing shows only part of Unruh's prime template metaprogram. The full metaprogram fools the compiler into generating an error message for each prime, which has the effect of communicating the result of the prime calculations to the user.

```
template <int p, int i> struct is_prime {
 enum {
   prim = ( (p == 2) ||
            ( (p % i) &&
              is_prime< (i > 2 ? p : 0),
                        i-1 >::prim)) };
};

template <> struct is_prime<0,0>
  { enum { prim = 1 }; };
template <> struct is_prime<0,1>
  { enum { prim = 1 }; };
```

This demonstrates the power of the C++ template mechanism; not only does it provide generic types, it offers computation. By combining types, runtime code, *and* such template computation into a single framework, powerful reconfigurable software can be built.

4.2.3 Type traits

Traits [10] are a C++ template programming idiom in which information about a type is stored not in the type itself, but in different template instantiations off to the side. Definitions of traits make use of *template specialization*, in which a generic traits template is *specialized* for each type for which it serves information. For example:

```
template <class T> struct MyTraits;

template <>
struct MyTraits<MyType> {
  static const char* desc =
    "This is my type.";
};

template <>
struct MyTraits<MyOtherType> {
  static const char* desc =
    "This is another type of mine.";
};
```

Essentially, then, traits provide compile-time mappings from a type (or compile-time integral constant) to any collection of types and constants. These mappings are external to the type itself, so can be "added on" at any point textually in the program that a struct definition is permitted and in any C++ namespace.

4.2.4 Typelists

Alexandrescu [1] defines a Typelist:

```
template <class T, class U>
struct Typelist {
  typedef T Head;
  typedef U Tail;
};
```

The simplicity of this definition is quite deceiving. This construct can be used to allow arbitrary parameterization of template code with zero runtime execution overhead.[7]

Because types can contain concrete values (given that they are compile-time constants), we can perform compile-time computation with Typelists. For example, to sum an arbitrarily long list of numbers, one can write the code in Figure 4.1. This code also demonstrates the use of `TYPELIST` macros, which simply expand to a sequence of scoped pairs of Typelist template instantiations, similar to list construction in Lisp with a chain of `cons` forms. Although this is a simple example, this basic construct will be built upon in Section 4.3 to perform flexible RMA on a reconfigurable task set using a template metaprogram.

4.2.5 Functors

Functors [10] are an abstraction for an operation. C++ function pointers are simple Functors, but any type that supports parenthetical application—`operator()`—models a Functor. A Functor concept is defined by its input argument types and its return type, although it may also specify semantic requirements and guarantees.

4.3 Approach

We have implemented a template metaprogramming framework, coded in C++, that performs rate-monotonic analysis at compile-time and enables code to reflect at compile-time

[7] As with all templates, each instantiated template type is a separate type and can lead to increased runtime footprint. However, we primarily will use Typelists only for compile-time computation, where each instantiated template contains no data members and is thus zero size at runtime.

```
template <int i, class Tail>
struct sum<Typelist<value<i>, Tail> > {
  enum { result = Head::value + sum<Tail>::result };
};

template <>
struct sum<NullType> {
  enum { result = 0 };
};

template <int i>
struct value<i> {};

const int mysum = sum<TYPELIST_3(value<4>,
                                 value<5>,
                                 value<7>)>::value; // = 16
```

Figure 4.1: A C++ template metaprogram to sum an arbitrary-length number of integers at compile time. Note the use of the macro `TYPELIST_3` to enhance readability of the code. Writing `TYPELIST_3(a, b, c)` is equivalent to writing `Typelist<a, Typelist<b, Typelist<c, NullType> > >` — in fact, it will be replaced by the C preprocessor in exactly this way.

upon its task sets and reason about their feasibility. Generally, we believe compile-time "reflection" of this sort—which does not require runtime support—to be valuable in C++ real-time software development. We use the technique to achieve the following specific requirements:

- real-time tasks can be specified as optional;
- "cheap" task sets that have the critical features of standard task sets can be linked to their more "expensive" versions;
- the "best-fit" versions of expensive services can be automatically selected and compiled in with no user intervention or runtime penalty in time or space or the size of the executable; and
- truly infeasible task sets can be automatically rejected; if there is no guarantee that a task set can be scheduled, the compiler can be used signal an error.

We provide details on these particular aspects of our approach in the rest of Section 4.3 and in Section 4.4, but the above list is not an exhaustive one. First, we specify the base of our approach, which allows us to construct task sets and perform basic queries of them.

```
struct my_task {
  enum { cost       = 100,
         period     = 600,
         phasing    = 50,

         droppable  = 0,
         importance = 1000 };

  static void do_task(const context& c) {
    cout << "my_task::do_task()" << endl;
  }
};
```

Figure 4.2: A sample Task.

4.3.1 Specification

We define a generic programming *concept* [10] Task, implemented in C++ as a `struct`, which, along with zero or more associated TaskTraits providing additional, optional information (discussed in Section 4.4.1), fully specifies a *periodic real-time task*. A Typelist [1, 2] of Tasks then describes a *task set*. In addition to the standard parameters that we need to perform RMA for each periodic task (*i.e.*, task cost and period), we include other useful information for scheduling the task. A sample Task definition is shown in Figure 4.2.[8] Its elements are:

cost specifies the logical *cost* of the Task. This may be a measurement on a particular platform or a theoretical upper-bound, calibrated to agree with the other time-based parameters below.

period specifies the logical *period* of the Task.

phasing specifies the logical *phasing* of the Task. This is the offset of the logical clock at which its logical period begins.

deadline specifies the logical *deadline* of the Task, measured from its logical time of release. For RMS, the deadline of a task equals the period.[9]

droppable is a boolean value indicating whether or not a task can be dropped if necessary to make its task set feasible—this value, in effect, declares whether or not the task is optional.

[8] *Note:* All C++ code examples in this chapter have been tested and compile properly on the GNU C++ compiler v3.2.2 [48].

[9] *Deadline-monotonic scheduling* has been proposed to relax this constraint [9]. Our task model could be extended to relax it as well, though that is outside the scope of this work.

importance is an integer value specifying the relative willingness of the compile-time scheduling analysis to drop the task. Tasks with lower importance are dropped before higher-importance tasks.

do_task is a Functor [10] that specifies the work to be performed by the Task.

Once the basic structures defining tasks have been built, task sets can be constructed using `typedef`:

```
typedef TYPELIST_2(taskA, taskB) my_tasks;
```

In this case, a task set type (called `my_tasks`) of two independent task types is constructed: `taskA` and `taskB`.

4.3.2 Operation

We then wish to perform basic operations on this task set. These operations include:

- sorting the task set by period;
- determining the schedulability of such a task set;
- generating code to schedule the task set at runtime; and
- querying on the task set regarding its constituent tasks, its feasibility, and its utilization.

Further, we wish to perform these operations at compile-time to the fullest extent possible. Obviously, the tasks will actually *execute* only at runtime, but we wish to perform the queries and other operations above at compile-time. We also wish to expand and inline a specialized `start()` routine specifically for this task set so that starting the tasks has as little overhead as possible. Finally, we want the associated structures and queries to be reasonably easy and intuitive to use. By offering an interface to user code in the metaprogram, we introduce a mechanism similar to compile-time *structural reflection* into a real-time program. Using this facility, a real-time programmer can write code that is easy to read and reconfigure despite being tailored for a particular task set. In effect, the task set introduces various *constraints* onto the program, and the C++ compiler (by evaluating the template metaprogram) is able

to resolve these constraints and generate a specialized executable, even though the source code remains modular and generic.

Fortunately, these operations can all be performed by manipulating the task set with a template metaprogram. In this chapter, we focus on the last operation: determining the feasibility and expected utilization of a task set and integrating this with the program. We define a `Schedule` template, shown in Figure 4.3. This template calls an `RMA_Feasible` template metaprogram shown in Figure 4.4. This metaprogram solves inequality (4.2) directly, for each i, by trying different values of t as necessary. It utilizes the support templates of Figure 4.5, which compute the set of all $l \cdot T_k$ and the j-summation.

```
template <class TaskSet> struct Schedule;

template <class Head, class Tail>
struct Schedule<Typelist<Head, Tail> > {
  typedef Typelist<Head, Tail> TL;
  enum { feasible=RMA_Feasible<TL>::Result };
  static const double utilization =
         Schedule<Tail>::utilization +
         double(Head::cost) / Head::period;
  static void schedule(void) {
    /* (not shown) */
  }
};

template <>
struct Schedule<NullType> {
  static const bool Result = true;
  static const double utilization = 0.0;
  static void schedule(void) {
    // no action necessary
  }
};
```

Figure 4.3: The `Schedule` template.

Given this metaprogramming mechanism, client code using our framework can then be specified in a very straightforward manner (Figure 4.6). The `schedule()` method of the `Schedule` template (implementation not shown in this chapter) is used to set up the proper threading mechanism for a specified compilation target and invokes the `do_task` routines of the task set's constituent task types as appropriate. Because this can be inlined, no runtime overhead need exist for permitting this flexibility of task types as template parameterization, as this is sorted out by the C++ compiler at compilation time. Providing the task-invocation capability in a parameterized fashion (which could automate the choice

```
template <class TL, int m, int i>
struct check_i;

template <class Head, class Tail,
          int m, int i>
struct check_i<Typelist<Head, Tail>, m, i> {
  enum { task_result =
           task_feasible<Typelist<Head,Tail>,
                         i>::Result,
         Result = check_i<Typelist<Head,Tail>,
                          m, i+1>::Result
                  && task_result };
};

template <class Head, class Tail, int m>
struct check_i<Typelist<Head, Tail>, m, m> {
  enum { Result =
           task_feasible<Typelist<Head,Tail>,
                         m>::Result };
};

template <class TaskSet>
struct RMA_Feasible {
  enum { m = Length<TaskSet>::value,
         Result = check_i<TaskSet,
                          m, 1>::Result };
};
```

Figure 4.4: The main "loop" of the `RMA_Feasible` template metaprogram.

of threading model, for example) is the subject of ongoing work and is not described in this chapter. In Section 4.3.5, we describe a way to cause a compiler error if an infeasible schedule is encountered.

4.3.3 A walkthrough example

As an example of how this template expansion works,[10] consider the task set of Figure 4.7. In this case, tasks `taskA` and `taskB` have only costs and periods—for simplicity of the example, other parameters have been omitted from the listing.

In evaluating the `RMA_Feasible<my_tasks>` template instantiation (at the bottom of Figure 4.7), we must direct the C++ compiler to check that inequality (4.2) holds for

[10]The discussion of this section is by necessity abbreviated and imprecise. The reader is referred to [99] for a more careful treatment of this material.

each task i in our example task set. We do this by first counting the number of tasks in the set, then instantiating another template (`check_i`) to perform these checks individually. The `check_i` instantiation is parameterized by the value of i it is to check; but `check_i` recursively makes another instantiation of `check_i` with the next value of i, so `RMA_Feasible` only needs to instantiate a single `check_i`. The result of these checks are composed together with logical *and* ($\wedge$), since each sub-check must be satisfied for the task set to be feasible. In this way, the final computed feasibility of the task set is dependent upon the feasibility of each sub-check.

In our example, the size of the task set is calculated to be 2; `check_i<my_tasks,2,1>` is instantiated. This instantiation does two things: it computes the check for $i = 1$ (by instantiating `task_feasible<my_tasks,1>`), and, later, it will compose its result with that of the *next* `check_i`.

The `task_feasible` template's job is to find, given a fixed i and task set, a value of t for which inequality (4.2) holds. To do this, it must try successive values of t, chosen from the appropriate set, and compute the summation over $1 \leq j \leq i$. It uses two other templates to accomplish this—`get_t<my_tasks,1,0>`, which gets the "first" value of t (subject to an arbitrary ordering we impose on the set, discussed below), and `sum_j<my_tasks,1,t>` to compute the summation (once t has been computed).

Therefore, in our running example, we have at this point `RMA_Feasible<my_tasks>` instantiating `check_i<my_tasks,2,1>` instantiating `task_feasible<my_tasks,1>` instantiating `get_t<my_tasks,1,0>`.

`get_t`'s purpose is to compute and return a value of t based on an index (the `t_ix` parameter). This indexing scheme is arbitrary—we choose it to start with $(k = 1, l = 1)$ and increase to the maximal (k,l) value pair in the set.[11] The code of `get_t` (which has an implicit $k = 0$ parameter if unspecified) first gets the period of tasks i and k and computes the maximum value permitted for l for the given k (see inequality (4.2)). The value of t is then computed by instantiating `get_t` to service the next-larger value of k, or, if this instantiation has a sufficient k to service index `t_ix`, then it returns the value directly (which corresponds to $l \cdot T_k$ in inequality (4.2)).

[11] The implementation actually uses zero-based indexes.

In our running example, `get_t<my_tasks,1,0>` computes `Ti` $= 10$, `Tk` $= 10$, `num_l` $= 1$, and `Result` $= 10$. Therefore, `task_feasible<my_tasks,1>` uses $t = 10$, and thus instantiates `sum_j<my_tasks,1,10>`.

The `sum_j<my_tasks,1,10>` instantiation is straightforward. First, notice that such an instantiation uses the default parameter $j = 0$—the summation will be recursively computed by recursively instantiating `sum_j`, and $j = 0$ serves as the entry to this recursion. The jth task (`A` in our example) is given an alias `J`, and `Cj` and `Tj` get the values for its cost and period, respectively. `my_result` is computed (this is $C_j \cdot \lceil t/T_j \rceil$), and the result is summed together with further instantiations of `sum_j`.

Finally, `task_feasible<my_tasks,1>` performs its computation by checking to see if this sum is less than or equal to t, as required in inequality (4.2); if this test fails, it creates another `task_feasible` for another value of t. The computation continues along similar lines, and the task set is ultimately determined feasible by the compiler.

4.3.4 Tasks as types

Our system models tasks as C++ types. Type systems are typically used in high-level languages to help ensure that the logical intent of the programmer matches the code as written. Generally, developers have types in mind when designing and writing programs, and making this explicit in a language can flag logical errors that are difficult to track down otherwise. We provide something analogous for real-time developers; with our constructs, various nonfunctional aspects of the program (in this case, task schedulability guarantees) become part of the structure of the program. The next section demonstrates how to signal type errors for infeasible task sets.

4.3.5 Feasibility and program correctness

Using techniques developed by Alexandrescu [1] and embodied in the Loki C++ library [2], we can enforce the requirement that a particular task set declared in a program is always feasible. We do this using the `STATIC_CHECK` macro of Loki, which conditionally raises a C++ type error:

```
typedef Schedule<TYPELIST_2(taskA, taskB)>
    my_schedule;
STATIC_CHECK(my_schedule::feasible,
             Schedule_Infeasible);
```

The `Schedule_Infeasible` macro parameter is a description string—typically, compiler output indicates this description in its error listing. The GNU C++ compiler v3.2.2 [48], for example, gives the following useful output if `my_schedule` is infeasible:

```
mysched.cc: In function ‘int main(void)’:
mysched.cc:20: aggregate
   ‘Loki::CompileTimeError<0>
     ERROR_Schedule_Infeasible’ has
   incomplete type and cannot be defined
```

Using this technique, a global policy can be enforced that requires every task set to be feasible. In this case, the use of the `STATIC_CHECK` macro is placed in the `Schedule::schedule()` method.[12] This will verify that every task set that could be scheduled at runtime is feasible.

4.4 Extensions to the base model

It is possible to add a number of useful extensions to our base model. We discuss here our ideas regarding *enhanced tasks* and *searching a feasibility space*.

4.4.1 Enhanced tasks

We have found it useful to use template metaprograms to specify task dependence and task alternation. By using *traits* (as briefly described in Section 4.2.3) we can make such enhancements without changing our base code or the requirements of the Task generic-programming concept as specified in Section 4.3.1.

Task dependence refers to interdependence of tasks within a task set. It is important to note that RMA assumes independent tasks. We do not break that assumption here because our notion of dependence is not a dependence on a particular computational result; rather, a dependence of task A on task B is merely a requirement that any task set including

[12] An additional template parameter to the `Schedule` template can be used to achieve maximal flexibility in specifying such a policy.

task A must also include task B. This can be flexibly used to group tasks into common configurations, or to model execution dependence loosely.[13] However, since synchronization is not taken into account in classical RMA, any computational dependence should only be a dependence upon the generated value guaranteed to complete before the start of the task performing the computation. Dependence is easily represented as a trait (Figure 4.8). For each type T modelling the Task concept that has one or more task dependencies, a `task_dependencies` template specialization is written for the type specifying as a Typelist the tasks upon which T depends.

Task alternation allows one task to be readily "swapped out" for another, cheaper task. This can be quite useful, especially for optional, debugging, or logging tasks that are not critical but are nice to include when other tasks do not "starve them out" of feasibility. Basically, the idea is to check the programmer-specified task set for feasibility; if the task set is infeasible, the least important task in the task set is exchanged for a cheaper alternative or dropped (if the task concept is specified as *droppable*). This process continues until either the task set becomes feasible or an infeasible task set is reached in which no constituent task can be exchanged or dropped. Alternation is traited simply (Figure 4.9).

4.4.2 Searching a feasibility space

It is also possible to search a feasibility space to discover potential rates for tasks in a given task set. By performing a feasibility search over task sets with different task costs and periods, suitable task frequencies can be singled out and ones of interest can be metaprogrammatically chosen and applied. When composed by a metaprogram with suitable models of target platforms (expressed as generic programming concepts), this technique can be quite powerful.

This idea can be extended to a dynamically scheduled system as well, in which the actions of the dynamic scheduler are partially evaluated—for example, resource credits can be partially handed out at compile-time to reduce the cost of scheduler start-up.

[13] There is a correspondence here to the valid configurations of Chapter 8.

4.5 Related work

Template metaprogramming has been used for fast Fourier transforms [109], prime number computation [107, 108], and many other computations. Our work is similar but brings metaprogramming techniques to the compilation of real-time programs.

Other analysis tools are commercially available for real-time applications using RMS. TimeWiz [104] from TimeSys supports graphical modelling, analysis, and simulation of real-time software, and RapidSched [116] performs its real-time task analysis in a front-end for TriPacific Software's PERTS [78]. Our approach is oriented toward analysis rather than graphical modelling or simulation; however, it does the analysis inside the language itself and requires no additional tools. Further, our approach automatically stays in-sync with the program: it is *part* of the program. We do not dictate a way of arriving at an estimate for the execution budget of a task; such an estimate could certainly be reached using the analysis or simulation modes of such tools, or by other means.

4.6 Chapter summary

We have described and presented code for a compile-time Rate-Monotonic Analysis (RMA) computation performed within the parametric type system of standard C++. We specify tasks and task sets as types to gain flexibility, and we leverage template metaprogramming mechanisms to compute feasibility of these task sets and to perform additional functions. Because our approach is entirely within the C++ language itself, we achieve complete integration with the language without a requirement of preprocessing or translation from a higher-level language. Thus, with minimal effort, real-time software developers implementing periodic task sets in C++ can apply our techniques to gain flexibility and retargetability, organize tasks into groups, easily specify task dependence and alternation, and reason metaprogrammatically about processor utilization and schedule feasibility, all within the language.

This implementation is in C++, *not* in an aspect-oriented programming language. We could implement this in AspectJ, but in doing so we cannot get the benefits of the C++ RMS template metaprogram—namely, we'd pay a price at runtime despite the fact that this computation can, in principle, be performed at compile time. Many aspect-oriented tools, including AspectJ, are aimed primarily at runtime flexibility and don't provide a way

to perform type computation of the sort exploited in this chapter. Chapter 7 continues the discussion of aspectual limitations.

```
template <class TL, int i, int t, int j = 0>
struct sum_j {
  typedef typename TypeAt<TL, j>::Result J;
  enum { Cj = J::cost,
         Tj = J::period,
         my_result = Cj * ((t%Tj > 0 ? 1 : 0)
                           + (t / Tj)),
         Result = sum_j<TL,i,t,j+1>::Result
                  + my_result };
};

template <class TL, int i, int t>
struct sum_j<TL, i, t, i> {
  enum { Result = 0 };
};

template <class TL, int i, int t_ix, int k=0>
struct get_t {
  enum { Ti = TypeAt<TL,i-1>::Result::period,
         Tk = TypeAt<TL,k>::Result::period,
         num_l = Ti/Tk,
         Result = (t_ix >= num_l)
           ? get_t<TL, i,
               t_ix - num_l, k+1>::Result
           : (t_ix + 1) * Tk };
};

template <class TL, int i, int t_ix>
struct get_t<TL, i, t_ix, i> {
  enum { Result = 0 };
};

template <class TL, int i, int t_ix = 0>
struct task_feasible {
  typedef get_t<TL, i, t_ix> t_type;
  enum { t = t_type::Result,
         Result = (t > 0) &&
           ( sum_j<TL, i, t>::Result <= t
             || task_feasible<TL, i,
                  t_ix + 1>::Result ) };
};

template <class TL, int i>
struct task_feasible<TL, i, i> {
  enum { Result = 0 };
};
```

Figure 4.5: Supporting templates for the `RMA_Feasible` template metaprogram of Figure 4.4.

```
typedef Schedule<TYPELIST_3(
        taskA, taskB, taskC)> my_schedule;
if(! my_schedule::feasible)
  cerr << "WARNING: infeasible!" << endl;
my_schedule::schedule();
```

Figure 4.6: Instantiating and using the `Schedule` template.

```
struct taskA { enum { cost       = 5,
                      period     = 10 }; };
struct taskB { enum { cost       = 5,
                      period     = 15 }; };
typedef TYPELIST_2(taskA, taskB) my_tasks;
const int isFeasible =
       RMA_Feasible<my_tasks>::Result;
```

Figure 4.7: An example task set specification and its feasibility test.

```
// default case, no dependencies
template <class Task>
struct task_dependencies {
  typedef NullType dependencies;
};

// sample specialization for My_Task
template <>
struct task_dependencies<My_Task>
  typedef TYPELIST_2(
              My_Second_Task,
              My_Third_Task) dependencies;
};
```

Figure 4.8: Specifying traits for task dependence.

```
// default case
template <class Task>
struct task_alternative {
  // "NullType" means no alternative
  typedef NullType alternative;
  enum { importance = 0 };
};

// sample specialization for My_Task
template <>
struct task_alternative<My_Task> {
  typedef My_Cheaper_Task alternative;
  // importance relative to other tasks
  enum { importance = 100 };
};
```

Figure 4.9: Specifying traits for task alternation.

Chapter 5

Reference-Counting Aspects

Most programming languages provide dynamic storage allocation, and many languages offer garbage collection. While the popularity of languages like Java can be partially attributed to its automatic storage management, there are applications and platforms for which traditional garbage collection is inconvenient, inefficient, or unavailable.

We introduce an aspect-oriented reformulation of reference-counting that is particularly well-suited to Java applications and does not share the error-prone characteristic of manual, user-driven reference counting. We present our method in the context of the Real-Time Specification for Java [21] (RTSJ) and demonstrate that it can recycle dead objects in bounded time. We apply automatically-generated, type-specific advice that has the effect of *partially evaluating* aspect-generated code, which substantially reduces the reference-counting overhead.

Acknowledgments

This chapter expands upon work performed jointly with Nick Leidenfrost, Matt Hampton, James Brodman, and Ron Cytron. That work was originally published in 2004 [40].

5.1 Introduction

Along with its collections library and familiar syntax, Java's [5] automatic memory management has helped make it an attractive platform for application development. In particular, Java's garbage collector has arguably increased productivity by automating the detection and deallocation of dead storage—a difficult and error-prone task. These factors, along with its widespread portability, have catalyzed the language's adoption to a variety of platforms, ranging from server applications to embedded, real-time systems.

This popularity of Java, coupled with advances in real-time computing technology, has motivated development of a standard that would help express the concerns of a real-time system. The Java community responded with the RTSJ [21], which attempts to match the needs of real-time programming with Java through Java Virtual Machine (JVM) [74] and library extensions. The RTSJ makes no changes to the Java language itself, and standard Java source-to-bytecode compilers can be used for programs intended for an RTSJ platform.

For better or worse, the RTSJ forbids threads with real-time guarantees from touching the garbage-collected heap [21]. As a result, programmers must organize their objects in *scoped* storage regions, each of which is deallocated as a block (with the constituent objects collected *en masse*) when the associated execution scope completes. There is no system-mandated garbage collection within a scoped storage area. Moreover, scoped allocation and deallocation are unsuitable for a wide range of applications, and there exist nontrivial programs with bounded live storage but with unbounded dead storage in scopes [106].

However, the standard cannot prevent an application from identifying dead objects on its own and recycling such objects for the application's use. While this is a tricky and error-prone undertaking, we discuss in this chapter how AOP techniques can automate the task. Our contributions are as follows:

1. We present an aspectual formulation [67] of object reference-counting [113], suitable for inclusion with ordinary Java programs (Section 5.2). To enable reference counting for a given class, that class need only implement the (empty) interface *ReferenceCountable*.

2. Our reference-counting approach is tailored for AspectJ [66] in the sense that local variable modifications cannot be traced within AspectJ, and our approach does not require such tracing. We offer a heap-only reference-counting scheme (Section 5.2.1)

that must be used carefully. We also introduce a safe, conservative approximation of stack (local variable) reference activity that avoids tracing all local references (Section 5.2.2).

3. We present an algorithm for automatically determining which Java classes can usefully implement the *ReferenceCountable* interface (Section 5.3).

4. We present an aspectual formulation of object *recycling* [26], so that the storage associated with a dead object is saved and can be recycled when a new object of the dead object's type is instantiated (Section 5.4).

5. We present a scheme for *partially evaluating* [61] the aspect-generated code to eliminate runtime reflection (Section 5.5). Without such optimization, AspectJ requires each store to an object field to use relatively expensive (Section 5.6) reflection. The overhead of this reflection is significant in our benchmarks

While our work was specifically motivated by the RTSJ memory model, it can also be used in conjunction with a garbage-collected heap to reuse instances of heavily-instantiated classes. We demonstrate on some benchmarks that we can compete with a garbage collector (Section 5.6).

5.1.1 Related Work

In considering the needs of real-time programs, the RTSJ authors eschewed automatic garbage collection of any form, because of the impact it might have on real-time systems. Garbage collection schemes suitable for real-time systems have been proposed [30, 85, 88, 13]. Some require extra storage or processors. Moreover, these algorithms must be provided with some statistics about the application, such as its allocation rate and the number of non-null pointers on average, if real-time bounds on collection are to be maintained. Our approach reduces application reliance on a garbage collector within a JVM.

5.1.2 Treatment of dead storage

When automatic garbage collection finds dead storage, such storage is typically returned to a generic heap for subsequent reallocation. While this approach is the most general, programs

tend to allocate the same types of objects repeatedly. Due to current trends in object-oriented programming, such as the encapsulation of software patterns in objects, short-lived objects such as *Iterators* [49] often accomplish relatively simple tasks. Moreover, middleware may create "helper" objects without the knowledge of the end user—such objects act as carriers, strategies, and temporary-state storage. These objects, while useful in creating a well-designed implementation, are often abandoned almost immediately after instantiation, creating garbage each time the application executes their associated operation. While a reference-counting scheme may be able to collect this memory quickly, it cannot prevent the fragmentation to the heap caused by repeated allocation and deallocation.

One alternative to freeing the memory associated with such objects, and indeed, all types of objects, is to *recycle* them [26, 96]. When a new object is instantiated, its storage must be allocated and its header information must be initialized. If an object of a given type can be recycled, so that it is reallocated as an object of the same type, then the storage and header information need not be re-established. Thus, reference counting and object recycling are attractive in certain contexts. Manual introduction of code into an application to manage these cross-cutting concerns is tedious and error-prone. In this chapter we present an AOP approach for accomplishing both of these features.

This work has been implemented and tested in the context of collections objects for RTSJ.

The rest of this chapter is organized as follows. Section 5.2 provides more details about our approach, including its formulation, application, and limitations. Section 5.3 presents a simple algorithm for statically determining those objects that cannot be involved in reference cycles; the liveness of such objects can accurately be determined using reference counting. Section 5.4 describes how objects can be recycled rather than collected. Section 5.5 provides some implementation details, Section 5.6 provides an experimental indication of the benefits and limitations associated with our approach to reference counting and object recycling, and Section 5.7 offers a summary.

5.2 Approach

Recall that *reference counting* detects object liveness by tracking the sum of *live* references to a given object. When an object's reference count reaches zero, the application cannot

subsequently reference that object. The storage associated with such an object can be deallocated, which may, in turn, cause other objects' reference counts to decrease.

A reference-count tally is typically maintained incrementally: when a heap pointer-cell is changed, the reference count of the previously-referenced object is decremented and the count of the newly-referenced object is incremented. Conceptually, counting the references from the Java stack is similar. When an object reference is pushed (for example, through an `aload` instruction), the count is incremented; when an object reference is popped, the count is decremented. Counts within Java's registers ("local variables") can be similarly maintained; the distinction is largely unimportant for our purposes, and in our presentation we will assume that Java's registers are in fact part of the Java stack.

In our approach, we account for references inexactly. Our *approximate* object reference count is the sum of

- S—an approximation of references from the stack, and
- H—the exact number of references from the heap.

S may underestimate the *number* of stack cells that reference an object. However, with respect to object liveness, our approximation guarantees that S is positive for an object if the stack holds any references to that object. Thus, it is safe to base an object's collection on the reference-count value of $S + H$.

We chose an approximate accounting of stack references for several reasons. Most importantly, it can be expensive to exactly calculate S, as reference counts need to be modified to reflect the use of objects within a method. Further, in Java, the contents of a method's stack frame cannot be accessed or modified except by the method itself. As a result, object references from the stack are easily summarized and approximation can work well.

Section 5.2.1 describes heap-only reference counting and situations where it suffices. Our approximation S of object reference counts from the stack is formally described in Section 5.2.2, Section 5.2.3 discusses issues related to multithreaded target applications, and Section 5.2.4 applies our approach to a simple linked-list example. Section 5.2.5 discusses cyclic storage, and Section 5.2.6 gives details of our approach.

5.2.1 Heap-only reference counting

Because AspectJ and Java reflection cannot change the nature of Java's runtime stack,[1] we must simulate a program's stack frames in order to track the stack, essentially replicating the structure already present in the runtime image of an executing Java program. This simulation becomes even more expensive if multiple threads can have stack references to an object.

As a result, it is worth considering the conditions under which reference counting can be accomplished by ignoring stack activity completely and tracking only inter-heap references.

- The object will be considered dead when its reference count $S + H$ is decremented from 1 to 0.
- H is the only active component of the object's reference count: S remains 0 throughout.
- Thus, when H decrements from 1 to 0, there should be *no* non-heap references to the object.

With heap-only reference counting, failure to prevent non-heap references at the moment the reference count becomes 0 goes unchecked and can cause program failure if the storage associated with the object is reallocated or recycled. Conversely, an object will fail to be collected (its associated storage effectively leaked) if it is instantiated but never referenced from the heap. Given these considerations, it is clear that a heap-only approach to reference counting is appropriate only in specialized circumstances. If an object is appropriately protected, so that its reference cannot be exported out of a class or package, then the author of that class or package can make certain that no stack references exist when the last heap reference disappears.

Figure 5.1 shows the instrumentation we use to maintain a heap-only reference count and detect dead objects, based only on references to that object from the heap. Collection occurs when the H component of the reference count changes from 1 to 0. The advice of Figure 5.1 is applied on Java's heap-changing instructions, namely

[1] Future releases of Java may have introspective JVM features allowing this.

- those `putfield` and `putstatic` instructions which write an object reference into an object or class field; and
- the `aastore` instruction, which writes an object reference into an element of an array.[2]

Figure 5.1 is not AspectJ source; it is aspect pseudocode, representing instrumentation in an aspectual form. Three pieces of *before* advice are shown: the supplied code takes action *before* the bytecode performs its standard function. (For simplicity, bytecodes are shown with their full set of parameters here, instead of collecting some parameters from the instruction stream and some from the stack.) The *currentValue* gesture retrieves the current reference value from a field or array slot. The procedures *increment()* and *decrement()* maintain reference counts from the heap. The function *count()* returns the H component of the current reference count of an object. The *recycle()* procedure places an object onto a per-type list of freed objects for subsequent reallocation; this functionality is discussed in Section 5.4.

The full AspectJ source for heap-only reference counting is in Appendix A on pages 174–177.

5.2.2 Approximation of stack references

A formal description of our instrumentation for conservative, approximated stack-aware reference-counting appears in Figure 5.2.[3] It builds upon the heap-counting approach, adding instrumentation to the `getfield`, `getstatic`, `areturn`, and `athrow` bytecode instructions. Instrumentation on `putfield`, `putstatic`, and `aastore` instructions require an additional check to ensure that an object associated to a stack frame is not collected.

The full AspectJ source for stack-approximating reference counting is in Appendix A on pages 178–181.

S represents the number of stack references that point at an object. In our approach, the S component is not explicitly maintained, but is effectively 0 or 1, as follows.

[2]The current AspectJ language does not have a mechanism for advising array assignments. We feel that such a feature would fit with the other mechanisms supported by the language.

[3]This implementation is limited to single-threaded programs; it is easily generalized but incurs extra expense.

```
before putfield object class-field reference-value:
    if object = null
    | return
    fi
    increment(reference-value )
    decrement(object.class-field.currentValue)
    if count(object.class-field.currentValue) = 0
    | recycle(object.class-field.currentValue)
    fi
before putstatic class-field reference-value:
    increment(reference-value )
    decrement(class-field.currentValue)
    if count(class-field.currentValue) = 0
    | recycle(class-field.currentValue)
    fi
before aastore array index reference-value:
    if array = null or index out of range
    | return
    fi
    increment(reference-value )
    decrement(array[index].currentValue)
    if count(array[index].currentValue) = 0
    | recycle(array[index].currentValue)
    fi
```

Figure 5.1: Heap reference-counting instrumentation. This approach leads to prematurely-collected objects unless heap inter-referencing behavior guarantees safety.

```
after new class-type [using newly-created-object]:
    associate(newly-created-object, thisFrame)
before getfield object class-field:
    if object = null
    | return
    else if unassociated(object.class-field.currentValue)
    | associate(object.class-field.currentValue, thisFrame)
    fi
before getstatic class-field:
    if unassociated(class-field.currentValue)
    | associate(class-field.currentValue, thisFrame)
    fi
before aaload array index:
    if array = null or index out of range
    | return
    else if unassociated(array[index].currentValue)
    | associate(array[index].currentValue, thisFrame)
    fi
before areturn reference-value:
    foreach object x associated to thisFrame:
    | if x ≠ reference-value
    | | unassociate(x)
    | | if count(x) = 0
    | | | recycle(x)
    | | fi
    | fi
    end foreach
    associate(reference-value, prevFrame)
before athrow reference-value: same as for areturn
before propagate-exception reference-value:
      same as for areturn
before putfield object class-field reference-value:
    if object = null
    | return
    fi
    increment(reference-value )
    decrement(object.class-field.currentValue)
    if count(object.class-field.currentValue) = 0
    | if unassociated(object.class-field.currentValue)
    | | recycle(object.class-field.currentValue)
    | fi
    fi
before putstatic class-field reference-value:
    increment(reference-value )
    decrement(class-field.currentValue)
    if count(class-field.currentValue) = 0
    | if unassociated(class-field.currentValue)
    | | recycle(class-field.currentValue)
    | fi
    fi
before aastore array index reference-value:
    if array = null or index out of range
    | return
    fi
    increment(reference-value )
    decrement(array[index].currentValue)
    if count(array[index].currentValue) = 0
    | if unassociated(array[index].currentValue)
    | | recycle(array[index].currentValue)
    | fi
    fi
```

Figure 5.2: Heap reference-counting together with approximate stack reference-counting instrumentation. This approach is safe with respect to Java referencing behavior: it never determines that a live object is dead.

- If an object x *is not* currently associated to a stack frame, then *unassociated(x)* is *true*, no cell on the stack references the object, and S is effectively 0.
- If an object x *is* currently associated to a stack frame, then *unassociated(x)* is *false*, some stack cell *may* reference the object, and S is effectively 1; when the frame associated to x pops, S will either effectively remain at 1 because x is returned to the frame's caller (either by an explicit *areturn* or exception propagation), or else S will effectively become 0.

The *count()* function and the *increment()* and *decrement()* procedures only deal with the H-component of the reference count.

Throughout, we maintain a *stack association invariant*:

Stack association invariant:

If and only if an object x is associated with a stack frame F, then F is the last-to-be-popped frame with a stack cell holding a direct reference to x: the S component of x's reference count is 1, but there could be any number of stack cells in frame F and above (those popped earlier than F) that could reference x.

This invariant provides our system with *safety*: our approach will never determine that a live object is dead.

- When an object is instantiated (via *new*), it becomes associated to the stack frame of the currently executing method (called *thisFrame* in Figure 5.2). The invariant is thus satisfied for the object just after instantiation.
- When an object reference is retrieved from another object (a `getfield` or `getstatic` instruction) or an array (an `aaload` instruction), a heap cell containing a reference is copied onto the stack. Because the user program could subsequently extinguish all heap references to the object (its H-component dropping to 0 accordingly), we must ensure that the object's reference count has a positive S component. We therefore associate an object to the current stack frame if it is not currently associated to a frame. Since we never reassociate an object to a shorter-lived stack frame, the object is always associated to the last-to-be-popped stack frame that could reference it, maintaining the invariant.

- If an object x is associated with frame F, its S component must be 1, and the following happens when frame F pops.
  - If x is returned (or thrown) to F's caller, then S stays at 1 and x becomes associated with the caller's frame (*prevFrame* in Figure 5.2).[4] The invariant holds.
  - If x is *not* returned to F's caller, there are two subcases.

    If the H component of x's reference count is 0, x is dead and its storage can be immediately reclaimed.

    Otherwise, the S component of x's reference count becomes 0 but its H component is positive so x is still live.

    In either case, the invariant holds because x is no longer associated to any frame.
- The H component of x's reference count becomes 0 at the moment that nothing in the heap references the object. At that point, if x is not associated with any frame, it is immediately recycled; otherwise, it is considered live due to a (potential) reference from the stack. The invariant holds in either case.

Our aspect formulation instruments application code to track stack behavior as in Figure 5.2.

5.2.3 Multithreading and reference counting

Since multiple stack references to an object may exist with multithreading, our single stack frame approach cannot currently account for the effects of concurrent programming. Future work in simulating stack frames for each running thread will allow us to collect all objects in multithreaded programs. Multiple stack frames will add an additional S components to an object's reference count. Currently, our work only tracks one method stack. Objects shared between execution stacks of threads are assumed to live forever.

[4] An exception may not be caught by the caller, and may propagate some distance before being caught; we thus require *propagate-exception* instrumentation as well as *athrow* instrumentation (both listed in Figure 5.2) to associate the exception object with the correct frame.

5.2.4 A simple example

Consider three objects in a simple linked list as shown in Figure 5.3. Keeping with standard practice, the links between the objects are not present in the objects themselves but are instead held in *container* objects, depicted as large circles and given the names a, b, and c. Completing the structure is a `List` object, inside of which a *head* field either points to the first element of the linked list or is *null* to represent an empty list.

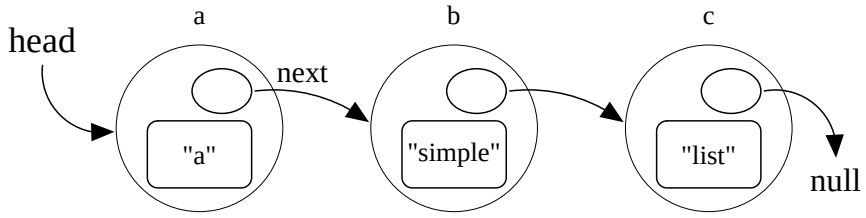


Figure 5.3: Linked list with carrier objects.

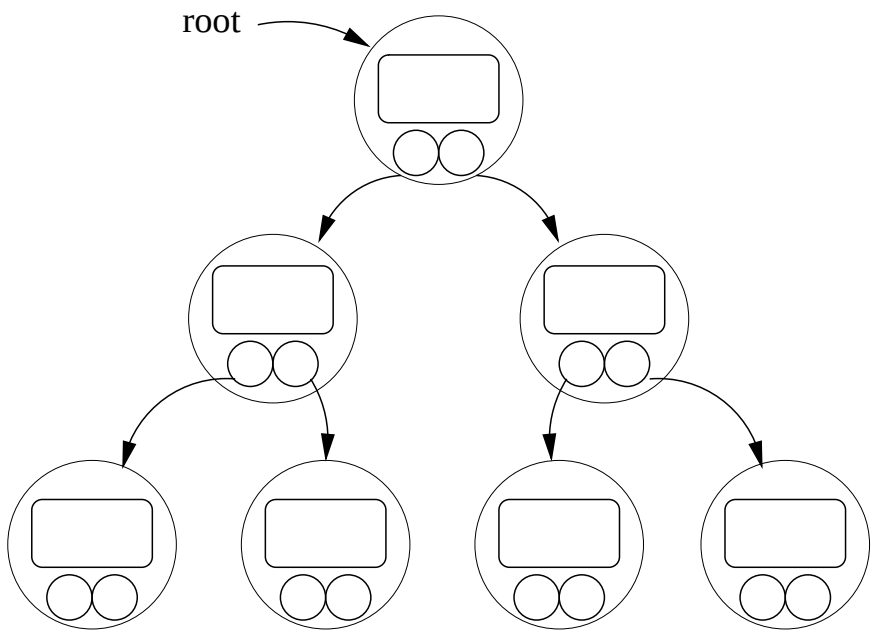


Figure 5.4: A tree with carrier objects.

Modern software-construction methodology favors hiding internal representation as much as possible, exposing functionality only through a published Application Programming Interface (API). For example, if the containers shown in Figure 5.3 were exported outside the list object, other classes could depend on their structure, making it more difficult to

modify the container class. Thus, list container-nodes are excellent examples of objects amenable to reference counting, and, in particular, our heap-only approach.

More ambitious implementations of lists use *backward* links, in which case the container objects become involved in cycles and cannot be collected using reference counting. However, it is often possible to identify references that cause cycles and make such references *weak* [50], if they do not need to contribute to the target object's liveness.

We next provide details on removal from and subsequent insertion to a singly-linked list, with and without reference-counted container objects.

- In Figure 5.3, each container object has a reference count of 1; the head of the list is referenced by the list itself, and each ensuing container object is referenced by its predecessor.
- When an element is removed from the list, the associated container object is no longer live. Removing the first element involves changing the list's head from a to b, which decrements the heap-reference count of a to 0. Since heap-only reference counting can safely collect these container objects, container object b becomes collectible at this point.
- When a new element is inserted into the list, a new container object is created to contain it. After creation, the new container object is spliced into the list by setting its *next* field and the *next* field of the container object preceding it. Setting these fields creates references from the heap, so the reference counts of the appropriate container objects must be incremented.

5.2.5 Cycles and weak references

While the approach described thus far suffices to collect most objects, the death of objects that reference each other in a cycle cannot be detected by reference counting. Other garbage-collection techniques [28] can handle objects in cycles, but in this section we describe another approach.

Java offers the notion of a *weak reference*, which is a reference from one object to another that does not count toward the target object's liveness. A typical use for a weak reference is in a hash table, so that liveness from the hash table does not contribute to the

liveness of keys or values stored in the table. If a key is not otherwise live, then it cannot be accessed in the hash table; thus, by making the reference from the table to the key *weak*, the object can be collected even though it is technically still reachable from the application's live roots.

Weak references could be used carefully in certain data structures to avoid cycles. For example, in a doubly-linked list, the backward links need not be counted toward objects' liveness, since liveness is just as well implied by the forward links. Thus, by making the backward links *weak*, reference counting could collect objects that otherwise appear to be live due to reference cycles.

5.2.6 Aspect implementation details

Aspect-Oriented Programming (AOP) provides *implicit invocation* [45]: when particular events ("join points") occur in the program code, code from elsewhere ("advice") is patched in to perform an additional (or alternate) computation.[5] Typically, AOP languages provide some sort of reflective facility so that advice can reason about the computation it is advising; advice triggered *after* (upon successful or exceptional completion of) a join point can make use of its computational or exceptional result.

It is precisely these characteristics of AOP that we find so well-suited to reference counting. When *get* and *set* join points occur (corresponding to `getfield`/`getstatic` and `putfield`/`putstatic` bytecode instructions), we wish to update our reference counts. In AspectJ (an extension to Java providing AOP facilities) we can specify in a principled fashion exactly which events we wish to thusly advise. The pseudocode in this chapter is easily translated into AspectJ advice.

Once the appropriate classes have been augmented to implement the *Reference-Countable* interface, either in response to the automatic analysis described in Section 5.3 or as determined by hand, our advice uses the presence of that interface to introduce reference counting into the appropriate classes.

The heap-only reference-counting is much simpler than our full approach, which requires simulation of the run-time stack. For the heap-only approach, two pieces of advice

[5] We adopt here the terminology of AspectJ; however, the description provided is sufficiently general to encompass many aspect languages.

can instrument the code so that objects' reference counts are adjusted in response to pointer changes in the heap. On the other hand, accounting for stack activity requires knowledge of Java's method stack. *Around* advice is applied to the execution of all methods and constructors to simulate Java's frames with a `Stack` object. Advice is applied before method calls to push a `Stack` object, and advice after calls is used to find objects that are dead.

At present, writing "generic" pieces of advice in AspectJ requires the use of Java reflection, greatly increasing overhead. Runtime reflection is not actually necessary, however, and we augment the aspects with type analyses as discussed in Section 5.5 to remove all reliance on runtime reflection.

5.3 Reference-countable objects

In this section we present a straightforward algorithm for determining those objects in a complete Java program whose liveness can be determined solely by reference counting. Such objects are statically determined *never* to participate in reference cycles. Results reported in Section 5.6 were obtained by identification of reference-countable objects using the approach described here.

The algorithm we describe below is conservative, in that it may omit classes that could be reference countable but appear statically to be unsuitable. There is no harm in viewing *any* class as reference countable, except for the unnecessary overhead in maintaining reference-counting information for objects that cannot be collected using such information.

Our approach is to build a graph whose nodes represent instantiable classes and whose edges indicate potential references between classes. An edge is placed between classes x and y if an object of type x could reference an object of type y. Any type *not* involved in a cycle in such a graph can be determined using reference counting.

- A graph is constructed with a vertex for every class type.[6]
- For a field variable of declared type x, let *CouldBe(x)* denote the set of actual runtime types that could be referenced by the variable of type x. We describe the computation of this set below.

[6] As noted in Section 5.1, we assume all classes that will ever be loaded into the JVM are available for this analysis.

- For an actual class of type c, let *HasA(c)* represent the set of declared variable types in

  $$c, super(c), super(super(c)), \ldots, \text{Object}$$

  This set represents the (declared) types of objects that could be referenced from an instance of c.

We then perform the following computation:

foreach class c
  foreach type t in *HasA(c)*
    foreach type $u \in$ *CouldBe(t)*
      place an edge in the graph from node c to node u

Finally, the computation of *CouldBe(t)* is the *fix point* of the following:

- $t \in$ *CouldBe(t)*
- If class $c \in$ *CouldBe(t)* then so is every subclass of c.
- If interface $i \in$ *CouldBe(t)* then so is every class that implements i.
- If interface $i \in$ *CouldBe(t)* then so is every interface that extends i.

By repeating the above rules until nothing is added to *CouldBe(t)* we arrive at a fix-point answer.

5.4 Recycling objects

Once an object's death has been ascertained, systems vary as to what can be done with the object *post-mortem*. Some platforms offer an explicit mechanism to return a given object to the storage-management facility for subsequent reallocation; other platforms lack such a facility. In particular, the RTSJ standard has no such mechanism for reusing an object's storage. This is particularly unfortunate because RTSJ offers *scoped-storage* areas, in which objects can be allocated but never collected individually for arbitrary reuse.

In this work, we adopt the practice of *recycling* [26, 96] dead objects as follows.

```
recycle(object ):
    foreach object-field f in object
    | decrement(object.f.currentValue)
    | if count(object.f.currentValue) = 0
    | | recycle(object.f.currentValue)
    end foreach
    recycleList[object.class].push(object )
around new class-type constructor-args:
    if recycleList[class-type].isEmpty()
    | return proceed(class-type, constructor-args )
    fi
    return recycleList[class-type].pop()
```

Figure 5.5: Definitions for *recycle()* and a recycling-aware version of *new*.

- A dead object of type t is placed on a covert linked list of all dead objects of type t.[7]
- When an object of type t is subsequently allocated, (any) one of the dead objects on the covert list can satisfy the allocation request.

Manual modification of the application code to manipulate the covert lists and to modify constructor calls is a tedious and error-prone undertaking. We automate object recycling by providing the appropriate advice using AspectJ. Objects are recycled as shown in Figure 5.1 and Figure 5.2, and a suitable definition of the *recycle()* procedure is shown in Figure 5.5. Also shown in Figure 5.5 is a redefinition of Java's *new* operation: this instrumentation can be achieved in AspectJ by providing "around" advice on object construction, which takes the place of the computation that triggers it (in this case, instantiating a new object). The *proceed* facility in the figure computes the underlying join point (in this case, standard Java's standard *new* behavior). In our case the standard *new* behavior is invoked only when the needed type's recycle list is empty.

Note that the recursive call to *recycle* in Figure 5.5 can appear to take unbounded time, and may thus appear unsuitable for real-time systems. The code in question processes each field f of a dead object. If f references an object t, then t's reference count is decremented, possibly reaching zero and thereby deserving recycling. This activity can proceed at leisure and can be paced so as to occupy a bounded percentage of the CPU.

While RTSJ compelled us to recycle objects, recycling has other benefits:

[7]The covert list does not contribute to the liveness of objects on its list.

- All objects of a given type are the same size in Java, so the recycled object is just the size needed for any instantiation of that object's type.
- Objects typically include header information, most notably their type, which need not be initialized in a recycled object.

Because recycled objects must appear just as though they had been newly instantiated, recycled objects must be reinitialized properly. This can be problematic, because at the source level, Java constructors actually perform two tasks: allocation and initialization. While object recycling should circumvent the allocation of a new object, the initialization of the object is still desired.

Ideally, we would like to run the proper constructor on the recycled object. While we could achieve this directly in Java bytecode, the resulting program would not bytecode-verify, and AspectJ does not allow this action either. We thus require recyclable objects to implement *reset()* methods,[8] which we invoke in place of a constructor to reinitialize a recycled object.

5.4.1 Automatic Approaches

Our scheme uses a combination of aspect-weaving and bytecode manipulation to achieve a set of classes that perform user-level, reference-counting object recycling without such additional requirements on programming style.

Other options include dynamic aspects, which could transfer initialization code from all constructors into matching *reset* methods, and issue the *reset* call for the programmer.

Alternative approaches through the Java Native Interface (JNI), Jasmin, and reflection may allow less intrusive introduction of recycling into code. Through JNI, a call could be made to Java's *init* method, allowing us to separate the concerns of allocation and initialization from the constructor call. Similarly, with Jasmin, a recycled object could be pushed onto the stack as the target of the *init* call, instead of a newly allocated one, though this would not pass the bytecode verifier as pointed out earlier.

[8] This requirement can actually be relaxed due to code transformation, as discussed in Section 5.5.

With object recycling, creating a new object such as ListItem involves checking to see if old ListItem objects are available for reallocation. So the simple allocation of a ListItem object might now look like:

```
ListItem insert = psuedoAlloc().reset(contents);

public ListItem reset (Object contents) {
  this.contents = contents;
  return this;
}
```

/ Return previously collected ListItems if present, otherwise, return a new ListItem. */*

```
public ListItem psuedoAlloc () {
  ListItem ans = null;
  if (recycleList == null)
    ans = new ListItem();
  else {
    ans = recycleList;
    recycleList = recycleList.recycleNext;
  }
  return ans;
}
```

As our reference counting aspect currently has no other method of collecting storage, we will use the *ReferenceCountable* interface introduced above to apply our recycling aspect to.

Although the reinitialization of objects does require some additional work, the recycling aspect itself is relatively simple:

- To store the list of collected objects that have yet to be reallocated, we introduce a static field to each type.
- To maintain the list of collected objects without the introduction of additional carrier objects, we introduce a *next* field into each type.

- To check the list of collected objects introduced above, we supply "around" advice applying to all calls to a constructor of any *ReferenceCountable* type with any signature (Figure 5.5). This advice identifies an existing object of the correct type (if any) and reinitializes it by invoking its *reset* method. If no such object exists, the object is allocated and its constructor called in the standard fashion.

As discussed in Section 5.5, we automatically introduce a *reset()* method corresponding to each constructor, releasing the programmer from structuring their code according to any particular convention.

5.5 Implementation

We initially formulated a "generic" aspect that used Java reflection to insert reference-counting instructions and recycling. Reflection was necessary for the AspectJ advice to apply to any class that implements *ReferenceCountable*, for the following reasons:

- When a pointer p points away from x and points to y, the reference count at x must be decremented, and the reference count at y must be incremented. In AspectJ, the pointer change is captured by a *set* join point. While advice can be applied prior to the pointer change, AspectJ does not have a mechanism for retrieving the *old* value of the pointer.[9] Thus, reflection is required to open the `JoinPoint` and access the old (current) value of the pointer prior to the pointer change.

- When an object x's reference count reaches 0, x can be collected. Moreover, any *ReferenceCountable* object referenced by x needs to have *its* reference count decremented. Because the advice is applied to any *ReferenceCountable* object, the fields of that object must be elaborated within the advice. This is accomplished by reflection within advice that is intended to be applied to an arbitrary class.

- When an object x is detected as dead, it is recycled by appending it to a free-list of identically typed objects. Reflection allows access to the actual field value from within the generic advice.

- When an object is recycled and then reused, its *reset()* method must be invoked to reconstruct the object. Our AspectJ advice intercepts Java's *new* gesture, retrieves

[9] Previous releases of AspectJ did have this facility.

an object from the type's recycle list, and calls the *reset()* method with the same arguments that the user invoked the constructor with; this last step is performed reflectively.

This code is reproduced in Appendix A, on pages 174–181.

While the code resulting from application of the generic advice is correct, the performance is unacceptable—a few hundred times slower than Java's standard garbage collector on micro-benchmarks, and *many orders of magnitude slower* on larger benchmarks. Careful performance profiling revealed that almost all of the overhead was in the reflection code described above.

To remedy this, we essentially precompile or partially evaluate the aspects in terms of the types they affect. We still rely on AspectJ to handle the tedious and error-prone activity of identifying *when* appropriate action should be taken. The action itself is written directly into the class as follows:

- A *setter* method for field p can be written into a class so that when p is changed, the reference counts of the old and new target can be adjusted appropriately. Although such code is common across all classes, writing the code into the class simplifies calling the code through AspectJ.
- For any given object, the fields of that object are known statically and elaboration of those fields can be written into each class. Thus, in class `c`, there is no need to elaborate `c`'s fields reflectively when `c`'s count reaches 0. Instead, a method is written into the class to consider each field explicitly rather than reflectively; the reference count of any such field is decremented if the field currently points to a *Reference-Countable* object.
- A method can be written into each class to place dead objects on that class's free-list.

Further, *reset()* methods can be introduced into the class, one matching the signature of each constructor, and Java's *new* can be replaced with a corresponding call to the appropriate reset() method.

The process we currently use to generate functioning, reference-counting- and object-recycling-enabled Java bytecode from Java source is as follows.

```
pointcut reference(Object obj,
                   ReferenceCountable newVal) :
    target(obj) &&
    args(newVal) &&
    set((Object || (ReferenceCountable+)) *);

Object around(Object obj,
              ReferenceCountable newVal) :
    reference(obj, newVal) {
  ReferenceCountable oldVal;
  /* reflective code to get oldVal
   * (if it is ReferenceCountable) */
  Object retval = proceed(obj, newVal);
  Class c = oldVal.getClass();
  do {
    Field fields[] = c.getDeclaredFields();
    int length = fields.length;
    for (int i = 0; i < length; ++i) {
      /* reflective processing of value */
    }
  } while ((c = c.getSuperclass()) != null);
  return retval;
}
```

Figure 5.6: AspectJ advice to process object fields.

1. *We compile the classes normally.* Classes implement the marker interface *ReferenceCountable* or, optionally, an aspect is used to make certain classes implement the interface, resulting in no change the the original source code for such classes.

2. *We scan the resulting bytecode and generate AspectJ source which partially implements type-specific reference counting in each class.* This aspect introduces various fields and methods necessary for proper reference counting and object recycling.

3. *Using the AspectJ compiler, we compile the original classes together with the ReferenceCountable interface, the generated aspects from step 2, and a boilerplate reference counting implementation aspect.* This generates Java bytecode that mostly implements reference counting and object recycling, including all the necessary code to update reference counts and check for dead objects when Java's `putfield` and `putstatic` instructions execute.

4. *We inject the resulting bytecode with final reference counting and object recycling tweaks.* Using a program built on Clazzer [75], we clone and adjust each constructor's code to form a corresponding *reset* method, and we introduce a few type-specific helper methods, most notably *processDeadObject*, which is specially generated per type and handles inter-object references when an object is recycled.

The resulting classes use no reflection for their implementation of object recycling and reference counting.

5.6 Experimentation

We implemented the partially-evaluated aspectual approach for both heap- and stack-reference-counting schemes and we present the results of experiments conducted to measure efficiency.

As one benchmark, we used an RTSJ-suitable collections object `LinkedList` we developed, repeatedly creating lists of various sizes. This benchmark can be used with heap-only reference counting, and we used the heap-only implementation in gathering these results. In each case, reference-counting can detect the death of the list elements. We measured

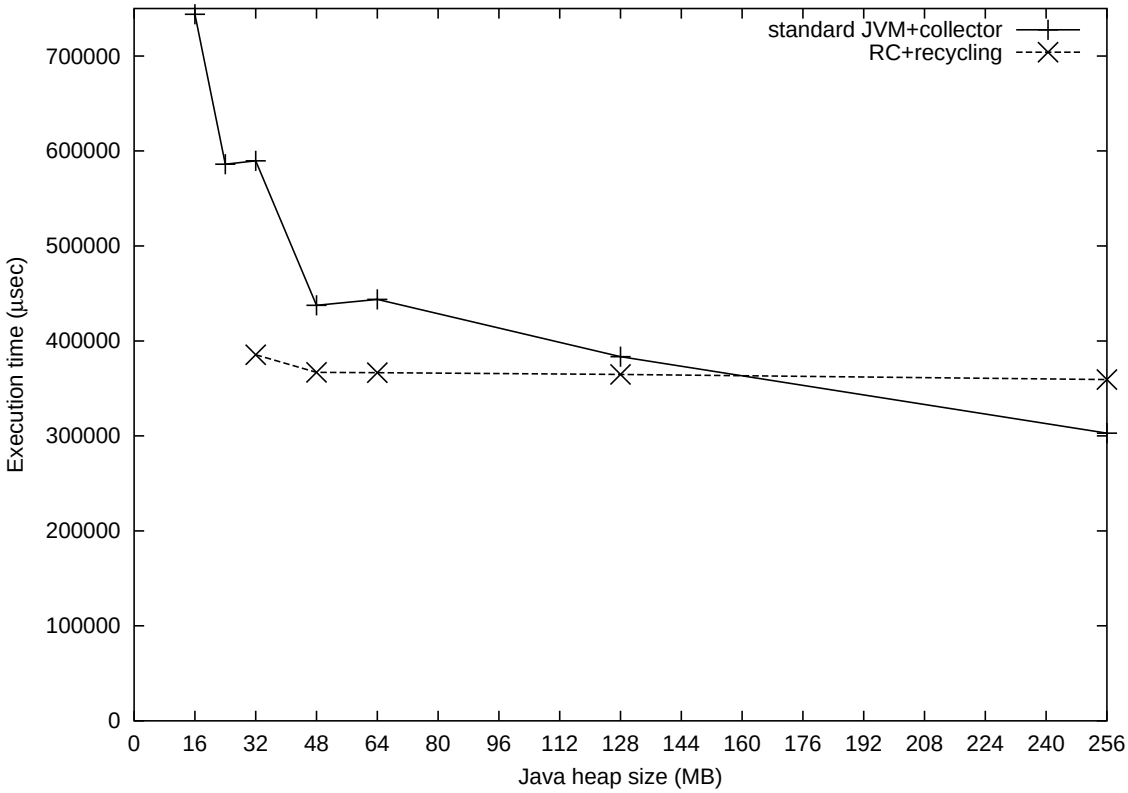


Figure 5.7: Average execution times for the Singly-Linked List benchmark generating lists of 10^6 carrier objects.

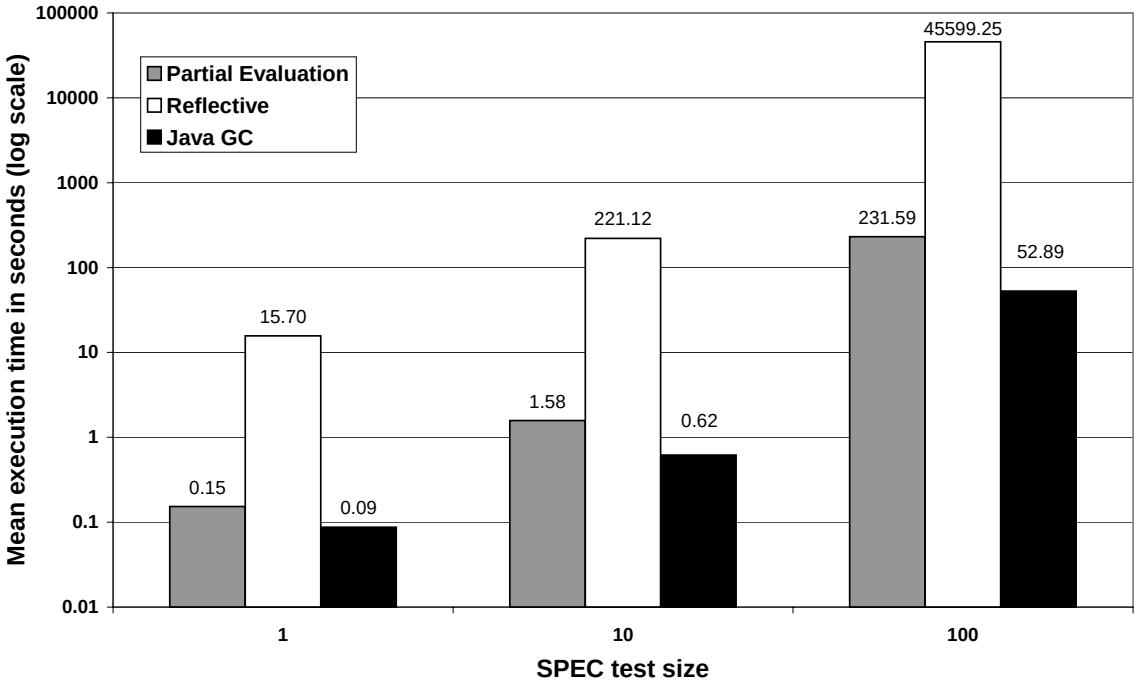


Figure 5.8: Results for `jess`.

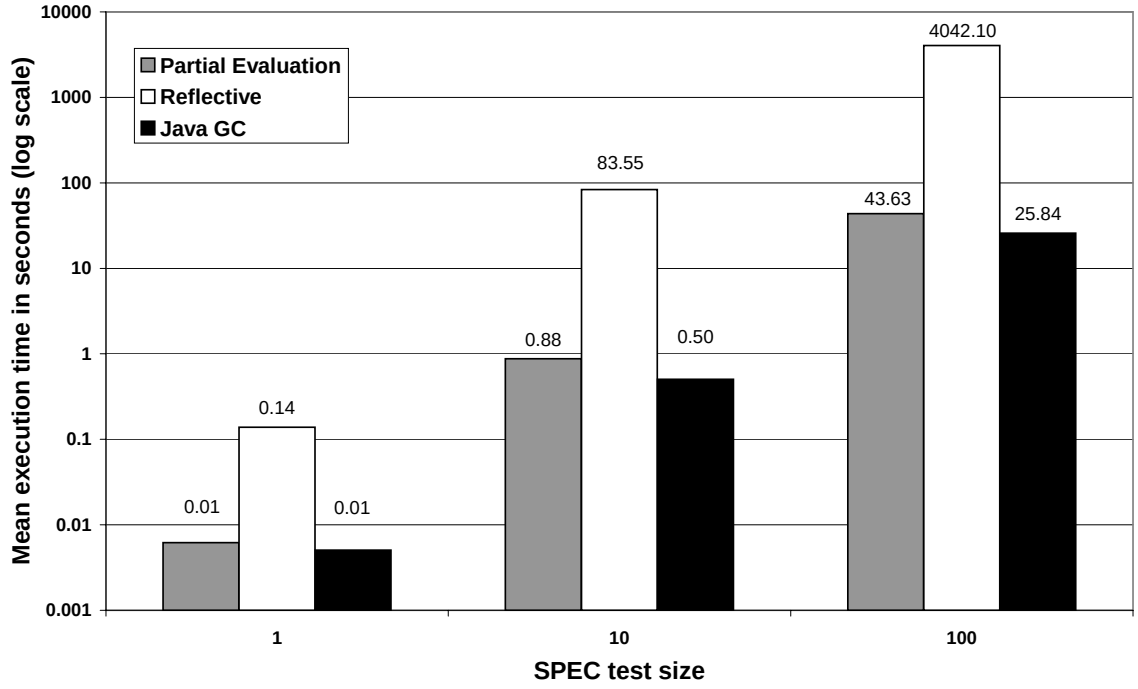


Figure 5.9: Results for `db`.

the time taken to run our benchmark with the traditional Java JVM[10] and its automatic garbage collector, and we compare this to times measured for our approach, which include the time to manage reference counts, to place dead objects on a linked list at deallocation time, and to unlink them for reallocation in response to *new* instructions. Figure 5.7 makes the comparison for different heap sizes for moderately large lists. In this case, the benchmark builds a list of 10^6 *null* references (so that there are 10^6 carrier objects), removes these references from the list (so that the carrier objects become dead storage), and repeats this whole process a total of ten times. The flat, consistent execution time over all heap sizes is characteristic of our approach. Note that we *must* still allocate 10^6 objects to build the first list, but for the nine subsequent lists, our approach need not allocate any additional objects—we can just recycle and reinitialize those objects already allocated, leading to very consistent timing for the remaining nine iterations, which do not make use of the standard memory manager at all. (Shown in Figure 5.7 is the average over all ten iterations.)

For smaller heap sizes, our approach performs better on this benchmark than the standard JVM approach with garbage collection, because smaller heaps force the standard JVM into a costly mark/sweep garbage collection phase more frequently. Besides performing consistently across different heap sizes, our timing is more consistent in recycling objects than the JVM is in allocating objects. Our approach, then, may be desirable for a system that requires high predictability in object allocation—as long as it properly provisions for the initial creation of the number of objects maximally live (as determined by reference counting). At the extreme end of its application range, then, our approach provides a statically-allocated object pool while encouraging the use of usual Java *new* gestures. For systems that do not permit standard garbage collection within special segments of the heap (such as RTSJ's *scoped memory* regions), our approach provides a programmer-level alternative that competes with, and performs more consistently than, a Java collector.

We next evaluate our approach on two of the SPECjvm98 [98] Java benchmarks.[11] Figure 5.8 and Figure 5.9 show the results we obtained on the `jess` and `db` benchmarks, respectively. Each benchmark was run on all three sizes (1, 10, 100), and the execution times for these two are shown using Sun's Java 1.4.1 in interpreted-only mode on a 2.4GHz

[10] Specifically, we used Sun's JVM 1.4.1 on a 2.4GHz Xeon running Linux 2.4.18 under the FIFO scheduling class and with paging disabled. We used the standard garbage collection strategy provided by the JVM (we didn't provide the `-Xincgc` option), and ran in *mixed* (JIT-capable) rather than *interpreted-only* mode.

[11] We required benchmark source code for the instrumentation, and thus could not apply our approach to the entire SPECjvm98 benchmark suite. Moreover, the `mpegaudio` and `compress` benchmarks are computational and don't reference many objects.

Pentium IV Linux box. The JVM was run in interpreter-only mode, and the heaps were sized as specified for the benchmarks.

Without the partial evaluation, our results are some orders of magnitude slower. The unacceptable performance is remediated by partial evaluation to obtain performance within a factor of 2–4 of the garbage-collection times.

While our approach is slower on these two SPEC benchmarks than Sun's garbage collector, we are targetting RTSJ scoped memory regions, where access to the garbage-collected heap is not permitted by the class of threads with stringent real-time guarantees. To provide garbage collection of objects within these regions and maintain RTSJ conformance, one must provide a user-level object recycling scheme like that described and evaluated here.

These results reflect heap-only reference counting; it is useful also to know the overhead of the stack approach. We implemented our stack-approximated reference-counting approach and performed some similar partial evaluation as for the heap-only implementation. The resulting code uses no reflection (which was the chief motivation to our partial evaluation), but neither is it as efficient as it could be; we are further refining and improving this implementation in ongoing work. Figure 5.10 shows the overhead of stack reference counting on a simple recursive micro-benchmark; this benchmark recursively calls a function N times that creates an object and establishes one pointer to it from the stack and one from the heap. Approximately N objects are created in N stack frames before being collected.

5.7 Chapter summary

In this chapter we have presented our aspectual approach for reference counting-based garbage collection for Java. Our work enables collection of objects that otherwise result in storage leaks for the Real-Time Specification for Java [21]. Without such an approach, developers are faced with the error-prone, time-consuming task of doing this work on their own.

Our approach makes a useful distinction and compromise between the error-prone parts of this undertaking, which are relegated to AspectJ, and the customization of classes to support reference counting without resorting to reflection within the aspects.

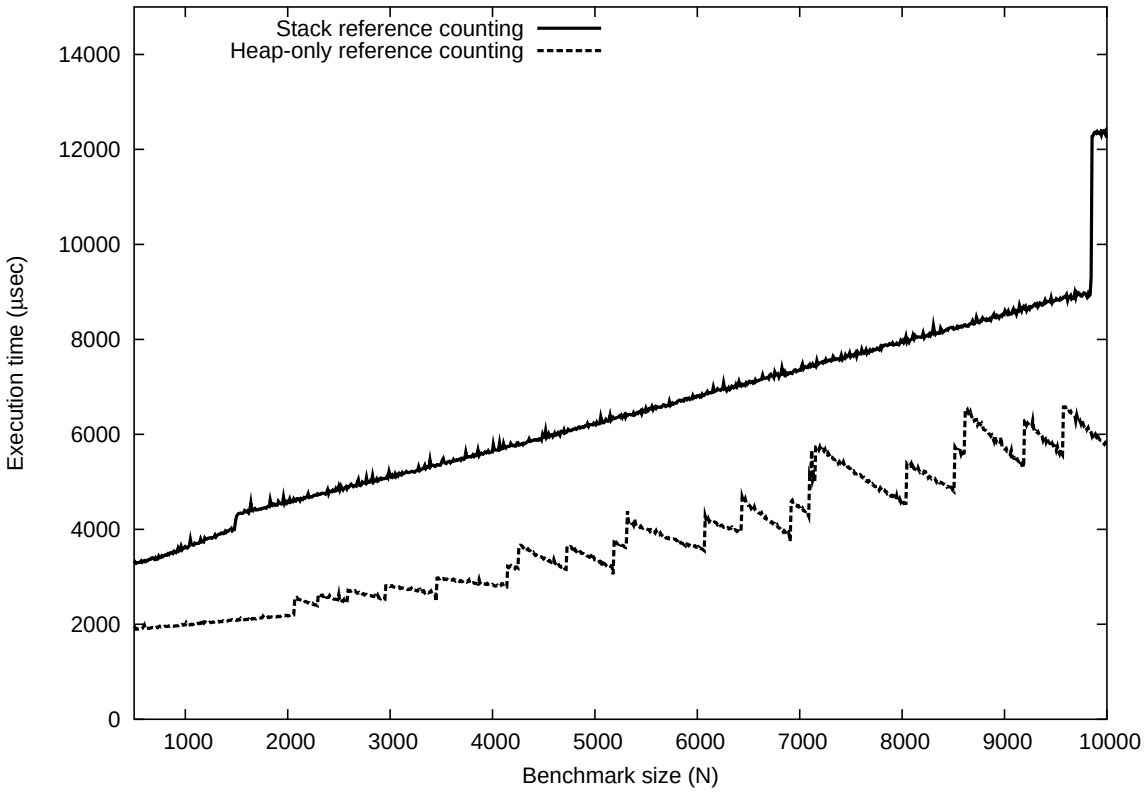


Figure 5.10: Stack *vs.* heap reference counting.

benchmark	**standard**	**RC+recycling**	$\frac{\textbf{standard}}{\textbf{RC+recycling}}$
compress	9.01	9.02	1.00
jess	4.04	4.03	1.00
raytrace	2.42	2.77	.87
db	23.13	23.87	.97

Figure 5.11: Comparison of the average execution times (sec) of four SPEC jvm98 benchmarks (size 100) for unmodified benchmarks and benchmarks with reference-counting, recycling-aware code injected.

In concept, it is possible to use aspectual reference-counting. However, current tools for performing this task (AspectJ) lead to unacceptable performance, so alternative approaches are required.

An aspect language that provided for compile-time, structurally-reflective decision-making (as, for example, advanced template metaprogramming in C++) could significantly reduce the complexity of our code generating type-specific aspects and bytecode, enabling programmers to more easily employ similar, cross-cutting memory management strategies in very general and reusable, yet very efficient, ways.

Alternatively, an AspectJ compiler able to partially-evaluate certain uses of reflection might be able to perform better in this task; however, the language would be less suited to the task, requiring the use a runtime API to perform a compile-time, metaprogrammatic task.

Aspect-oriented programming can provide benefits for expressing the separate concerns of compilers and runtime systems, just as it is for expressing application concerns. In this case, however, we were forced away from an aspect-oriented approach: either we could opt to use reflection, destroying performance, or create per-type advice, eliminating the leverage of aspect-oriented languages.

Our solution created per-type advice, but did so automatically, demonstrating that a tool combining aspect-oriented features and also type operations at compile time could provide the leverage of aspect tools without the runtime performance hit observed in this chapter.

Chapter 7 will continue the discussion of our observations of aspect-oriented languages and tools.

Chapter 6

Dual Heap Aspects

Garbage collection is a specialized process that causes significant memory bus traffic, pollutes memory caches, and can take an unpredictable amount of processor time. Many solutions to these problems have been proposed and implemented in recent work; one such solution is to off-load garbage collection work onto a separate hardware unit. Free from memory housekeeping chores, the general-purpose processor can devote itself to executing application logic.

This chapter examines a prototype that uses this approach, and it details the software support necessary to realize such a design. The chapter is self-contained and makes a contribution by studying the software support necessary for a particular hardware collector.

This software support is not written in an aspect-oriented programming language, but it could benefit from a suitable (yet hypothetical) aspect language. Ideas for this language are briefly discussed; they are further developed in Chapter 7.

Acknowledgments

The work of this chapter was sponsored by the Air Force Research Laboratory (AFRL) and the Boeing Company. The hardware work described was performed by Brett McNerney, Matthew Dallmeyer, and John Weber of the University of Dayton, and Justin Thiel of Washington University in St. Louis. In this chapter, this hardware design and implementation are referred to collectively as the University of Dayton hardware garbage collector, or simply the "hardware unit."

6.1 Introduction

Dynamic allocation of memory is a fact of life: most complex programs are not easily implemented by using stack- and statically-allocated memory alone. But dynamic allocation is a complicated mechanism. Memory being a limited resource, dynamic allocators must determine *which* slice of memory to allocate for a given request and *when* that slice should be reclaimed and reused.

In many popular imperative programming languages (ALGOL [12], Pascal [59], FORTRAN 90 [83], C [64], C++ [99], and others), software developers are provided a dynamic allocation mechanism, but they are also required explicitly to notify the allocator of heap storage they no longer require (commonly referred to as *freeing* storage). This is error-prone, and the problems are manifold: prematurely freeing storage can lead to type-unsafe data aliasing, double-freeing storage can confuse the allocator, and forgetting to free storage causes storage leaks. The symptoms are severe: each of these problems can ultimately cause program abortion, and in many instances they can be exploited by rogue code to gain control over the program. Often these problems are highly difficult to detect and debug.[1]

For complex projects, many developers now prefer to allow the runtime system to manage their heap.[2] A *garbage collector* performs that task and simplifies the programming model considerably; no longer is the developer required to understand the complex relationships between his program's data structures, and no longer is he required to write code sensitive to these relationships. Even when he understands these relationships, it can be a nontrivial and error-prone process to write code that covers every case where live storage becomes dead. Indeed, garbage collection is widely considered to increase programmer productivity significantly.

However, garbage collection comes at a price. Under memory pressure, the processor may be required to perform garbage collection bookkeeping *in place of* application logic

[1] Interestingly, designers of early functional programming languages (notably, Lisp [82]) and object-oriented languages (Simula [35] and Smalltalk [63]) showed an understanding of the need for automated memory management in their languages [37]. Perhaps this is because these programming paradigms (that of Lisp, especially) make it difficult for the programmer to judge precisely when to free storage.

[2] By "runtime system" we mean to be as general as possible: garbage collection can be implemented as part of an interpreter or virtual machine (as typically done in Java platform implementations), as an add-on library (as the Boehm collector [20]), or through compiler-injected code (as reference counting often is), or through a combination (as in compiled versions of functional languages). The *garbage collector* should, however, be distinguished from the *language* context in which it is being used—though the objects and object relationships that are (in)expressible in the host language certainly affect collector design.

to satisfy the application's storage needs. The processing time required can be highly unpredictable, though recent work aims to address this problem for real-time systems [13, 81]; see Section 6.8 for a discussion. Further, garbage collection often scans infrequently-accessed (or even dead) data structures, which pollutes memory caches.

These problems can be mitigated by using a separate hardware unit to perform garbage collection (or, in our case, all storage functions). Freed from these memory concerns, the general-purpose processor can continue executing application logic at full speed without interruptions, so long as its memory allocation requests do not exceed the available memory (or otherwise overwhelm the collector unit).

This chapter describes the software support we designed and implemented for one such hardware garbage collector. This particular hardware was developed by colleagues at the University of Dayton in 2006. It was not designed to act transparently to the runtime system: the unit manages only a *small* heap, not large enough to perform many computations of interest (in this case, aviation navigation systems). Thus a second, larger heap is used as backup storage. However, garbage collection is not supported in the secondary heap; it is mainly intended for use by permanent objects. The small, primary heap is sufficient to hold the temporary, short-lived storage for this application. The two parts of this *dual heap* use very different interfaces, requiring software support.

The rest of this chapter is organized as follows. Section 6.2 introduces copying garbage collection in general; Section 6.3 describes this particular hardware collector design; Section 6.4 lists what is required of software support for this device; Section 6.5 gives an aspectual description of this software support and indicates why it is not possible to implement with AspectJ. Section 6.6 describes our real-time Java platform and documents all of the modifications required to support the hardware collector unit; Section 6.7 discusses limitations of the current design and implementation of the hardware collector and software support; Section 6.8 discusses alternative approaches to solving this problem, including some background on garbage collection approaches; and Section 6.9 summarizes. A full, detailed changelog of the software support for this collector is provided in Appendix B.

6.2 Semispace copying garbage collection

Semispace copying garbage collection is discussed at length in textbooks and research surveys [62, 113]. This section outlines a basic incremental method similar to Baker's [15]. The discussion provided is quite brief, but it should be adequate for understanding later sections of this chapter. It may be skipped by readers familiar with the technique.

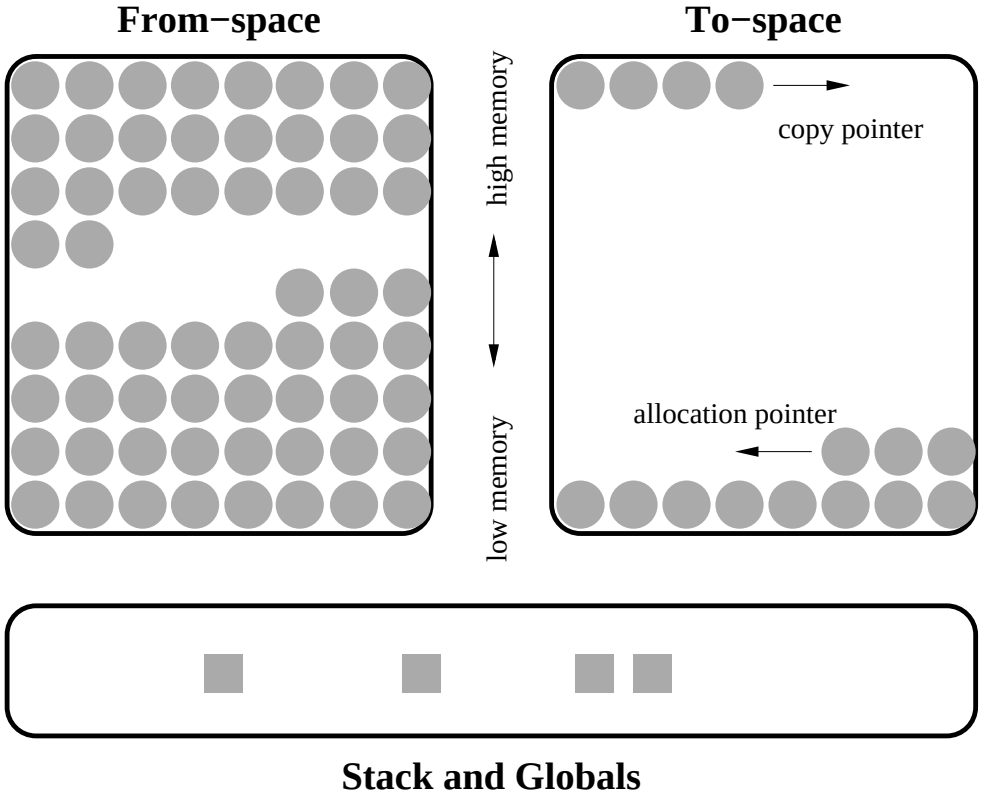


Figure 6.1: Semispace garbage collection in mid-cycle.

A *semispace copying garbage collector* maintains two *semispaces*, the *from-space* and the *to-space* (see Figure 6.1). Collection proceeds in cycles. At the start of a cycle, the from-space is populated and the to-space is empty. To initiate the cycle, a collector of this type copies all objects pointed to by the *roots*—typically the stack and global variables—into one end of the to-space. Each object's storage in from-space, now vacated, is left with a *forwarding pointer* to indicate the object's new location and the root pointers are updated with the new address.

The next phase is often performed concurrently with the mutator. The collector scans the objects it has copied (now in to-space). For every pointer into from-space, the referenced object is copied into to-space if it hasn't been already, a forwarding pointer is left, and the referring pointer is adjusted. This continues iteratively until all from-space objects in the closure of the points-to relation have been copied. Thus ends the cycle; dead objects are simply left uncopied in from-space and their storage is reused in the next cycle, as described below.

Concurrency with the mutator must be carefully managed. Naturally, the collector should only copy objects with the heap in a consistent state. A read barrier is typically employed to follow from-space forwarding pointers.[3] New memory allocations can be performed during this copying phase too; new allocations are made from the opposite end of to-space. Separate allocation and copying pointers are kept (as shown in Figure 6.1). If the two pointers meet before the copying process has completed, the collector typically gives up and declares itself out of memory, though fallback solutions are possible.

Finally, the two semispaces are atomically switched, the from-space becoming the new to-space and the to-space becoming the new from-space; this sets up the next cycle, which can begin immediately or when the garbage collector detects memory pressure exceeding a threshold.

6.3 Dayton's hardware garbage collector

This section is provided as background for the interested reader; it may be skipped without compromising his understanding of later sections. In particular, it does not describe any contributions by the author.

In 2006 the University of Dayton developed a semispace copying garbage collector in hardware, targetting a Xilinx ML403 board, a model intended for development. This board contains a Virtex 4 Field-Programmable Gate Array (FPGA) with a (mostly-standard) PowerPC 405 at its core. The design utilized memory available in the FPGA fabric rather

[3] Replicating garbage collection [85] is an alternative approach that requires only a write barrier: the barrier catches changes made to already-copied from-space objects and "replicates" the changes in to-space. This obviates the need for a double-indirection read-barrier. Another common approach is to install a forwarding pointer on *all* objects and follow it unconditionally; for to-space objects and uncopied from-space objects, the forwarding pointer points to the object itself.

Function	Arguments	Returns	Description
`New`	*type*	*object*	allocate an object
`NewArray`	*type, size*	*array*	allocate an array
`GetObjectData`	*object, offset*	*value*	read from a memory location
`PutObjectData`	*object, offset, value*	—	store into a memory location
`BumpRefCount`	*object*	—	note a new pointer to an object or array
`DecRefCount`	*object*	—	note a deleted pointer to an object or array
`CloneObject`	*object*	*object*	clone an object or array

Figure 6.2: The hardware garbage collector unit's interface.

than the (much larger) SDRAM chip on the board. This design decision eliminates the problem of data cache pollution, but limits also the utility of the collector, as it is in charge of so little memory.

To access this memory, the hardware unit exports a simple interface; Figure 6.2 shows this interface. As a set of functions, the interface is quite different from the standard way memory is read and written by PowerPC software. This means that when it references an object, the software must know where that object is—in SDRAM, or in the hardware collector unit in softcore—before it can perform the reference. We leave this challenge for Section 6.6. The hardware implementation of the interface is not reentrant; only a single call (in any *one* function) may safely be active at a time.

As a semispace garbage collector, this design has a from-space and a to-space; the two spaces together make up the available storage heap that the unit provides. The SDRAM heap is *physically and conceptually separate* from the hardware from- and to-spaces. Not only are pointers on the program stack and in global storage considered garbage collection roots, but also any pointers in the SDRAM heap; Section 6.6 describes the software support necessary to communicate this information to the collector unit.

With garbage collection offloaded to separate hardware, the general-purpose CPU can remain devoted to application logic. Conceptually, then, the collector can copy objects between its semispaces in the background at all times, starting another cycle immediately after one finishes. However, in the current hardware collector design, copying operations lock the entire block RAM heap. Therefore, to avoid starving the application of this resource, in practice it is better to trigger collection cycles when a certain threshold of used memory is exceeded.

6.4 Requirements for software support

In Java, all objects (and also arrays, which are just a specialized type of Java object) are allocated in the Java heap. The only storage that is stack- or globally-allocated are pointers to those objects and other items of scalar type (integral and floating point types of varying sizes). To a Java source program, the dual Java heap should appear as a unified heap—the runtime system should provide a transparent interface, manipulating the appropriate of the two memories when the program accesses an object. When the program allocates an object, the runtime should decide in which heap that object should be allocated.

The runtime system further should provide a mechanism for enabling and disabling allocation in the hardware memory management unit. If this mechanism is enabled *and* the class of the allocation request is suitable for allocation in the hardware unit, (see Section 6.7 for a discussion of unsuitable objects).

It is intended that a Java program start by allocating everything in the larger SDRAM heap (see Figure 6.3). After globals have been instantiated and initialized, the program can enable the hardware garbage collection unit for allocations (see Figure 6.4). At this point, objects are allocated in the unit, and the SDRAM heap typically becomes populated with references into the unit's memory. The unit's memory can also contain references into the SDRAM heap. At some point, garbage objects in the unit's memory are deallocated (by not being copied into to-space—see Figure 6.5).

The runtime system must inform the hardware unit when a reference into its heap (from the stack, global data space, or standard SDRAM heap) is constructed or destroyed. The hardware unit maintains an external reference count so that these externally-referenced objects are not collected until all such external references are destroyed.

The compiler must introduce read and write barriers for data accesses that *could* occur on a hardware unit-allocated object. In general, the compiler cannot statically determine which heap is indicated by a particular data access; this check must be performed at runtime, and the compiler must set up a runtime mechanism for determining the relevant heap and accessing it.

The hardware unit supports only 32-bit values in data slots; all data reads and writes are 32-bits. Java long integer and double-precision floating point types are 64 bits; these must be disassembled in software when written to the hardware unit's heap and reassembled

when read. (We restrict ourselves here to 32-bit architectures; pointers are always 32 bits and thus need not be disassembled or reassembled.)

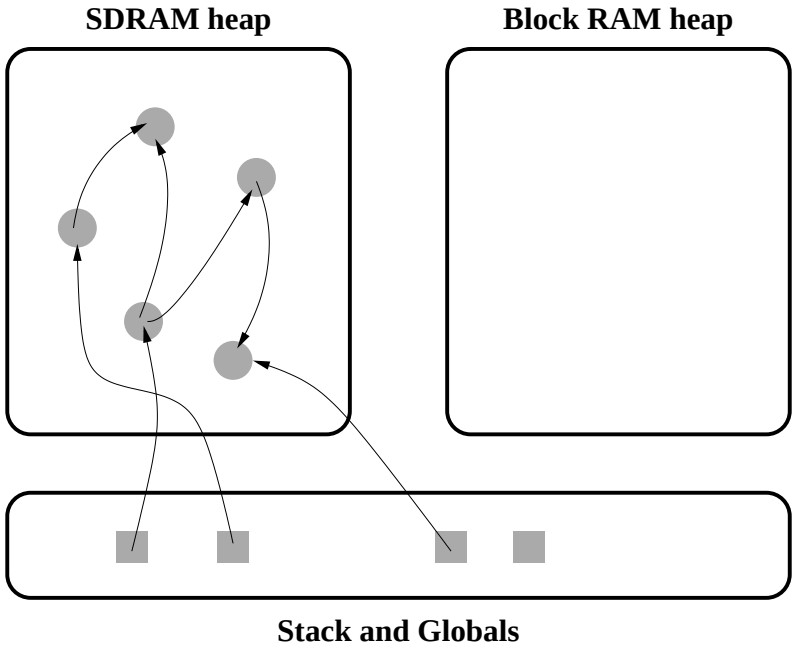


Figure 6.3: Application startup.

To summarize, software support must meet the following requirements:

1. The current allocation heap should be selectable via an application-visible interface.
2. Objects should be allocated in the current allocation heap. If the current heap is the hardware collector's heap and allocation fails or the type of object is not supported by the collector's allocator, then the allocation should occur in the standard heap.
3. Pointer bitmaps for classes must be generated for use by the hardware collector's allocator.
4. Read and write barriers must be implemented to perform runtime checks on object pointers to determine the heap to which they belong and implement the proper interface for accessing that heap.
5. For `long` and `double` types, data values must be disassembled when writing to the hardware unit's memory and reassembled when read.

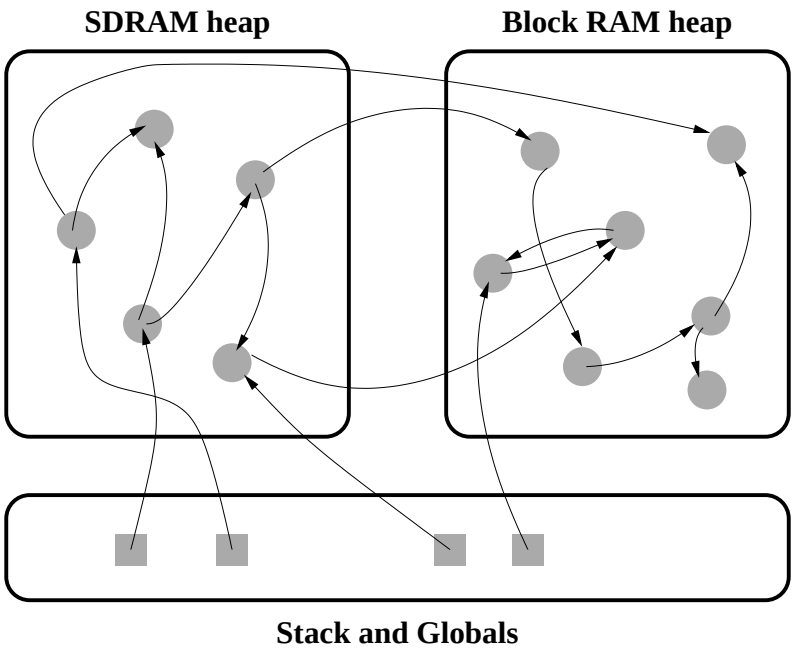


Figure 6.4: Application using garbage collector hardware unit.

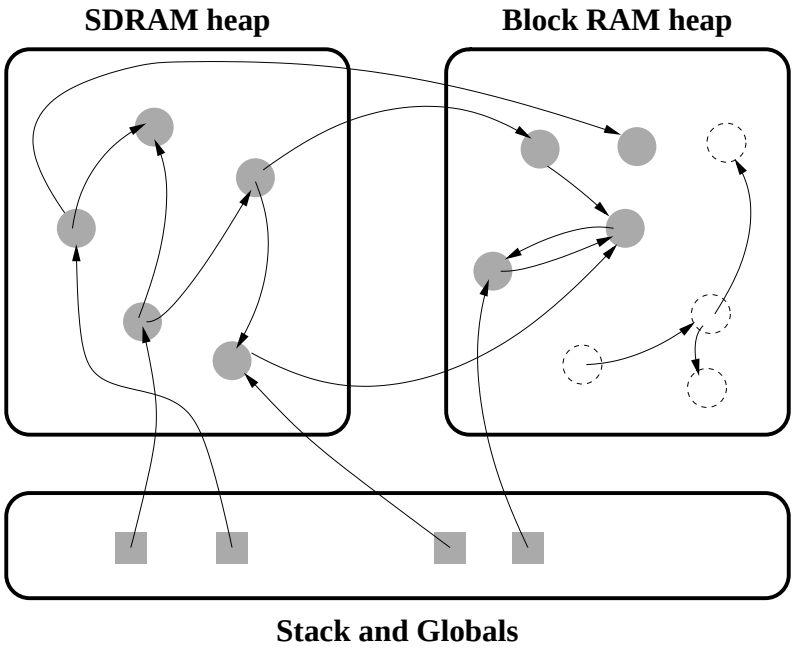


Figure 6.5: Application using garbage collector hardware unit; here, some objects are garbage.

Our specific implementation of these requirements is described in Section 6.4. The next section describes an aspectual description of this software support and indicates why we chose not to perform our instrumentation this way.

6.5 An aspectual description of the runtime system

An aspectual description of this software support is possible, though an aspectual implementation, given current tools, is impossible.

As mentioned in the previous section, read and write access to fields should test the object pointer, then access the field through the collector interface (using the function calls of Figure 6.2) or through the usual memory interface (using an addressing mode of the chip) as applicable.

This certainly has a high-level aspectual description.

1. On field or array accesses, determine the heap and access accordingly. For accesses of `long`- or `double`-type fields and arrays in the hardware unit heap, reassemble the value by accessing the two 32-bit slots for the field or array element.

2. On field or array assignments, determine the heap and assign accordingly. For assignments of `long` or `double` type in the hardware unit heap, disassemble the value and assign to two 32-bit slots.

3. When allocating an object of class or array type, allocate from the current allocation heap (either the standard SDRAM heap or the hardware unit's heap). For class types, determine the pointer mask and communicate it to the hardware unit's allocator. For array types, communicate to the hardware unit's allocator whether the array constituents are of pointer or non-pointer type.

4. When assigning a pointer value to the global data space, stack, or the SDRAM heap, and the pointer value is that of a hardware unit-allocated object, notify the hardware unit that an external reference to that object has been manufactured. For such assignments where the *overwritten* pointer value is that of a hardware unit-allocated object, notify the hardware unit that an external reference to that object has been destroyed.

This is relatively straightforward, and would be a complete implementation if an application-visible mechanism for selecting the current allocation heap were provided. However, employing AspectJ tools to perform the above tasks is currently impossible. Consider:

- Without relying on native code to perform the work, AspectJ programs cannot perform the necessary pointer manipulation.
- The instrumentation must affect *all* parts of the Java runtime environment, including system libraries and mechanisms internal to the implementation itself.
- The advice performing these tasks may be self-applicable. Self-application leads to interminable recursion and must be guarded against.

With native sections of code, a flexible runtime environment—Jikes RVM [58], for example[4]—and carefully-coded advice, these concerns can be discarded. Then the AspectJ language comes close to performing the task, *except*:

- Object and array layout, which differ across Java implementations, must be known by the advice.
- Method offsets into dispatch tables (vtables) must be known by the advice.

This destroys any possible portability of such advice. Even more importantly:

- Array operations must also be trapped and array pointers checked to determine in which heap they are allocated. Array accesses are not join points in AspectJ.
- The manufacture and destruction of local pointers (on the runtime stack) to FPGA-allocated objects must be accounted for the collector to be safe. Local variable assignment is not a join point in AspectJ.

For these reasons, the AspectJ language is currently not suitable for this task.

[4]Jikes RVM is entirely written in Java, using instances of "magic" classes to represent pointers and allowing standard operations on them from within Java. These magic objects have no runtime existence as on object—operations on them are transformed by Jikes RVM's just-in-time compiler into machine operations effecting the pointer operations.

6.6 Implementation of software support

jRate [34, 33] is a real-time Java platform aiming for RTSJ compliance, though it has proven useful as a research platform as well. It is based on the GNU Compiler for Java (GCJ) and the Java standard library that is part of the GNU Compiler Collection (GCC) [48].

As part of this work, jRate has been extended to support the Dayton hardware collector described in Section 6.3.

Recall the interface for the hardware unit shown in Figure 6.2. Each interface function is a C stub that locks (in software) the interface against multithreaded access, then communicates with the hardware unit. These "calls" into hardware are performed entirely synchronously, as a software procedure call would be: the interface functions do not return until the hardware has completed its task, regardless of whether they return a value. Together with the software lock, this ensures thread-safe access to the unit's internal data structures.

Because the interface to the hardware unit's memory is different than to the usual store, all the code emitted by the compiler must check—at runtime—which heap the dereferenced pointer resides in. This is a range check: addresses p satisfying $4096 \leq p < 5120$ are allocated in the hardware unit. But performing this check is suboptimal: performing this range check ultimately generates two conditional jumps. This can be reduced to one by performing an optimized, equivalent check on p. In C notation, the check we perform is:

$$p \texttt{ \& 0xfffffc00 == 0x00001000}$$

This generally performs better, and it reduces the size of the read barrier.

We treat Java and C++ separately in the implementation (as GCC provides two separate front-ends for these languages). We'll address them separately here also as the challenges of each implementation are somewhat different.

6.6.1 Java support

In the Java front-end, we force compilation from bytecode (and don't support compilation from source when supporting the hardware collector). JVMs are stack machines; accordingly, Java bytecode assumes an operand stack. There are also "local variables" (registers)

accessed, though they generally serve as secondary storage; aside from loads and stores, few bytecode instructions operate on these registers.

Recall that Java globals (that is, `static` fields), the Java stack and registers, and the standard SDRAM Java heap can all hold external references into the hardware unit's memory. All such references must be recorded by the hardware unit, and it is the job of the software support to communicate the presence of these external references to the hardware unit.

The following sections detail our support for catching all references to hardware unit storage from the Java stack and registers, global data space, and SDRAM heap, and our support for communicating the presence of these to the hardware unit.

Noting external references

The direct way to instrument stack-based external references would be to call `BumpRefCount` on push and `DecRefCount` on pop. However, we don't instrument stack accesses in this manner. Rather, on pop, we leave the value there and call `DecRefCount` it when it's overwritten by a subsequent push.

GCC normally splits the Java stack on access mode (floating point values get a stack, integral values get another) and allocates pseudoregisters lazily to (*stack-index*,*access-mode*) pairs. We leave this access mode requirement in place, but we add also pointer and non-pointer pseudoregister stacks. If a bytecode sequence pushes a pointer, pops it, then pushes an integer, our compiler won't emit a `DecRefCount` for the pointer since the values are in different pseudoregisters (and thus the pointer wasn't overwritten). This makes the front-end support easier to implement, as it is a simple matter to determine what is and what is not a pointer—a pseudoregister holding a pointer will never hold a non-pointer, and *vice versa*. This simplification of the implementation does not harm hardware register allocation; nonconflicting pseudoregisters can map to the same hardware register.

We emit a `BumpRefCount` at the following sites.

1. a pointer is passed as a method argument (at callee side); for non-static Java methods, this includes the *this* pointer, which is passed as an implicit method argument
2. a value is assigned to a pointer array

3. a value is assigned to a pointer field
4. a value is assigned to a pointer static field
5. a pointer is pushed onto the Java stack
6. a pointer is generated on the Java stack during a Java bytecode instruction in the `dup` family
7. a pointer is stored into a Java register

In all cases, the call to `BumpRefCount` is conditional; if the relevant pointer value is not to a hardware unit-allocated object, the call is not made.

We emit a `DecRefCount` at the following sites:

1. a pointer is overwritten on the Java stack
2. a pointer is overwritten on the Java stack during a Java bytecode instruction in the `dup` family
3. a pointer is overwritten in a Java local variable
4. a pointer is overwritten in an array
5. a pointer is overwritten in a field
6. a pointer is overwritten in a static field
7. a pointer is in a local variable (or function argument variable) at function exit, whether normal or exceptional; this includes the implicit *this* pointer
8. a pointer is in a Java stack slot (above or below the current stack pointer) at function exit

As for `BumpRefCount`, these calls are conditional; if the destroyed pointer is not to a hardware unit-allocated object, the call is not made.

Dealing with returned and thrown pointer values

Given the above, the most straightforward way to handle method return values of pointer type, as well as thrown values on exceptional exit (which are always of pointer type in

Java), would be to allow the pointer to be the subject of a `DecRefCount` on method exit, and do a corresponding `BumpRefCount` in the caller's context, as the pointer is now on the caller's operand stack. However, this leads to a race condition. The `DecRefCount` in the callee context could drop the object's external reference count to zero, making the object eligible for collection by the hardware collector if no hardware unit-allocated object refers to it. But a pointer to this object still exists, and it should not be collected.

Thus, returned and thrown pointers are handled as a special case. At the site of the `areturn` or `athrow` bytecode instruction, which return and throw a pointer, respectively, the top Java stack slot (which contains the pointer) is nullified. This keeps the object from getting a `DecRefCount` that could drop its external reference count in the hardware unit to drop to zero. On the caller side, function return values (of pointer type) and caught exceptions are, as a special case, *not* subject to a `BumpRefCount` when pushed onto the stack. Essentially, the reference count in use by the callee is transferred to the caller.

6.6.2 C++ support

On the C++ side, we only need to trap pointers to Java objects—C++ objects are never eligible for allocation within the hardware unit memory. We feel this is acceptable because the allocations in the Java programs we target are for Java objects, not C++ objects; the only allocations for non-Java objects are in native parts of the standard Java library implementation, mostly during one-time initialization of the runtime system or of loaded classes. Further, C++ object pointers are often directly manipulated within C++ programs via pointer arithmetic, and such pointer arithmetic is never valid on pointers into the hardware unit.

However, we don't make any attempt to close C++ typing loopholes that could be used to hide Java pointers. For instance, if a pointer to a hardware unit-allocated Java object is cast to an `int` and stored in an `int`-typed field, we won't catch the assignment. It is up to C++ code running as part of a Java application to manually increment and decrement the external reference count to objects; as mentioned above, we can control this body of code directly since it is part of the standard Java library.

We emit a `BumpRefCount` at the following sites in the C++ front end.

1. a Java-typed pointer is passed as a function argument (bump at callee side); for non-static Java methods implemented "natively" in C++, this includes the *this* pointer[5]
2. a pointer is assigned to a Java-typed pointer field
3. a pointer is assigned to a Java-typed global or local variable
4. a pointer is assigned to a Java array

We emit a `DecRefCount` at the following sites in the C++ front end:

1. a Java-typed pointer is overwritten in a field
2. a Java-typed pointer is overwritten in a global or local variable
3. a pointer is overwritten in a Java array
4. a pointer is in a Java-typed pointer argument at function exit (whether normal or exceptional)
5. a pointer is in a Java-typed local variable when that variable's scope exits (whether normal or exceptional)

A *pragma* is supported in the extended C++ compiler to turn on and off this C++ instrumentation. This is particularly useful for special cases where the external reference count for hardware unit-allocated objects should be handled explicitly by the code—for example, in the memory management runtime code itself, or in code that uses Java's `System.arraycopy()` library function to manipulate Java pointer arrays.

6.6.3 Read and write barriers

The emissions of `GetObjectData` and `PutObjectData` emissions are straightforward and are used both for fields and array elements. We include pseudocode here to make the read and write barriers of our compiler explicit. These are the Java versions, the C++ versions are identical except for arrays (discussed further below).

[5]Under the Compiled Native Interface (CNI) [], which we used exclusively for this project, the *this* pointer is passed as an implicit method argument to a C++ method implementing a Java class "native" method. This is not the case under JNI [], which passes the *this* pointer as an explicitly-named function argument.

The read barrier is split into two cases, *word* and *doubleword*. The write barrier is split into three cases—*word primitive*, *doubleword primitive*, and *pointer*. Pseudocode for non-static field read and write barriers is in Figure 6.6.

```
to GET double/long fields:
    if object handled by SMM unit
        ( SMM_getObjectData(object, field_hi) << 32 |
          SMM_getObjectData(object, field_lo) )
    else regular_getfield(object.field)

to GET non-double/long fields:
    if object handled by SMM unit
        SMM_getObjectData(object, field)
    else regular_getfield(object.field)

to PUT pointer fields:
    if object handled by SMM unit
        SMM_putObjectData(object, field, new_value)
    else
        // x refers to y from z
        old_value = regular_getfield(object.field)
        regular_assignment(object.field <-- new_value)
        if new_value is handled by SMM unit
            SMM_bumpRefCount(new_value)
        if old_value is handled by SMM unit
            SMM_decrRefCount(old_value)

to PUT double/long fields:
    if object handled by SMM unit
        // new_value_hi/lo are the hi/lo words of doubleword
        SMM_putObjectData(object, field_hi, (new_value >> 32) & 0xffff)
        SMM_putObjectData(object, field_lo, new_value & 0xffff)
    else regular_assignment(object.field <-- new_value)

to PUT non-pointer, non-double/long fields:
    if object handled by SMM unit
        SMM_putObjectData(object, field, new_value)
    else regular_assignment(object.field <-- new_value)
```

Figure 6.6: Pseudocode for (non-static) object field read and write barriers.

Arrays are similar; pseudocode is listed in Figure 6.7. Note there are *four* words of overhead in an array: the vtable pointer, the sync_info pointer (for use as Java monitor), the RTSJ memory area pointer, and the array length.

```
to GET double/long array elements:
    if array handled by SMM unit
        ( SMM_getObjectData(array, (index * 2) + 4) << 32 |
          SMM_getObjectData(array, (index * 2) + 5) )
    else regular_arrayload(array[index])

to GET non-double/long array elements:
    if array object handled by SMM unit
        SMM_getObjectData(array, index + 4)
    else regular_arrayload(array[index])

to PUT array elements of pointer type:
    if array handled by SMM unit
        SMM_putObjectData(array, index + 4, new_value)
    else
        // x refers to y from z
        old_value = regular_arrayload(array[index])
        regular_assignment(array[index] <-- new_value)
        if new_value is handled by SMM unit
            SMM_bumpRefCount(new_value)
        if old_value is handled by SMM unit
            SMM_decrRefCount(old_value)

to PUT array elements of double/long type:
    if array handled by SMM unit
        SMM_putObjectData(array, (index * 2) + 4, (new_value >> 32) & 0xffff)
        SMM_putObjectData(array, (index * 2) + 5, new_value & 0xffff)
    else regular_assignment(array[index] <-- new_value)

to PUT array elements NOT of double/long or pointer type:
    if array handled by SMM unit
        SMM_putObjectData(array, index + 4, new_value)
    else regular_assignment(array[index] <-- new_value)
```

Figure 6.7: Pseudocode for array element read and write barriers.

Static fields (globals) and local variables are much easier to handle since they never live in the hardware unit themselves. In particular, there need be no local or global variable read barrier. Figure 6.8 shows the pseudocode.

```
GET on statics/locals is normal and not instrumented.

to PUT a static/local of pointer type:
    // x refers to y from z
    old_value = regular_get(variable)
    regular_assignment(variable <-- new_value)
    if new_value is handled by SMM unit
        SMM_bumpRefCount(new_value)
    if old_value is handled by SMM unit
        SMM_decrRefCount(old_value)
```

Figure 6.8: Pseudocode for global (static field) and local variable write barrier. No read barrier is needed.

Two other mechanisms must be implemented as well—calling a method through a vtable and getting the length of an array; see Figure 6.9 for pseudocode.

```
to CALL A METHOD on an object:
    if object handled by SMM unit
        vtable = SMM_getObjectData(object, 0)
    else vtable = object.vtable
    invoke_through_vtable(vtable, method)

to GET THE LENGTH of an array (including all those pesky bounds checks):
    if array handled by SMM unit
        SMM_getObjectData(array, 3)
    else regular array.length access
```

Figure 6.9: Pseudocode for calling a method (through the vtable) and getting the length of an array.

Note that this software support will exhibit a data race if the application being instrumented has a data race: a thread could get a pointer to a hardware unit-allocated Java object from an SDRAM-allocated pointer field, but before it has a chance to bump the reference count, another thread could destroy the pointer and decrement the reference count to zero. The object could be collected out from under the first thread.

C++ instrumentation is identical for fields and global and local variables, though arrays are treated very differently. CNI requires access to Java array elements from C++ by calling an `elements()` template function. We modified the standard library (instead of the

compiler) to implement hardware unit-allocated array external reference count adjustments when arrays are accessed through this template function. Array syntax is largely unaffected and only required a few manual changes (flagged by compile errors) in parts of the library code written in an unorthodox manner.

6.6.4 Implementation of compiler support

As indicated in previous sections, to perform its tasks the runtime system depends on compiler support. Beyond the usual support needed for an ahead-of-time-compiled Java platform, two main pieces of compiler support are needed. We described the first above: the replacement of normal memory reads and writes with more complicated read and write barriers that discriminate between SDRAM- and hardware unit-allocated objects.

The second main piece of compiler support needed is the building and exporting of pointer masks for each class of object. GCJ already performs a similar task for its support of the Boehm garbage collector [20]; we used this implementation as a base and tailored it for use with the Dayton hardware garbage collector.

6.6.5 Overhead of barrier instrumentation

There is considerable overhead in the read- and write-barrier implementations. The University of Dayton measured the speed and real-time suitability of their collector; here, we are concerned with the on-disk *footprint* of the instrumented binaries. This is an important concern, as they must fit on a compact flash card together with the FPGA bitfile and Linux kernel, and they must be loadable into the limited physical memory of the system; virtual memory isn't used in this system since there isn't a suitable disk to which to swap pages.

Figure 6.10 lists sizes for instrumented and non-instrumented binaries, both stripped of debugging symbols (with the *strip* utility) and unstripped, for three optimization levels. Both jRate and application binaries are shown; the application is real-time Java flight control software provided by the Boeing Company. jRate libraries are always built with the `-O2` optimization flag, so their sizes for the other optimization levels is omitted; however, the size of the `-O2` jRate binaries *are* included in the "Total upload" line, as these libraries must be uploaded to the board and loaded into memory in any case. Note that the smallest instrumented upload package in this table is the stripped `-O2` configuration at 39.4 MB;

to maximize space efficiency, this is the binary package we would choose to upload to the board and execute.

Figure 6.11 compares binary sizes at the three different optimization levels. Even though GCC's `-Os` flag is meant to optimize for size, it occasionally increases the size of the resulting binary.

Figure 6.12 shows the inflation of instrumented binaries over their non-instrumented counterparts. For the smallest instrumented upload package (the stripped `-O2` configuration), there is a factor of 3.55× inflation in the size of the package due to the instrumentation.

6.7 Limitations

Some object types are never allocable in the hardware collector unit:

- Only 1024 non-array object types (for the collector, types are just tuples (*size*, *ptrmask*)) can be registered with the unit; if the application uses more types, some will not be allocable in the unit.
- Objects and arrays larger than the unit's semispace size are unallocable.
- Object types larger than 64 words that have pointers beyond the 64th slot are unallocable in the unit, since pointer masks are limited to 64 bits.
- Object types that require special memory layout and access or byte-addressability (*e.g.,* `java.lang.String`) are unallocable in the hardware unit.
- Object types used internally by the JVM (in particular, mutex types, class loaders, classes, etc.) are unallocable in the unit.

These objects and arrays are allocated in the SDRAM heap (or, failing that, an out of memory exception is raised). Further, any other requests of the unit that fail (*e.g.,* for lack of space in the unit) are allocated from the SDRAM heap.

Size in bytes	**With instrumentation**					
Optimization level	O0		O2		Os	
Postprocessing	–	strip	–	strip	–	strip
`libgcc_s.so.1`	–	–	829972	32864	–	–
`libstdc++.so.5`	–	–	4275792	743696	–	–
`libjRateCore.so.0`	–	–	1010849	115800	–	–
`libgcj.so.4`	–	–	34570837	18086452	–	–
`libflightconcni.so`	13690	9304	11760	8056	11235	7544
`libmessage_PAYLOAD_x86.so`	1113108	51720	1145138	44608	1107495	41760
`libttimestampcni.so`	19697	13584	18440	12468	17976	12000
`prismj`	54750043	33433432	43749378	22225624	58364820	27309976
Total upload	**96583988**	**52486852**	**85612166**	**41269568**	**100188976**	**46350092**

Size in bytes	**Without instrumentation**					
Optimization level	O0		O2		Os	
Postprocessing	–	strip	–	strip	–	strip
`libgcc_s.so.1`	–	–	830276	32864	–	–
`libstdc++.so.5`	4257318	747632	4257318	747632	4257318	747632
`libjRateCore.so.0`	–	–	997509	111256	–	–
`libgcj.so.4`	–	–	22951308	5762696	–	–
`libflightconcni.so`	7871	4816	7408	4436	7251	4296
`libmessage_PAYLOAD_x86.so`	1103452	47912	1137617	43432	1100090	40484
`libttimestampcni.so`	18153	12420	17103	11456	16679	11028
`prismj`	29131975	7786524	26320058	4913820	28803920	5933308
Total upload	**59297862**	**14506120**	**56518597**	**11627592**	**58964351**	**12643564**

Figure 6.10: Size (in bytes) of x86 `prismj` application binary and dependent libraries with and without instrumentation (see Section 6.6.5). Sizes for three optimization levels are listed, and for each level the unprocessed size is given as well as the size after postprocessing with *strip* to remove debugging symbols. jRate libraries are always compiled with `-O2` and aren't shown in the other columns; however, their size *is* included in the total in the bottom row.

Optimization savings/loss	**With instrumentation**					
	$\frac{\text{Os}}{\text{O0}}$		$\frac{\text{O2}}{\text{O0}}$		$\frac{\text{Os}}{\text{O2}}$	
	–	strip	–	strip	–	strip
`libflightconcni.so`	0.82	0.81	0.86	0.87	0.96	0.94
`libmessage_PAYLOAD_x86.so`	0.99	0.81	1.03	0.86	0.97	0.94
`libttimestampcni.so`	0.91	0.88	0.94	0.92	0.97	0.96
`prismj`	1.07	0.82	0.8	0.66	1.33	1.23
Total upload	**1.04**	**0.88**	**0.89**	**0.79**	**1.17**	**1.12**

Optimization savings/loss	**Without instrumentation**					
	$\frac{\text{Os}}{\text{O0}}$		$\frac{\text{O2}}{\text{O0}}$		$\frac{\text{Os}}{\text{O2}}$	
	–	strip	–	strip	–	strip
`libflightconcni.so`	0.92	0.89	0.94	0.92	0.98	0.97
`libmessage_PAYLOAD_x86.so`	1	0.84	1.03	0.91	0.97	0.93
`libttimestampcni.so`	0.92	0.89	0.94	0.92	0.98	0.96
`prismj`	0.99	0.76	0.9	0.63	1.09	1.21
Total upload	**0.99**	**0.87**	**0.95**	**0.8**	**1.04**	**1.09**

Figure 6.11: Comparison of sizes of `prismj` application binary and dependent libraries under different GCC optimization levels (see Section 6.6.5). Note that the `-Os` option to GCC, which optimizes for size, sometimes increases the size of the binary. jRate libraries aren't shown, as they are always compiled with `-O2`. The totals are ratios of the totals in Figure 6.10, and thus include the jRate libraries.

Instrumentation inflation (ratio $\frac{\text{instrumented}}{\text{uninstrumented}}$)	O0		O2		Os	
	–	strip	–	strip	–	strip
`libgcc_s.so.1`	1	1	1	1	1	1
`libstdc++.so.5`	1	0.99	1	0.99	1	0.99
`libjRateCore.so.0`	1.01	1.04	1.01	1.04	1.01	1.04
`libgcj.so.4`	1.51	3.14	1.51	3.14	1.51	3.14
`libflightconcni.so`	1.74	1.93	1.59	1.82	1.55	1.76
`libmessage_PAYLOAD_x86.so`	1.01	1.08	1.01	1.03	1.01	1.03
`libttimestampcni.so`	1.09	1.09	1.08	1.09	1.08	1.09
`prismj`	1.88	4.29	1.66	4.52	2.03	4.6
Total upload	**1.63**	**3.62**	**1.51**	**3.55**	**1.7**	**3.67**

Figure 6.12: Comparison of sizes of `prismj` application binary and dependent libraries with and without instrumentation for hardware garbage collection support (see Section 6.6.5). Each number is the ratio of instrumented file size *versus* uninstrumented file size.

6.8 Alternatives

There is much related work in this area, and several alternatives exist to using the hardware collector developed by the University of Dayton.

The simplest alternative is to use an off-the-shelf, purely software collector, such as those that come with a Java virtual machine [101], or the Boehm collector [20] that comes as part of GCJ. However, this suffers from precisely the drawbacks that the hardware collector is intended to avoid—that the general-purpose processor must spend time performing collection work instead of executing application logic.

In real-time systems development circles, garbage collection has generally been ignored. Many real-time systems are built relying on explicit rather than automated deallocation; this approach has the liabilities discussed in Section 6.1. Other approaches include using memory regions [105] or the stricter RTSJ version of regions, *scoped memory* [21, 19, 33, 38, 36]. Many region systems are semi-automatic, where the work of deallocation is not fully automatic but shared between the user and the system. Others, like RTSJ's, perform consistency checks at runtime to verify that no dangling references can be manufactured; this leads to a different memory model where certain objects may not reference other objects [21, 36].

Many garbage collectors that are "real-time" in some sense have been proposed over the years [69, 15, 73, 25, 4, 86, 16, 60, 85, 30, 13, 97, 3, 37]. Recent work has demonstrated real-time collectors operating in realistic, modern applications with real-time constraints, and fall broadly into two categories. A *paced* collector, such as Metronome [13], attempts to hold the application logic's utilization of machine resources constant; the effect is as if the application is running on a slower processor without the overhead of garbage collection. *Work-rate* collectors, like the one in Aonix PERC [3], do a small amount of collection work at each allocation site to spread the cost of garbage collection throughout the program's execution. In both cases the collectors are tunable for the application so they can keep up with allocation requests. The as-yet-unreleased RTS 2.0 from Sun Microsystems is rumored to include a real-time garbage collector.

Previous approaches to hardware-assisted collectors [94, 87] have employed specialized hardware to perform the task. The University of Dayton approach uses an FPGA for memory management; this allows faster prototyping and more complete application-specific collector configuration.

6.9 Chapter summary

We have described a hardware garbage collector designed and implemented by the University of Dayton in 2006 and the compiler and runtime support we implemented to support that collector.

This software support is conceptually aspect-oriented (Section 6.5), but the AspectJ language and compiler don't support the features needed to implement the software support in this way. We resorted to modifying jRate [34], our real-time Java platform built atop the industrial-strength compiler GCC, to support this work (Section 6.6).

Chapter 7

Observations about System Aspects

Previous chapters have considered a number of system aspects. Each provides its own complete and separate contribution, including implementation and analysis. In each case, we ultimately performed our implementations outside of today's aspect-oriented languages and tools, despite the fact that the designs are aspect-like.

We now tie these system aspects together by making observations about them and performing an analysis about what this means for future aspect languages. To recap:

- Chapter 4 describes a C++ metaprogram to test the feasibility of a rate-monotonic schedule, rejecting programs that are not feasible at compile time or (at user discretion) dropping less-important tasks to ensure feasibility;
- Chapter 5 implements a reference-counting scheme that is able to reuse heap storage in a Java virtual machine, effectively implementing a user-level garbage collector; and
- Chapter 6 details changes to a production compiler that were required to support a hardware garbage collector.

The second of these three was initially a set of AspectJ aspects, reproduced in Appendix A; the performance was poor, partly as a result of the AspectJ compiler at the time, partly as a result of the language design itself. Ultimately the aspectual implementation had to be augmented using a preprocessing tool we implemented to create specialized, type-specific

advice. The other two system aspects were recognized for their aspectual qualities at design time, but were not implemented in AspectJ because of language limitations.

The following sections recap the failures encountered with aspect language implementations of the system aspects described in previous chapters.

7.1 Performance problems

Section 5.5 indicates the unacceptable performance of the aspect implementation of the reference-counting memory manager described in Chapter 5. This was almost entirely due to structural reflection (on classes) that *could* be performed at compile-time; however, there is no way to specify such type manipulation in AspectJ.

7.2 Lack of type-sensitive constructs

Not only do the lack of type-sensitive constructs in AspectJ lead to performance problems, they also don't permit certain types of analysis and reconfiguration to be conducted by the program during compilation. This was the chief problem of an aspectual formulation of the scheduling metaprogram of Chapter 4. The metaprogram *could* have been written as an aspect, but this wouldn't have permitted the compile-time configuration we desired, and it certainly wouldn't have supported raising a compile-time error when the task set was infeasible. In short, because compile-time computation wasn't supported, the functionality we desired was not available.

7.3 Model mismatch for system-level concerns

Chapter 6 demonstrates a rather different problem; AspectJ is simply unable to specify some of the needed functionality, including array access and assignment and local variable manipulation. This represents a *join point model mismatch*—AspectJ's join point model is simply at a different level than that required by the software support of Chapter 6.

7.4 Systemic aspect languages

We now consider what features we could add to an aspect language to solve the problems outlined in the previous sections.

7.4.1 Reflective aspect code

Consider a piece of generic AspectJ advice designed to reflectively iterate over the fields of an object and perform some processing. Such advice might look similar to the advice we used for heap-reference-counting objects in Chapter 5, a simplified version of which is shown in Figure 5.6. (The full code is in Appendix A starting on page 174.)

We use reflection heavily in the code.[1] This reflection is necessary to make the advice generic enough to work with all classes of objects. However, it is not necessary or preferable in an implementation for performance reasons, as noted in Section 5.5.

Chapter 5 demonstrates the tremendous benefits of partially evaluating reflective code like this to remove oft-used reflection. This scheme of automatically generating type-specific advice may be useful in aspect compilers for boosting performance, but it only addresses part of the problem.

Java's programmatic reflection is a mechanism to which programmers resort when necessary, but it certainly is not the language feature of choice in all circumstances—other Java features, like class inheritance and virtual dispatch, are preferable in many cases, not just for performance but also for their ability to better encapsulate and encode programmer intent and aid future maintenance effort and code reusability—in short, the right tool for the right job. Aspect languages like AspectJ provide additional, non-reflective features that users often prefer to apply instead of Java reflection.

Not only, then, could aspect compilers support non-reflective implementation or partial evaluation where possible, but aspect languages could provide also the ability to

[1] We use the term "reflection" in this circumstance somewhat reluctantly. Java boasts "reflection" as a feature but in fact merely provides *introspection*. The behavior of a class cannot be changed after it is defined, and, in particular, the behavior of one or more instances of a class cannot be changed (nor can their class identity be changed) after instantiation. Virtual machine-level features cannot be reflected upon: virtual method invocation, for example, does not have a reflective language-level interface. These are not the feature's only deficiencies. See [80] and [65] for a more complete vision of reflection.

perform some types of reflection with a syntactic effort similar to that of the AspectJ pointcut. When an aspect compiler applies a particular piece of advice to a particular join point, it can reify these gestures, as appropriate, to the weave being performed. With a second gesture, similar in spirit to the Java *generic type* [24] or the C++ template, the programmer could specify generic aspects parameterized by type rather than the primordial Java object type `Object`. This removes the need for programmer-specified reflection and, further, permits the programmer the flexibility to assert that certain members exist in the classes to which the advice applies; full type checking is only performed when the advice is woven.

The signature of the advice of Figure 5.6 (or that in Appendix A) then becomes parameterized on types `T` and `U`, instead of specified statically for types `Object` and *ReferenceCountable*. Statically-instantiated parameterization only provides advice genericity to a point, as the types `T` and `U` that are instantiated at a particular join point may themselves have subclasses. The parameterization represents the static nature of the join point's types, and Java's dynamic features, including reflection, can pick up at that point.

Such parameterized advice would permit an aspect programmer to write generic aspects in a more natural way and benefit from some compile-time type checking that reflection simply does not provide. A parameterized mechanism for AspectJ's static crosscutting feature, *intertype declarations*, has been proposed by Hanenberg and Unland [51].[2]

This mitigates the problems identified in Section 7.1 and Section 7.2.

7.4.2 Systemic join point model

A *systemic join point model* would alleviate the problems of Section 7.3. For instance, adding *read-barrier* and *write-barrier* join point primitives, parameterized by the kind and arguments of access and assignment, could solve the issue and make the compiler modifications of Chapter 6 expressible using the (extended) aspect language.

[2]Please note that the cited work calls intertype declarations by a different name, *introductions*. See Section 2.1.5 for a discussion of the name discrepancy.

7.4.3 Generalized join point model

The proposal of Section 7.4.2 is not a fully satisfactory solution, as it would bias the (hypothetical) aspect language toward a specific *kind* of system aspect. Another, more general solution is to define an extensible join point model that would allow aspectual code to cooperate in its own weaving.

This would look much like a compile-time metaobject protocol [65]. Language mechanisms could be represented in the host language using higher-order abstract syntax [91], and the weaving process itself designed to be extensible.

The resulting language, though it knows nothing of read- and write-barriers directly, would be capable of expressing concerns about them. This addresses the issue identified in Section 7.3.

7.5 Chapter summary

We have identified places where AspectJ is unable to specify system aspects, and we have identified possible solutions to these problems that future aspect languages could incorporate.

Part II

Reducing the Testing Burden

The language designer must not ask "what do you want?", but rather "how does your problem arise?" For, the answer to the first question will inevitably be "jumps, type-less operands, and addresses."

– *Niklaus Wirth, On the Design of Programming Languages [115]*

Chapter 8

The Testing Problem

Modern tools support a *feature-oriented* approach [17, 102] to writing software: a minimal core is written, and features are independently added atop this core. These features can be included in or excluded from a particular build, so the software naturally can form an entire product line, allowing users to select only the features they want from the software.

This chapter identifies two outstanding problems in this area: that of enumerating valid configurations of the software, and that of testing the (possibly quite large) number of such configurations. The first problem is solved algorithmically: we provide an algorithm that enumerates the set of valid configurations in time proportional to the size of this (output) set. The second problem is discussed and addressed further in later chapters.

8.1 Introduction

A serious challenge of industrial middleware is that of its size; while a piece of middleware may start out life as a single-purpose, simple, small package, it can grow large and become unwieldy. Regular refactoring takes a large amount of developer time, and nonobvious dependences between different parts of the software makes the task difficult [53]. Large middleware with interdependent pieces can be too large for some tasks, especially for embedded systems with small memories and little or no secondary storage.

One way to achieve a small footprint is to design software to be *subsettable*. Users of the software can choose precisely those components they want and leave out the balance. However, as the number of independently selectable features grows, so the number of valid configurations of the software grows. This considerably increases the testing liability for the software author, who may be required to test every such configuration to ensure software quality. In practice, the number of configurations can lead to a heavy testing burden.

This chapter investigates this problem. Section 8.2 describes subsetting in more depth. Section 8.3 provides a novel algorithm for enumerating the tests given a specification of valid configurations. Section 8.4 introduces the notion of *feature dependence* and indicates how it might inform a solution to the testing problem, and Section 8.5 summarizes.

8.2 Subsetting

Software *subsetting* is the practice of refactoring software code so that it may be built in multiple *configurations*, each consisting of a subset of the functionality of the full software package. To a user with specific requirements, a subsettable software package can provide a targetted build that has just the needed functionality without spending processor time and disk and memory storage on other, unwanted tasks.

In practice, subsetting software is a hard problem after the fact [53]. Many modern software projects opt to make subsetting a design goal from the start [100, 42, 92, 55].

Subsetting can be achieved in a variety of ways. Linux kernel modules [22] and web browser content plugins and interface extensions involve subsetting: users can select precisely what parts they need. Often in such systems, software functionality is added only through a well-defined, rigid interface based on the *Interceptor design pattern* [93]. This is suitable for web browser content plugins, where the browser code itself should control when and how the plugin is invoked; however, certain highly-pervasive software features cannot be adequately specified through such a rigid interface.

Recent work uses aspect-oriented programming to implement different parts of software functionality so that they needn't be constrained to a particular interface [92, 55, 79, 53], and it is on this approach that we focus our attention in this work. Aspect-oriented languages, defined in Chapter 2, provide language-level hooks; different software features

can use these hooks in different ways to implement their functionality. Structural modifications of code, such as with AspectJ's intertype declarations, allow these features to enlarge shared data structures in a modular way. Traditional compile-time configurability of data structures, not using aspects, can waste memory at runtime or compromise code maintainability.[1]

A rigid interface based on the interceptor pattern has its advantages, though. In particular, program state can be protected from the plugin, essentially isolating its functionality and protecting the core software. As the interface is relaxed to accommodate more extensive and pervasive feature specifications, however, the core software code necessarily becomes vulnerable to state changes imposed by the added functionality, and its memory and processing requirements merge with those of the added feature. The core software code and the feature specification may be separate at the source level, but at runtime they are one, whole.

This means that adding a feature to an error-free core can cause an error in that core; further, the composition of two (independently) error-free features to an error-free core can exhibit an error at runtime. In general, this means that *every possible configuration* of the software must be tested to ensure software quality. If every feature in the feature set $\mathcal{F}$ is independently selectable, the possible configuration space is the power set of features, $2^{\mathcal{F}}$.

This exponential explosion of valid software configurations quickly becomes unmanageable as the number of features grows. If not all $2^{|\mathcal{F}|}$ feature subsets are valid configurations, the valid configurations among them must be found. Then they must all be tested. The rest of this chapter looks at enumerating valid configurations and reducing the testing burden placed on software authors.

[1] Consider a C program written to have a compile-time-configurable data structure. One approach is to make heavy use of the C preprocessor's `ifdef` feature. Code maintainability can be sacrificed when there is considerable complexity in the `ifdef` conditions. A second approach is to install a pointer in a data structure that can be used at runtime to point to fixed-size additional data. Essentially a non-inlined version of the first approach, this wastes memory and execution time. It may provide a clearer definition of the data structure, but sacrifices some code maintainability since the handling of the linked structures is more complex than necessary.

8.3 Finding valid configurations

When not every software feature is independently selectable, one of the problems of testing the software is merely determining which configurations are valid; *valid configurations* satisfy all the feature composition requirements of the software under consideration. If a feature $A \in \mathcal{F}$ requires the presence of a feature $B \in \mathcal{F}$, then the configuration $\{A, B\}$ may be valid but the configuration $\{A\}$ certainly is not. The mechanism for specifying these feature requirements is a graph, as explained in Section 8.3.1.

While a simple algorithm can iterate over all $2^{|\mathcal{F}|}$ possible configurations, filter out invalid ones, and list those remaining, this is an exponential-time algorithm even when the number of valid configurations is small. This section proposes a novel graph-theoretic algorithm to enumerate valid configurations with an asymptotic running time in the size of the output set. The author is not aware of a previous algorithm that performs this enumeration with this complexity bound.

The work of this section was inspired by Ravi Pratap Maddimsetty and his work testing the thousands of legitimate configurations of Framework for Aspect Composition for an EvenT channel (FACET) [56, 79].[2]

8.3.1 Feature set specifications

To enumerate valid configurations, we take as input a *feature set specification* that describes the relationship between features. In our case the feature set specification is provided by the build system, which often includes some encoded knowledge about feature relationships, or by the developer personally. It may be possible in many circumstances to determine automatically these feature relationships rather than require them as input, but that is not investigated in this work.

We augment the complete feature set $\mathcal{F}$ with a *core feature* that stands for the base program with no optional features. The feature set specification is then a directed acyclic graph $G = (V, E)$ with $V = \mathcal{F}$. A feature $A \in \mathcal{F}$ that requires the presence of another feature $B \in \mathcal{F}$ represents that requirement with a directed edge $(B, A) \in E$. Note this edge is in the *reverse* of the dependence direction. G will be rooted at the core feature,

[2] Ravi Pratap Maddimsetty, personal communication.

as all features depend upon the core, and the core depends on no feature. All vertices are reachable in the graph from the core feature.

This perspective of feature dependence is based on FACET [56]. The FACET build system can be made to output just such a graph of its features, though its feature set specification is somewhat richer than we use in this chapter; it allows features to be added to mutual exclusion sets. Figure 10.13 on page 141 shows FACET's full feature set specification.[3]

8.3.2 A configuration-enumerating algorithm

Our feature set specification now a graph $G = (V, E)$ with a designated "core feature" root vertex r, we address the problem as a purely graph-theoretic one.

Definition 8.1 *Immediate dominator.* We define the *immediate dominator* of a vertex v, written $idom(v)$, to be the node d such that:

- all paths from r to v in G include d, and
- there exists no single vertex y that lies strictly between d and v on *all* of these paths.

We also use the inverse *idom* mapping, written $idom^{-1}$. $idom(v)$ (where $v \neq r$) is the *single* vertex that immediately dominates v. $idom^{-1}(v)$ is the possibly-empty *set* of vertices immediately dominated by v.

Definition 8.2 *Dominance and strict dominance.* For vertices $x, v \in V$, we say x *dominates* v, written $x \underline{\gg} v$, if and only if $x = v$ or x is in the transitive closure of $idom(v)$. We say x *strictly dominates* v and write $x \gg v$ if and only if $x \underline{\gg} v$ and $x \neq v$.

These dominance definitions are compatible with that of [84] and other texts.[4]

[3] Note that FACET's feature set specification is the *inverse* of the one we use here: the directed edges point in the opposite direction.

[4] We note that [84] and other references typically introduce *dominance* on execution paths through a program. Here we are defining it in the language of graph theory, independent of execution paths, and using it on feature set specification graphs. The differences here are superficial; the ideas are complementary and compatible.

Definition 8.3 *Rooted subgraph.* Given a directed acyclic graph $G = (V, E)$ and a vertex $v \in V$, we define a *v-rooted subgraph of G*, denoted *Subgraph*(G, v), to be the subgraph $S = (V', E')$ of G rooted at v with all vertices in S dominated by v:

$$\begin{aligned} V' &= \{u \in V \mid v \gg u \text{ (in } G)\} \\ E' &= \{(a, b) \in E \mid a, b \in V'\} \end{aligned}$$

Definition 8.4 *Valid configuration.* Given a finite, rooted, directed acyclic graph $G = (V, E)$, call any nonempty set of vertices $S \subseteq V$ a *valid configuration of G* if $v \in S \Longrightarrow$ *pred*$(v) \subset S$. We write $\mathcal{V}_G$ to denote the set of all valid configurations of a graph G.

Definition 8.5 *Owned vertex.* A graph vertex u is said to *own* a vertex v if *pred*$(v) = \{u\}$.

Given these definitions, the graph-theoretic problem statement is then as follows:

Problem statement. For a finite directed acyclic graph $G = (V, E)$ rooted at a single vertex $r \in V$ (all vertices reachable from, and therefore dominated by, r), find $\mathcal{V}_G$, the set of all valid configurations of vertices of G.

We have discovered an algorithm that computes the set of all valid configurations of G with asymptotic time complexity proportional to the size of the number of valid software configurations. Further, this solution performs as well as *any* solution to the problem: *any* algorithm solving this problem must enumerate the entire set $\mathcal{V}_G$, and therefore must have computation complexity at least $O(\mathcal{V}_G)$.

The algorithm proceeds as follows. Following the technical presentation, a discussion tries to illuminate the algorithm, and correctness and complexity are proven.

Algorithm. Pseudocode for the top-level algorithm is listed in Figure 8.1. First, the immediate dominators *idom*(v) for all $v \in V$ are calculated. Now, each vertex $v \in V$ is considered in reverse topological order and *find*(G, v) is computed (pseudocode in Figure 8.2). Where *find* appears to call itself recursively, it can in fact use a previously computed value for the invocation.[5]

find(G, v) computes the set of all valid configurations of *Subgraph*(G, v). Reverse topological order guarantees that r is the last vertex considered, so *find*(G, r) is the last

[5]Computing in reverse topological order ensures this property.

```
configurations(G = (V, E), r) :
    begin
        compute idom^{-1} mapping
        consider vertices v ∈ V \ {r} in reverse topological order
            compute find(G, v)
        return find(G, r)
    end.
```

Figure 8.1: Our algorithm to enumerate valid software configurations based on a feature set specification (see Section 8.3.2).

find computation performed; $find(G, r)$ is the set of all valid configurations of G as defined in the problem.

The *cross* operator $\times$ is defined a little differently than it normally is to allow for crossing multiple unordered sets together (Figure 8.3). Examples are useful here. If w, x, y, and z are configurations (sets of features), A and B are sets of configurations (sets of sets of features), and $\alpha, \beta \in \mathcal{F}$ are features:

$$\begin{aligned} \{w, x\} \times \{y, z\} &= \{w \cup y, w \cup z, x \cup y, x \cup z\} \\ \{\{\alpha, \beta\}\} \times \{x, y\} &= \{\{\alpha, \beta\} \cup x, \{\alpha, \beta\} \cup y\} \\ A \times \emptyset &= \emptyset \\ A \times \{\emptyset\} &= A \end{aligned}$$

8.3.3 Discussion

The key to this algorithm is that it exploits a recursive, independent subgraph structure. Diagrammed in Figure 8.4, these recursive are reminiscent of recursive structure in dynamic programming [32]; in fact, that is quite similar to what is done here.

The *configurations* subroutine performs all of the main graph work; *find* computes a set of valid configurations for one of these subgraphs based on what is already known;

```
    find(G = (V, E), v) :
        begin
(1)         G' = (V', E')  := Subgraph(G, v)
(2)         combos  := { {v} }
(3)         owned(v)  := { w ∈ V' | pred_G'(w) = {v} }
(4)         foreach s ∈ 2^owned(v) \ {∅}
(5)             foreach t ∈ X_{c∈s} find(G', c)
(6)                 combos  :=  combos ∪ { {v} ∪ t }
            combos'  :=  ∅
(7)         foreach i ∈ idom^-1_G'(v) \ owned(v)
(8)             foreach t ∈ { x ∈ combos | pred_G'(i) ⊆ x }
(9)                 combos'  :=  combos'  ∪ ( {t} × find(G', i) )
(10)        return combos ∪ combos'
        end.
```

Figure 8.2: The *find* helper subroutine to compute all valid configurations of $Subgraph(G, v)$.

$$A \times B \;=\; \left\{ x \cup y \;\middle|\; x \in A,\ y \in B \right\}$$

$$\begin{aligned} \bigtimes_{z \in \{x\} \cup B} f(z) &= f(x) \times \bigtimes_{z \in B} f(z) \\ \bigtimes_{z \in \emptyset} f(z) &= \{\emptyset\} \end{aligned}$$

Figure 8.3: Definition of the *cross* operator $\times$, which operates on sets of configurations (sets of sets of features). Intuitively, it computes the set of all paired unions of two sets of configurations. As defined here, $\times$ is commutative and associative.

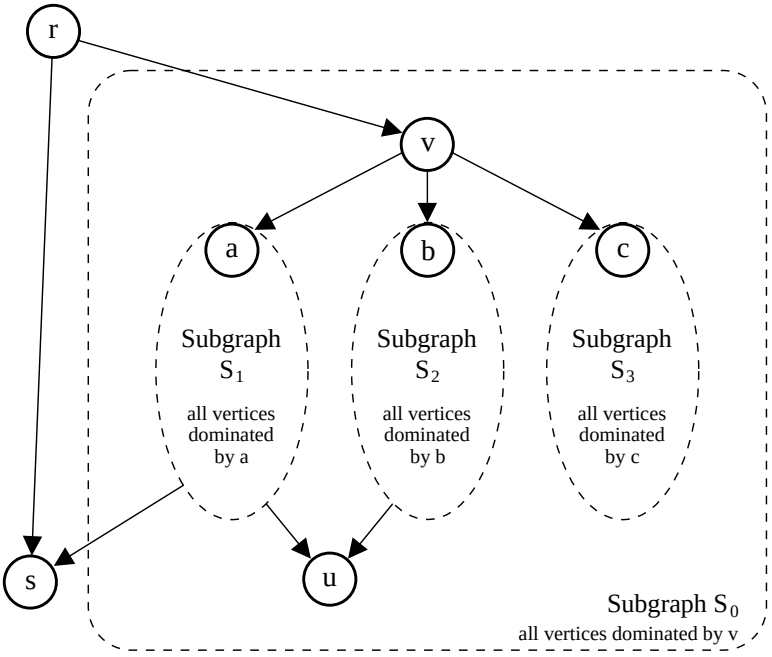


Figure 8.4: Recursive independent subgraphs.

the "recursive" call to *find* has in fact been precomputed by *configurations* in a bottom-up fashion, as in dynamic programming. This is shown in the next section.

The *find*(G, v) subroutine computes a "*combos*" set in three stages. Initially, it only contains $\Big\{\{v\}\Big\}$—the single configuration of only the root vertex ("core feature") of *Subgraph*(G, v). Clearly this is a valid configuration for *Subgraph*(G, v), but it may not be a complete set of valid configurations.

The second contribution to *combos* is at line (6). Line (4) iterates through all combinations of "owned" successors of v (recall Definition 8.5, above); s is one such set of owned successors. Line (5) computes the *cross* of all valid configurations of *Subgraph*(G, c); here, c ranges over the owned successors of v. For each configuration t in the cross, *combos* is amended at line (6) to include the configuration $\{v\} \cup t$.

Consider Figure 8.4. In the computation of *find*(G, v), the second contribution to *combos* corresponds to all valid combinations of valid configurations of subgraphs S_1, S_2, and S_3. Vertices a, b, and c are owned by v. Feature v is included in every valid configuration of S_0; it *must* be. Features r and s are not dominated by v, so aren't considered by *find*(G, v). Feature u *is* dominated by v, but is not considered by lines (4)–(6).

The third and final contribution to *combos*, at line (9), explains its name; this contribution is a *combination* of those sets of configurations calculated in the previous two contributions. Line (7) iterates over each vertex i that is immediately dominated by *but not owned by* v. This corresponds to vertices like u in Figure 8.4, those for which the immediate dominator is not the sole predecessor. Line (8) considers each configuration t in the *combos* set that includes all predecessors of i. Each such configuration can be augmented with i to form another valid configuration (by definition). Line (9) adds such configurations to *combos*; it also includes all valid configurations of $\mathit{Subgraph}(G, i)$, since i could dominate its own recursive independent subgraph (even though vertex u in Figure 8.4 does not).

Note that line (9) may execute multiple times, each time using the current assignment of *combos*; in this way, the augmentations to *combos* in the third contribution are multiplicative.

With these three contributions, the *combos* set contains all valid configurations of $\mathit{Subgraph}(G, v)$.

8.3.4 Correctness

To show that this algorithm is correct, we compare $\mathcal{V}_G$ to $\mathit{find}(G, r)$, where r is the designated, unique root of G. We show *soundness* ($\mathit{find}(G, r) \subseteq \mathcal{V}_G$) and *completeness* ($\mathcal{V}_G \subseteq \mathit{find}(G, r)$), which together prove that we have found the right set of valid configurations, that is, $\mathit{find}(G, r) = \mathcal{V}_G$.

Theorem 8.1 (*find* can use G) *find* computes G', but in all cases it can use G instead. Specifically, line (1) of *find* computes $G' = \mathit{Subgraph}(G, v)$; replacing line (1) with $G' := G$ results in equivalent output.

Proof Lines (3), (5), (7), (8), and (9) use G'.

- Line (3). For all vertices w in $\mathit{Subgraph}(G, v)$, by definition, $v \underline{\gg} w$. If any predecessor of w fell outside the subgraph, w would not be dominated by v. A contradiction results: therefore, $\mathit{pred}_{G'}(w) = \mathit{pred}_G(w)$.
- Line (5). c is a vertex dominated by v. Every vertex c dominates is thus also dominated by v; $\mathit{Subgraph}(G', c)$ is itself a subgraph of $\mathit{Subgraph}(G, v)$. Further, all vertices of G that c dominates are in G' and thus also in $\mathit{Subgraph}(G', c)$. It follows that $\mathit{Subgraph}(G, c) = \mathit{Subgraph}(G', c)$ and therefore $\mathit{find}(G, c) = \mathit{find}(G', c)$.

- Line (7). All nodes immediately dominated by v in G are also in $\mathit{Subgraph}(G, v)$. Further, all nodes immediately dominated by v in G' are also in G. Thus $\mathit{idom}^{-1}_{G'}(c) = \mathit{idom}^{-1}_{G}(c)$.
- Line (8). Same as for line (3).
- Line (9). Same as for line (5).

Because every use can be replaced with a use of G, an explicit computation of G' is not needed. ∎

Theorem 8.2 If $s, t \in \mathcal{V}_G$ for a rooted, directed acyclic graph G, then $s \cup t \in \mathcal{V}_G$.
Proof By definition, if s is a valid configuration of G, then the existence of a vertex $u \in s$ implies $\mathit{pred}(u) \subset s$. Similarly, $v \in t$ implies $\mathit{pred}(v) \subset t$. Therefore, for each such u and v, we know that $u, v \in s \cup t$, that $\mathit{pred}(u) \subset s \cup t$, and that $\mathit{pred}(v) \subset s \cup t$.

Assume that $s \cup t$ is *not* a valid configuration of G. Then there must exist a vertex w and another vertex $x \in \mathit{pred}(w)$ such that $w \in s \cup t$ and $x \notin s \cup t$. But this is a contradiction—if $w \in s$, then $\mathit{pred}(w) \in s$, and we know that $s \subseteq s \cup t$. Symmetrically, a similar contradiction exists if $w \in t$. Thus, $s \cup t$ is a valid configuration of G. ∎

Theorem 8.3 Given a rooted, directed acyclic graph $G = (V, E)$, if a vertex $u \in V$ owns vertex $v \in V$, then $u = \mathit{idom}(v)$.
Proof Assume $u \not\gg v$ in G. Then there exists some path from the root of G to v that doesn't include u. But this is a contradiction: by definition, the only predecessor of v is u, so u must be the penultimate vertex on any path in G from the root to v. So $u \gg v$, and because u is the immediate predecessor of v, $u = \mathit{idom}(v)$. ∎

Theorem 8.4 If $s, t \in \mathcal{V}_G$, then $s \cup t \in \mathcal{V}_G$.
Proof By definition, $x \in s \implies \mathit{pred}(x) \subset s$, and similarly for t. Because $\mathcal{V}_G$ is the set of *all* valid configurations, it suffices to show that $x \in s \cup t \implies \mathit{pred}(x) \subset s \cup t$, which is trivial. ∎

Theorem 8.5 If $A, B \subseteq \mathcal{V}_G$, then $A \times B \subseteq \mathcal{V}_G$.
Proof From the definition of $\times$ (Figure 8.3), if $s \in A$ and $t \in B$, then $s \cup t \in A \times B$. We know $s, t \in \mathcal{V}_G$; an application of Theorem 8.4 completes the proof. ∎

Theorem 8.6 (Correctness) For any directed acyclic graph $G = (V, E)$ rooted at a vertex $r \in V$ with $r \underline{\gg} v$ $(\forall v \in V)$, $\mathit{find}(G, r) = \mathcal{V}_G$.

Proof As G is finite, directed, and acyclic, the length of the longest path through G is finite. The proof is by induction on the length of the longest path in G. Let $\mathit{MaxPathLength}(G)$ be the length of the longest path.

Base. If $\mathit{succ}(r) = \emptyset$, then the longest path in G is of length 1, and $\mathit{find}(G, r)$ returns $\Big\{\{r\}\Big\}$, which is trivially correct.

Inductive step. With r the root of G, let:

$$k = \max_{c \in \mathit{succ}(r)} \mathit{MaxPathLength}(\mathit{Subgraph}(G, c))$$

The longest path through G is clearly larger than for any of these subgraphs (prepending r to the longest path of any subgraph results in a longer path in G); therefore, by the induction hypothesis we know that $\forall c \in \mathit{succ}(r)$, $\mathit{find}(G, c) = \mathcal{V}_{\mathit{Subgraph}(G,c)}$.

Assume that $\mathit{find}(G, r) \neq \mathcal{V}_G$; then it must either include an extra, invalid configuration that *is not* present in $\mathcal{V}_G$, or it must miss a valid configuration that *is* present in $\mathcal{V}_G$.

Extra configuration. Assume there is an extra configuration $s \in \mathit{find}(G, r)$ such that $s \notin \mathcal{V}_G$. Then $\exists x \in s$ s.t. $\mathit{pred}(x) \not\subset s$. Clearly $x \neq r$, since $\mathit{pred}(r) = \emptyset$. Consider *find* lines (2), (6), and (9): these are the only contributions to its return value. Line (2) cannot produce s, since we know $x \neq r$. Line (6) produces configurations of the shape $\{v\} \cup t$, where t is a cross-configuration of sets of configurations that are known valid by the induction hypothesis; by Theorems 8.4 and 8.5, these configurations must too be valid. Line (9) produces configurations of the shape $\{t\} \times \mathit{find}(G', i)$; t is valid by the arguments above, and $\mathit{find}(G', i)$ is valid by Theorem 8.1 and the induction hypothesis. Using Theorems 8.4, and 8.5, the combination configurations constructed on line (9) must too be valid. Lines (2), (6), and (9) cannot produce configurations of the right shape to be "extra," invalid configurations. We have arrived at a contradiction; $\mathit{find}(G, r)$ cannot include a configuration $s \notin \mathcal{V}_G$.

Missing configuration. Call the missing configuration s: $s \in \mathcal{V}_G$ and $s \notin \mathit{find}(G, r)$. By definition of *valid configuration*, $x \in s \implies \mathit{pred}(x) \subset s$. Consider the set $t = s \backslash \bigcup_{x \in s} \mathit{pred}(x)$; t is the set of independent features of configuration s; all other features are pulled in by their dependence and by definition. Now consider each feature $x \in t$. x is either a successor of r or it is not. If it *is not* a successor of r, it was considered in the computation $\mathit{find}(G, c)$ for some successor c of r, and we know $\mathit{find}(G, c)$ to be correct by the induction hypothesis.

If it *is* a successor of r, it is immediately dominated by r and is considered at either line (6) or (9) and included in the *combos* set. Since all subsets of such successors of r (and their *find* sets) are considered and crossed, t must be considered, and $s \in \mathit{find}(G, r)$, a contradiction.

There is a contradiction in each case: therefore $\mathit{find}(G, r)$ doesn't miss any set $x \in \mathcal{V}_G$, and doesn't include an extra set $x \notin \mathcal{V}_G$. Thus $\mathit{find}(G, r) = \mathcal{V}_G$. ∎

We have established the correctness of *find* and (by extension, trivially) the *configurations* subroutine.

8.3.5 Computational complexity

Next we investigate the computational complexity of the *configurations* subroutine. We already know that references to G' can be replaced by references to G (Theorem 8.1); this considerably lowers the computational cost of an implementation.

Theorem 8.7 A program implementing the *find* algorithm can use *array* and/or *list* data structures in all set computations without having to check for duplicate elements.
Proof For each assignment, duplicate elements cannot result.

Line (2). Trivial assignment.

Line (3). The *owned* mapping can be computed for all vertices as indicated in *configurations*; see below.

Line (6). t does not include v; each set added to *combos* at this line includes v.

Line (9). Each configuration added to *combos′* is not already a member; for different iterations of the outer loop, i is different and is not included in *combos′* for any other iteration of the outer loop. For different iterations of the inner loop, values of t are distinct.

Line (10). Each element of *combos′* includes some feature i that is not present in any configuration in *combos*; therefore the set union doesn't risk a duplicate element. ∎

Theorem 8.8 Let $G = (V, E)$ be a finite, rooted, directed acyclic graph, and let $\mathit{owned} = \bigcup_{u \in V} \{v \in V \mid u \ \mathit{owns}\ v\}$. Then the **foreach** loop on line (7) of the *find* algorithm executes precisely $|V| - |\mathit{owned}| - 1$ times over all *find* invocations.

Proof For each vertex $v \in V$, precisely one vertex is the immediate dominator of v. Therefore the inverse mappings $idom^{-1}(v)$ are disjoint for $v \in V$. Ownership sets are also disjoint; if a vertex r owns a vertex v, no other vertex owns v.

Further, if a vertex r owns a vertex v, $v = idom^{-1}(r)$, so all vertices trimmed from the $idom^{-1}$ mapping on line (7) are in that mapping. Therefore, for a given invocation of $find(G, r)$, the **foreach** loop on line (7) executes exactly $|idom^{-1}(r)| - |\{v \in V \mid r \ owns \ v\}|$ times.

In a rooted, directed acyclic graph, all nodes have an immediate dominator except for the root. Therefore, $\sum_{v \in V} |idom^{-1}(v)| = |V| - 1$. Since the ownership sets are disjoint, $|owned| = \sum_{u \in V} |\{v \in V \mid u \ owns \ v\}|$.

Therefore, if calls to *find* are memoized and performed in reverse topological order over G, line (7) executes precisely $|V| - |owned| - 1$ times for a call to $configurations(G)$. ∎

Theorem 8.9 The **foreach** loop on line (5) of the *find* algorithm always executes at least once, for any value of s picked at line (4).

Proof In a call to $find(G, v)$, line (4) picks an element s from the power set of $owned(v)$, but excludes the empty set. Therefore, s is always a set of at least one element. By the definition of $\times$, the result is not empty. Therefore, there is always at least one t which can be chosen at line (5), and that **foreach** loop must execute at least once. ∎

Theorem 8.10 When the **foreach** loop on line (5) of the *find* algorithm executes, it increases the cardinality of *combos* by precisely one.

Proof The sets yielded by $find(G, c)$, over all $c \in s$, are disjoint, as $find(G, c)$ operates on the disjoint subgraph of G rooted at c. By definition, the $\times$ operator chooses an element from each set, so the result sets yielded by the $\times$ operation at line (5) must also be disjoint.

Therefore, the sets t are disjoint over all iterations of the outer **foreach** loop. Then, $\{r\} \cup t \notin combos$ before the **foreach** loop at line (5) executes, and, therefore, the $\cup$ operation increases the cardinality of *combos* by one. ∎

Corollary 8.10.1 The running time of the **foreach** loop at line (4) is $O(|combos|)$.

Proof Immediate from Theorem 8.9 and Theorem 8.10. ∎

Theorem 8.11 The running time of $configurations(G)$, which includes $|V|$ calls to $find(G, v)$, is $O(|find(G, v)|)$, that is, the size of the result set.

Proof Computing the required mappings (*succ*, *pred*, $idom^{-1}$, and *owned*) requires $O(|E|)$ time.[6] The runtime of the entire **foreach** loop of line (4) is $O(|combos|)$ by Corollary 8.10.1, and the **foreach** loop of line (7) executes less than $|V|$ times over all calls to *find* by Theorem 8.8. $|V| - 1 \leq |E| < |find(G, r)|$, so the entire procedure requires $O(|find(G, r)|)$ time. ∎

Theorem 8.12 Given a directed acyclic graph $G = (V, E)$, rooted at r, computing *pred*, *succ*, *owned*, and $idom^{-}1$ mappings over the vertices of G and computing $find(G, r)$ can be done in $O(|\mathcal{V}_G|)$ time.

Proof Let $n = |V|$ and $m = |E|$. The computation of *succ* and *pred* mappings takes $O(m)$ time. The $idom^{-}1$ mapping can be computed in $O(m\,\alpha(m + n\ ,\ n) + n)$ time.

Every iteration through either of the *find* loops adds elements to the result set. Computing $\alpha \times \beta$ can be performed in $O(|\alpha| \cdot |\beta|)$ time, and yields an output set of size $|\alpha| \cdot |\beta|$. Therefore each iteration of the first loop can be performed in time proportional to the number of configurations added to the *combos* set. Each iteration of the second loop adds at least one element to the *combos* set.

Therefore the *find* operation can be done in time proportional to the size of its output set, that is, in $O(|combos|)$ time. Since $find(G, r) = \mathcal{V}_G$ (by Theorem 8.6), *find* can be calculated in $O(|\mathcal{V}_G|)$ time. ∎

This concludes our discussion of our valid configuration-enumerating algorithm.

8.4 Feature dependence

In the preceding section, we used feature dependence, represented in the feature set specification, as a way to rule out illegitimate configurations. These dependences are typically structural or behavioral in nature—one feature *depends* on a method introduced by another, or *depends* on another's behavior to make any sense in the system at all.

Another sort of dependence may be useful here as well, though. If two features manipulate completely independent, non-interfering parts of the core application, then they

[6] Computation of dominators dominates. Dominators computation is linear [27]; the well-known, near-linear dominators algorithm of Lengauer and Tarjan [72] can also be assumed, in which case the running time is $O(|E|\,\alpha(|E|, |V|))$.

might be testable in isolation. That is, if features A and B are independently tested, and each passes a rigorous set of tests, perhaps their combination need not be tested also.

The next few chapters explore this idea.

8.5 Chapter summary

Modern software is complex; often, software becomes large in footprint and its features difficult to maintain. To address excessive footprint, developers refactor their software or design it initially to support sets of user-selectable features; each user can then choose just the right feature set to fit the task at hand.

This chapter introduced the *testing problem*, which arises in precisely such circumstances. In an effort to ensure software quality, authors of such subsettable, featureful software must test all valid configurations of software.

First, testers must have a list of all legitimate software configurations; this chapter introduced a novel algorithm to enumerate such configurations and established a bound on its computational complexity. This algorithm is asymptotically as fast as any algorithm to solve the problem, since it runs in time proportional to the size of its output.

Second, the number of tests required can be staggering in practice; in general, it is exponential in the number of features. *Feature independence* might be an successful criterion for reducing the testing burden. Forthcoming chapters examine feature dependence in depth.

Chapter 9

Aspect Independence

Recent work has studied aspect independence. However, much of this work has defined independence in differing ways and in different contexts. Here, we briefly survey the different notions of *aspect independence* from the literature and conclude in Section 9.6 by defining independence as it pertains to this work. We also discuss the relevance of each view of independence to AspectJ [6], a widely-used, general-purpose, aspect-oriented programming language.[1]

The notion of aspect independence is particularly important in automatic program translation and analysis for a variety of reasons. First, information regarding the dependence of aspects upon one another is of use to the development team in packaging particular configurations of a software system as in Chapter 8.

Further, modular software can be tested in parts, and in particular, parts of software that provably do not affect each other (as independent *slices* [112, 89], discussed in Section 9.4) can be tested in isolation; this is particularly important for aspect-oriented software constructed in a feature-oriented manner, like FACET [55, 54, 53, 92, 79], with a large number of different configurations that each need to be tested or proven equivalent to other configurations (for the purposes of testing).

Finally, provable observations about aspect independence can be used by an optimization engine, resulting in higher-quality compiled code. For instance, if two aspects

[1] All code in this section has been tested with abc 1.2.1 [90] and AspectJ 1.2.0 [6].

provably do not interfere, code transformations that execute before weaving may reorder advice application, or perform optimizations over advice trigger points.

9.1 A taxonomy of dependence

There are several forms of aspect dependence; the term *dependence* is inconsistently used in the literature. To be explicit about our use of the term, we define five different forms of dependence:

- explicit dependence (Section 9.2)
- weave dependence (Section 9.3)
- control/data dependence (Section 9.4)
- sensitivity (Section 9.5)
- conceptual dependence (Section 9.6)

These are discussed in following sections of this chapter; we then discuss aspect dependence in the context of AspectJ (Section 9.7) and summarize (Section 9.8).

9.2 Explicit independence

First, we define a *valid program* to be a program that is valid within it's language: it is *well-formed* and passes any semantic checking required by the language.

An aspect A has an *explicit dependence* on an aspect B over a base program P if $B(P)$ is not a valid program but $B(A(P))$ is a valid program. A base "program" P has an *explicit dependence* on an aspect A if P is not a valid program but $A(P)$ is a valid program.

This captures the notion of packaging requirements: if an aspect A directly refers to a package, field, method, or other language entity belonging to aspect B, A requires the presence of B in any configuration of the software. If B is not presence, the compiler will flag an error.

The feature set specifications described in Chapter 8 model explicit dependence relations. Section 4.4.1 discussed this sort of dependence relation between real-time tasks in a task set.

9.3 Weave independence

Weave independence seeks to determine if a uniquely-determined woven program results from aspect weaving. Two aspects A and B are said to be *weave independent* if a uniquely-determined woven program results from applying either the transformation $A(B)$ or the transformation $B(A)$.

AspectJ does not require weave independence of AspectJ programs. The language specification [8] defines a weave order between pieces of advice within an aspect, but without specific instruction, the weave order between pieces of advice in different aspects is undefined and may occur in any order.

```
aspect Cancel {
  void around() :  call(void Foo.m()) {
    System.out.println("<CANCEL>");
  }
}
```

```
aspect A {
  declare precedence :  A, Cancel;
  void around() :  call(void Foo.m()) {
    System.out.println("<A>");
    proceed();
  }
}
```

```
class Foo {
  public static void m() {
    System.out.println("m()");
  }

  public static void main(String[] args) {
    m();
  }
}
```

Figure 9.1: Two weave-dependent aspects and their base program.

Figure 9.1 shows two aspects, `A` and `Cancel`, that are weave-dependent. In the figure, the two aspects both apply to the same join point. Weaving `A` first, then `Cancel`, leads to one outcome, while weaving `Cancel` then `A` leads to a another, distinct outcome. In this case, we employ AspectJ's *declare statement* to inform the compiler which weave order is desired.

9.4 Control/data independence

Let P be a program, G_P be its *control flow graph* (CFG) and D_P be its *program dependence graph* (PDG) [46]. We seek to find whether two aspects A and B interact in a control/data flow sense over program P—informally, whether the aspects' combined effects are greater than their sum.[2] We first define a few terms:

Definition 9.1 The *slice of a program dependence graph D_P with respect to vertex $s \in D_P$,* written D_P/s, is a directed graph:

$$\begin{aligned} V(D_P/s) &= \{w \in V(D_P) \mid w \rightarrow^* s\} \\ E(D_P/s) &= \{(v,w) \in E(D_P) \mid v, w \in V(D_P/s)\} \end{aligned}$$

Definition 9.2 The *affected points of P with respect to an aspect A,* written $AP_A(P)$, is a subset of the vertices of $D_{A(P)}$:

$$AP_A(P) = \left\{ s \in V(D_{A(P)}) \;\middle|\; D_P/s \neq D_{A(P)}/s \right\} \tag{9.1}$$

Intuitively, $AP_A(P)$ is the set of vertices of the woven program $A(P)$ that are not control/flow dependent on A—that is, they cannot detect the presence or absence of A.

Definition 9.3 Two aspects A and B are *control/data independent over P* if they satisfy two conditions:

$$AP_A(P) \cap AP_B(A(P)) = \emptyset \tag{9.2}$$

$$AP_B(P) \cap AP_A(B(P)) = \emptyset \tag{9.3}$$

There are two conditions since the aspects A and B may be weave-dependent. If one (but not both) of these conditions are satisfied, A and B are control/flow-independent for one weaving order and control/flow-dependent for the other. (This is most common where one aspect removes code that the other advises.) If both conditions are satisfied, the aspects A and B are control/flow-independent over P regardless of weave order.

[2] This exposition is inspired by, and some definitions are nearly identical to, the notes in Appendix B of [79].

There is no simple correspondence between weave independence and control/data independence in general. Aspects may be weave independent and control/data dependent, or weave dependent and control/data independent in this aspect model.

9.4.1 An example

Consider the aspectual code of Figure 9.2, defining aspects A and B and a base program (class) P.

We first construct the program dependence graph of P (Figure 9.3). Here, we use *basic blocks* [84] as the resolution of the graph (which we have indicated with comments in the code of Figure 9.2). Next, we individually weave A and B with the base program to produce programs $A(P)$ and $B(P)$. The program dependence graphs of these two woven programs ($D_{A(P)}$ and $D_{B(P)}$, respectively) are then constructed (Figure 9.4). Note that program $A(P)$ has an additional node (d) in its program dependence graph and additional edges (those incident to d) and that program $B(P)$ has *fewer* dependence edges than P.

We compute the affected points $AP_A(P)$ and $AP_B(P)$ (these are shaded in Figure 9.5), then apply the other aspect along each path and shade the affected points (Figure 9.6).

Finally, we can compute whether A and B are independent in a control/data-flow sense. Note that:

$$\begin{aligned} AP_A(P) \cap AP_B(A(P)) &= \{c, d\} \cap \{c\} \\ &= \{c\} \end{aligned}$$

and:

$$\begin{aligned} AP_B(P) \cap AP_A(B(P)) &= \{c\} \cap \{c, d\} \\ &= \{c\} \end{aligned}$$

So in this case, by Definition 9.3, A and B are *dependent*. Because of this dependence, they can affect each other at runtime; if this program is a feature-oriented program (Chapter 8) where $\{P\}$, $\{P, A\}$, $\{P, B\}$, and $\{P, A, B\}$ are valid configurations, this analysis cannot

```
aspect A {
  declare precedence : A, B; // or B, A

  int w;

  before() : set(int P.z) {
    // introduce basic block d
    w = P.x;
    System.out.println("A.d mod to c == "+w);
  }

  void around(int rhs) : set(int P.z) && args(rhs) && !within(A) {
    // mod to c
    proceed(P.x + P.y + w);
    System.out.println("A mod to z == "+(P.x+P.y+w));
  }
}

aspect B {
  void around(int rhs) : set(int P.z) && args(rhs) && !within(B) {
    // mod to c
    proceed(P.x + 4);
    System.out.println("B mod to c == "+(P.x+4));
  }
}

class P {
  static int x, y, z;

  public static void main(String[] args) {
    // basic block a
    y = 0;
    x = args.length;
    if(x > 3) {
      // basic block b
      y = 5;
    }
    // basic block c
    z = x + y;

    System.out.println(x);
    System.out.println(y);
    System.out.println(z);
  }
}

/* no advice  : java P foo bar baz ---> 3 0 3
   with A only: java P foo bar baz ---> 3 0 6
   with B only: java P foo bar baz ---> 3 0 7
   with A, B  : java P foo bar baz ---> 3 0 7
   with B, A  : java P foo bar baz ---> 3 0 6 */
```

Figure 9.2: Aspectual code (borrowing AspectJ syntax but using the nominal weaver of Section 2.2.1).

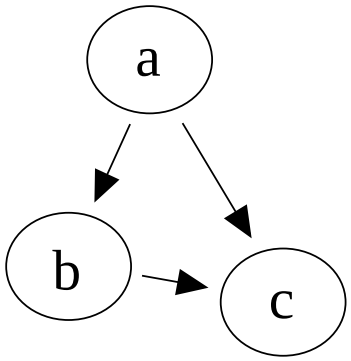


Figure 9.3: The program dependence graph of the code of Figure 9.2.

conclude that any tests are extraneous; we must test each of these feature subsets independently.

9.4.2 Discussion

As described here, determining control/data dependence and independence of two aspects is decidable but conservative, as it is based on a program dependence graph that is constructed to represent conservative approximations of program statements' dependence. Precise determination of aspect control/data dependence and independence is undecidable in general: aspects A and B could, for example, be constructed such that they are dependent if and only if the base program halts. In such a case, the conservative approximation described here would determine A and B to be dependent.

Application to testing. If two aspects A and B are determined to be independent in a control/data-flow sense, tests of certain valid configurations may be elided. This is discussed further in Chapter 10.

A note on termination. If it is known that a subcomputation contained within the base program or applied aspects does not terminate, edges exiting the subcomputation can be trimmed from all of the program dependence graphs associated with the program—both original and woven program dependence graphs—during dependence analysis. Known nontermination can either be determined by analysis or provided externally (if desired) where the termination condition is undecidable. Once these edges have been removed from the program dependence graphs, control/data flow dependence analysis can proceed as usual on the modified program dependence graph; the dependence results then apply to the given

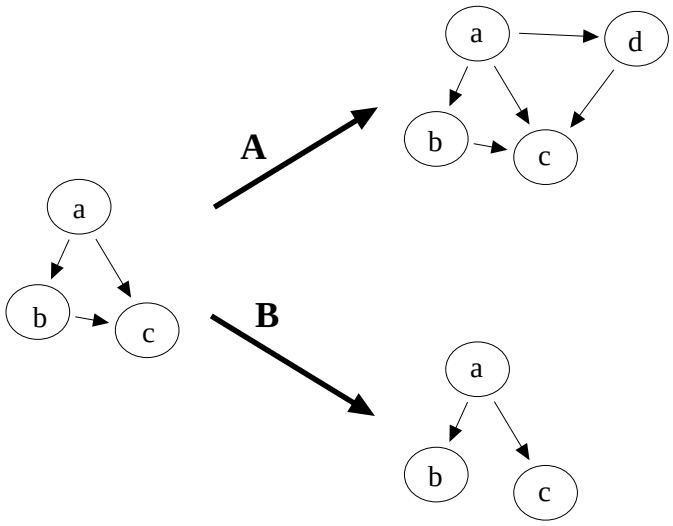


Figure 9.4: The program dependence graph of the code of Figure 9.2; also the program dependence graph of the code after application of A and (separately) B.

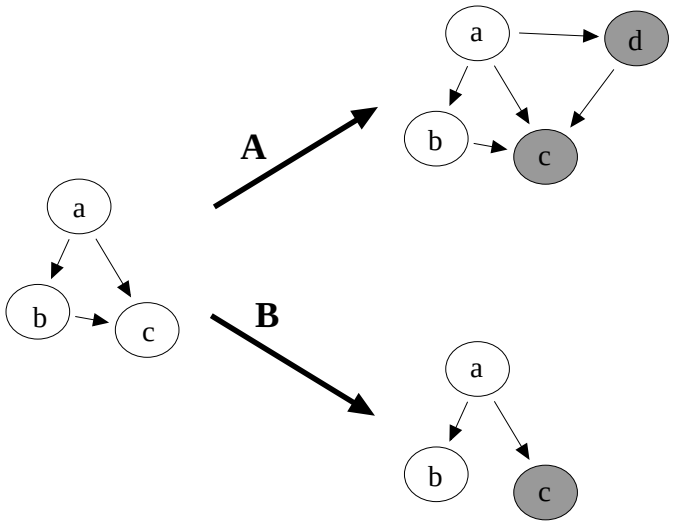


Figure 9.5: Same as Figure 9.4, but affected points ($AP_A(P)$ and $AP_B(P)$) have been shaded.

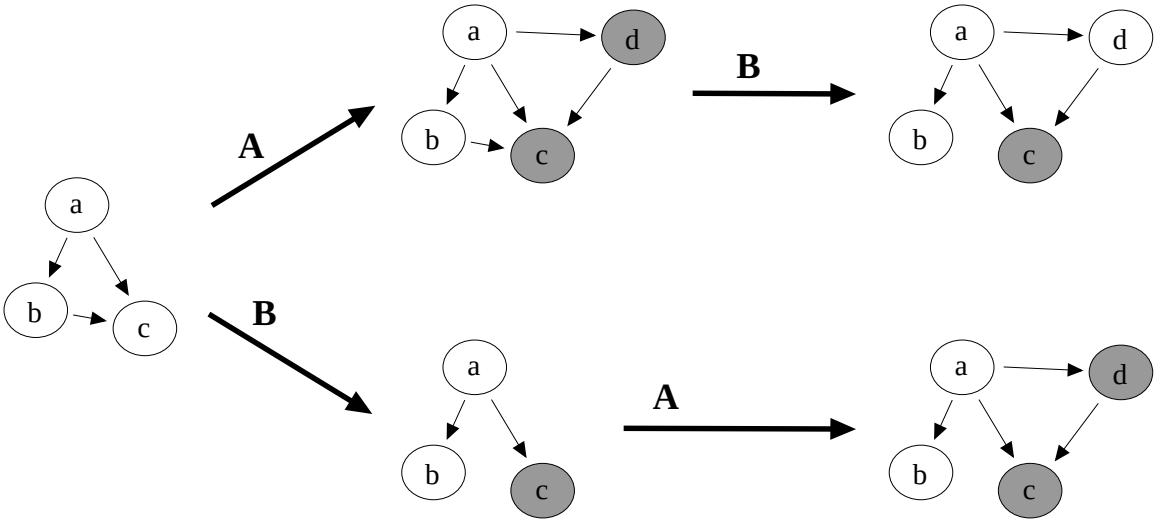


Figure 9.6: Figure 9.5 with application of the other aspect along each path. Here, all affected points—$AP_A(P)$, $AP_B(P)$, $AP_A(B(P))$, and $AP_B(A(P))$—are shaded.

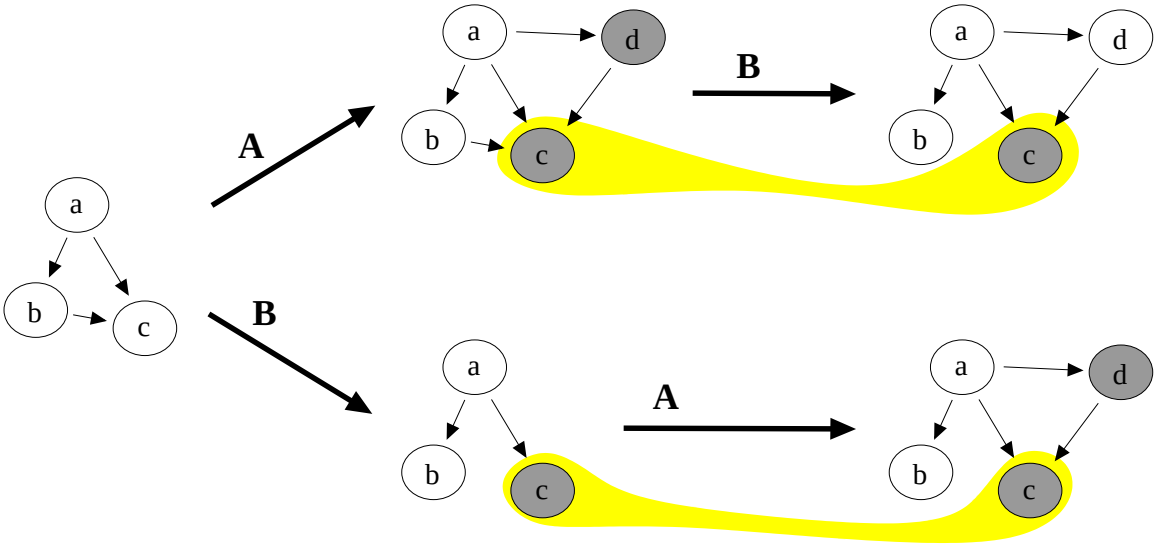


Figure 9.7: Figure 9.6 with a visual diagram of the dependence conditions. In this case aspects A and B are control/data-flow dependent.

aspects over the given base program under the assumption that the subcomputation does not terminate.

9.5 Insensitivity

An aspect A is *insensitive* to an aspect B if the presence of B cannot affect A. Conversely, A is *sensitive* to B if the presence of B can affect A. Such affects of B on A can be *functional* (A is said to be *functionally sensitive to* B or, equivalently, to *have a functional sensitivity to* B), *nonfunctional* (A is said to be *nonfunctionally sensitive to* B or to *have a nonfunctional sensitivity to* B), or both.

9.5.1 Functional sensitivity

Functional sensitivity captures the *functional* behavior of the aspects working on the base program—the specific instructions being executed and the data on which they perform. Functional sensitivity (respectively functional insensitivity) is, then, equivalent to the control/data dependence (respectively control/data independence) of the previous section.

9.5.2 Nonfunctional sensitivity

Nonfunctional sensitivity captures the behavior of aspects outside of their affect on instructions and data. The safety of concurrent access to shared data, resource starvation, and priority inversion [77] are nonfunctional behaviors that can have a dramatic effect on the behavior of other aspects—though two aspects may be functionally insensitive, they may be nonfunctionally sensitive if one starves the other of a needed resource.

Quite often, nonfunctional sensitivity can be observed among two nontrivial aspects. Even if aspects are functionally insensitive, they typically are nonfunctionally sensitive due to bus traffic, page swaps and other I/O, cache line eviction, and other interference.

There are important cases where two nontrivial aspects may be nonfunctionally insensitive, however. Aspects that implement two separate real-time tasks with resource guarantees may fall into this category.

Nonfunctional concerns are, by definition, implicit properties about the program which no one code artifact represents. To undergo analysis, they must be *reified* in some fashion—essentially rendering them a functional concern—and this reified version analyzed. Some approaches to dealing with nonfunctional attributes of programs employ specification and constraint languages to raise the nonfunctional to a functional, checkable level [47].

Therefore, the programmer annotations we require are somewhat different; we do not need a formal specification, we just need some way of determining its dependence relationships. This annotation can be provided in at least four ways, detailed in the next section.

9.5.3 Reification of nonfunctional concerns

The notion of dependence described in Section 9.4 is precisely that described by functional sensitivity; we wish to extend it to cover nonfunctional concern dependence. Effectively, we wish to transform our automated control/data aspect dependence analysis into an aspect sensitivity analysis.

We achieve such extension by translating nonfunctional concerns into functional modifications of the program dependence graphs under analysis. Then the normal determination of dependence of Section 9.4 can proceed. Such modifications, performed by the program's author, can reasonably be realized at these four obvious levels of abstraction (and perhaps others):

1. Modifications to the underlying source code can realize nonfunctional concerns, and the control flow and program dependence graphs can be automatically generated as usual.

2. Prior to weaving, modifications can be made to the intermediate representation used by the aspect weaver. In some cases this may be simpler than modifying the source while retaining the chief advantage of that approach—that the control flow and program dependence graphs are still automatically generated.

3. Direct modifications to the control flow graphs can be made to represent nonfunctional concerns as control flow entities. This must be done multiple times—once for the base program, once for each woven artifact—but may be the preferred way of mapping some nonfunctional concerns to functional concerns.

4. Direct modifications to the program dependence graphs can be made to represent dependencies of nonfunctional concerns. While dependence is explicitly manipulated here, the dependences are between nodes of program dependence graphs, not between aspects. Therefore this method is not begging the question as it might seem to be: aspect dependences are still discovered automatically by the process of Section 9.5 with the added nonfunctional dependences represented in the (modified) program dependence graphs.

Naturally, the graph modifications of the latter two methods can be represented either with the full generality of a source code artifact, or by simple graph rewriting rules. The largest advantage of the last option, setting it singularly apart from the other three, is that no artificial source code (or source code equivalent) need be generated to "trick" the dependence analysis into constructing an adequate program dependence graph. If modifications are performed directly on the program dependence graph, dependences can be explicitly manipulated; if modifications are performed at any of the other three levels of abstraction, code (which is otherwise extraneous) may need to be introduced to effect the right interdependence pattern.

9.6 Conceptual independence

Conceptual independence, new in this work, is defined on the *logical behavior* of aspects. Each aspect is viewed as a collection of invariants that are enforced in the base program. Aspects are said to be *independent* if they are logically *consistent* with others—that is, if the base program can be made to satisfy all of them. This sort of independence can be judged conservatively over all base programs, or over a particular choice of base program.

9.7 AspectJ independence

For AspectJ's aspect model, we typically prefer to add additional constraints on the structural modification of classes:

Definition 9.4 In AspectJ, two aspects A and B are *control/data independent over base code* P if they satisfy four conditions:

$$AP_A(P) \cap AP_B(A(P)) = \emptyset \quad (9.4)$$

$$AP_B(P) \cap AP_A(B(P)) = \emptyset \quad (9.5)$$

A does not (structurally) introduce any of B's requirements. (9.6)

B does not (structurally) introduce any of A's requirements. (9.7)

It should be noted, however, that these extra constraints can be seen as something of an intuitive convenience; the data "dependence" of the (structural) introduction of required code will be enforced by an AspectJ compiler in the same way explicit (code-based) dependences are.

In AspectJ, a correspondence can be made: control/data independence implies weave independence.

Theorem 9.1 In AspectJ, if two aspects A and B are control/data independent over a base program P, then they are weave independent over P.
Proof To be weave-dependent, two aspects must apply to the same join point. However, if two AspectJ aspects apply to the same join point, they will be control/data-flow dependent. Therefore A and B must be weave independent. ∎

This theorem does not necessarily hold for the nominal weaver of Chapter 2, since advice doesn't apply to *all* code; rather, it applies to all base program code, and advice code applied sooner than it.

9.8 Chapter summary

This chapter introduced a taxonomy of dependence, with definitions for *explicit dependence*, *weave dependence*, *control/data-flow dependence*, *sensitivity*, and *conceptual dependence*. It concluded with a look at AspectJ dependence.

Chapter 10

Aspect Dependence Analysis

In this chapter, we perform the program dependence graph-based aspect dependence analysis of Section 9.4 on a variety of aspects both to illustrate the analysis clearly and to determine its potential. For the expository purposes of this chapter, this analysis is performed by hand. We of course do not expect software developers to perform this analysis manually on their own programs; Chapter 11 describes our implementation of the analysis in a modern, widely-used compiler.

10.1 Graph conventions used in this chapter

Control flow graphs use dashed lines to indicate exceptional edges. Program dependence graphs are similar but additionally tag each edge with a "*c*" to indicate a control dependence, a "*d*" to indicate a data dependence, and "*c,d*" to indicate both control and data dependences. As an example, consider Figure 10.4 (on page 131), the program dependence graph of a simple Java program.

In this graph, the control flow is easily seen as a chain of "*c*" relations; many Java statements can throw exceptions, so some statements are control-dependent on the statement that immediately precedes them in the program text. Data dependences are represented by arrows decorated with a "*d*", and conditional branches are indicated by a "`T`" or "`F`" (for *true* and *false*), or in the case of an exceptional path, a dashed line.

10.2 Case study

One of the simplest aspects to understand is a *tracing* aspect. A tracing aspect advises method and constructor call join points and reports them to the user. It can be used in debugging or in understanding program structure. One possible implementation of this aspect is offered in Figure 10.1.[1] Due to its "conceptual" independence from expected application-level logic, this aspect may appear to be completely independent of any base program code it affects and from other aspects implementing different concerns. However, concluding such independence is problematic from a technical point of view.

Consider the simple base program of Figure 10.2. The method `main()` has the control flow graph of Figure 10.3 and the program dependence graph of Figure 10.4. When the tracing aspect is composed with it, however, and the program dependence graph is recomputed, the dependence picture becomes quite different (Figure 10.5): every call has been replaced with a call to an advice method, and therefore most basic blocks are considered to have changed (they are *affected points* as in Chapter 9).

We can tell the compiler that our advice shouldn't throw any exceptions (by writing the call-tracing aspect as shown in Figure 10.6), and we can inline the advice code rather than calling out to methods. This results in the control flow graph shown in Figure 10.7 and the program dependence graph shown in Figure 10.8. From now on we will perform such inlining and use an exception-less approach like this.

Now we can see how multiple pieces of advice could be weaved into the base program and not interfere.

Consider "metric" advice that changes the program to output liters instead of quarts. The control flow graph of the tracing- and metric-advised `CupsToQuarts` program is shown in Figure 10.9 and its program dependence graph in Figure 10.10. The program dependence graph shows that the tracing advice and the metric advice are *control/data-flow independent* over `CupsToQuarts` since there is not a control/data-flow path between them.

Next consider "input-gallon" advice that changes the program so that input is accepted in gallons instead of cups. The control flow graph of the metric- and input-gallon-advised `CupsToQuarts` program is shown in Figure 10.11 and its program dependence graph

[1] This implementation is limited in scope and function, but is intended as a simple example. In particular, this implementation doesn't provide exceptional method exit notification and is not configurable.

in Figure 10.12. This program dependence graph clearly exhibits interference between the two pieces of advice: there are data-flow dependences between the assignment to `quarts` by the input-gallon advice and the use of `quarts` by the metric advice. The metric advice and the input-gallon advice are *not* control/data-independent.

```
aspect Trace {
  /* all calls to methods and constructors */
  pointcut allCalls() : call(* *..*.*(..)) || call(*..*.*(..));

  Object around() : allCalls() && !within(Trace) {
    Logger.log("entering " + thisStaticJoinPoint);

    Object ret = proceed();

    Logger.log("exiting  " + thisStaticJoinPoint);

    return ret;
  }
}
```

Figure 10.1: Code for the simple *call-tracing* aspect of Section 10.2.

```
import java.io.*;

class CupsToQuarts {
  public static void main(String[] args) throws IOException {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

    System.out.println("Enter number of cups:");

    int cups = Integer.parseInt(in.readLine());

    if(cups >= 0) {
      float quarts = cups / 4.0f;

      System.out.println("There are " + quarts + " quarts in " + cups + " cups.");
    } else System.out.println(cups + " is an invalid number of cups.");
  }
}
```

Figure 10.2: A simple base program that converts cups to quarts.

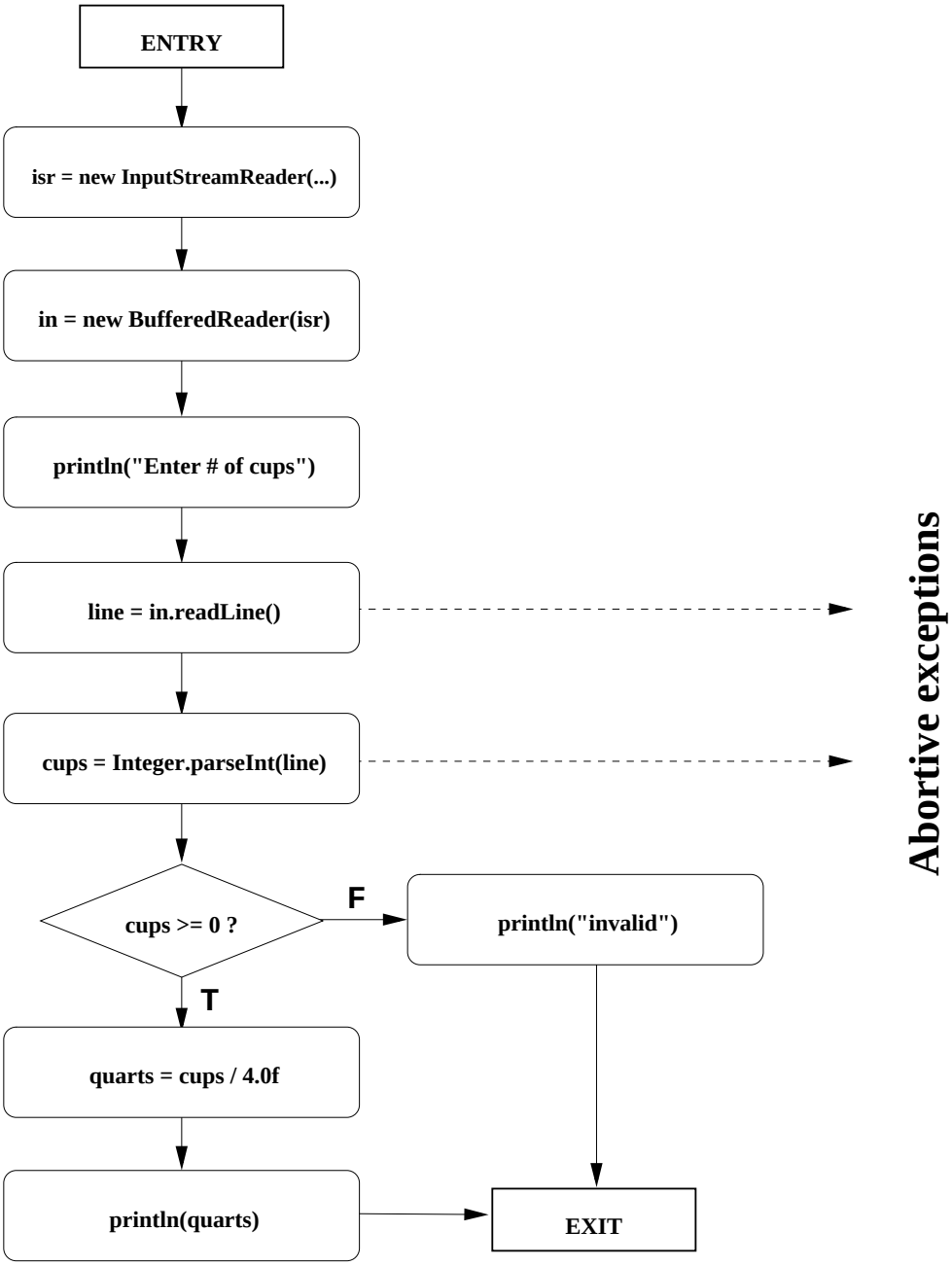


Figure 10.3: The control flow graph of the program in Figure 10.2.

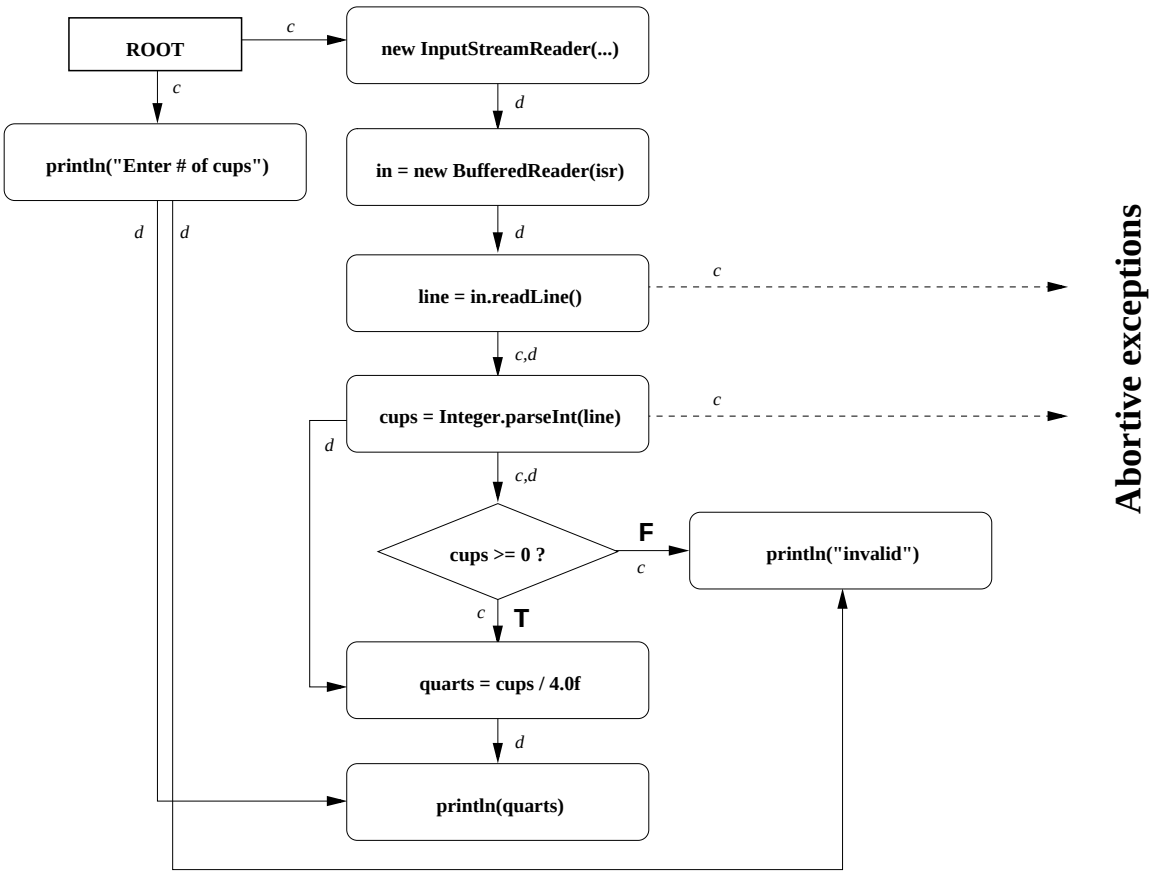


Figure 10.4: The program dependence graph of the program in Figure 10.2. A "*c*" indicates a control dependence, and a "*d*" indicates a data dependence.

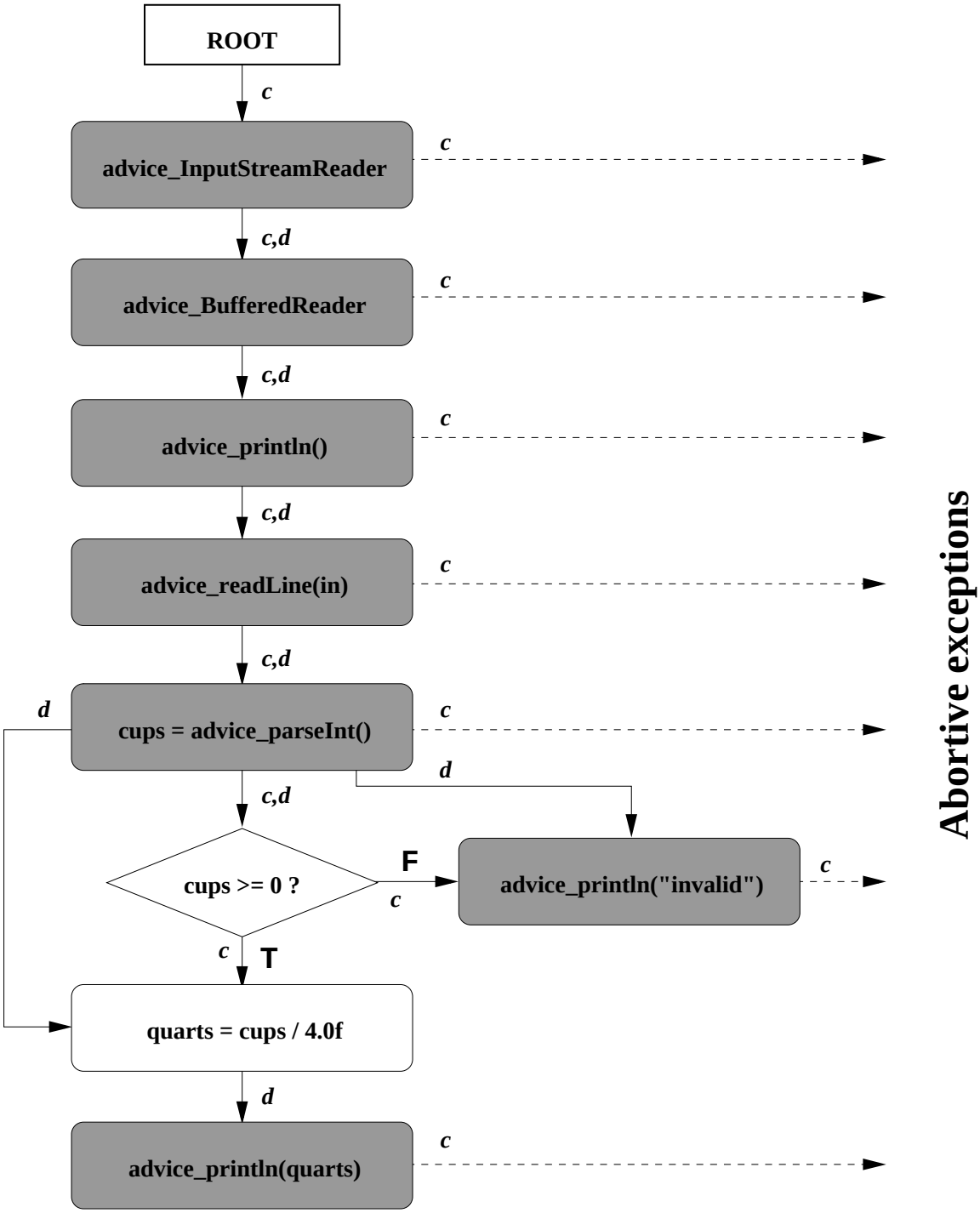


Figure 10.5: The program dependence graph of the program in Figure 10.2 after applying call-tracing advice.

```
aspect Trace { /* tracing aspect with no exception footprint */
  /* all calls to methods and constructors */
  pointcut allCalls() : call(* *..*.*(..)) || call(*..*.*(..));

  Object around() : allCalls() && !within(Trace) {
    try {
      Logger.log("entering " + thisStaticJoinPoint);
    } catch(Throwable t) { /* ignore exceptions caused by advice code */ }

    Object ret = proceed();

    try {
      Logger.log("exiting  " + thisStaticJoinPoint);
    } catch(Throwable t) { /* ignore exceptions caused by advice code */ }

    return ret;
  }
}
```

Figure 10.6: Code for a simple *call-tracing* aspect (Section 10.2) that has no exception footprint.

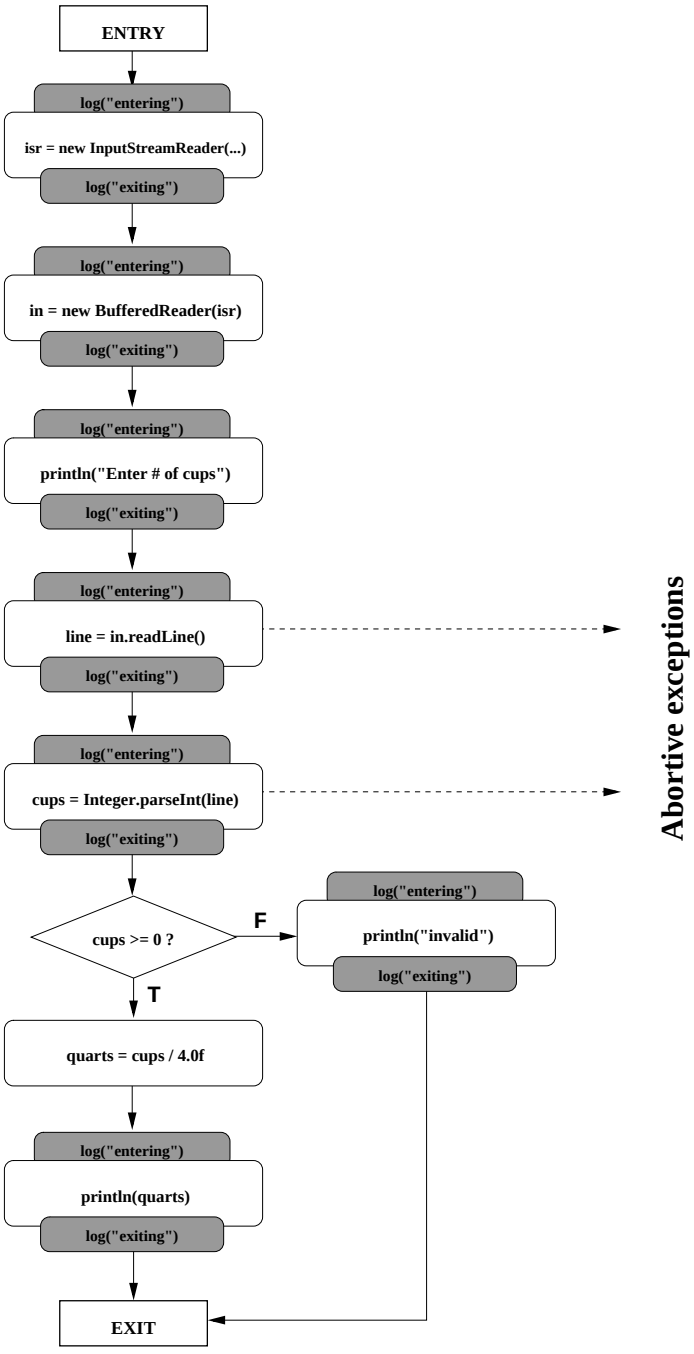


Figure 10.7: The control flow graph of the program in Figure 10.2 after applying *and inlining* call-tracing advice with no exception footprint. Program points affected by the advice are shaded.

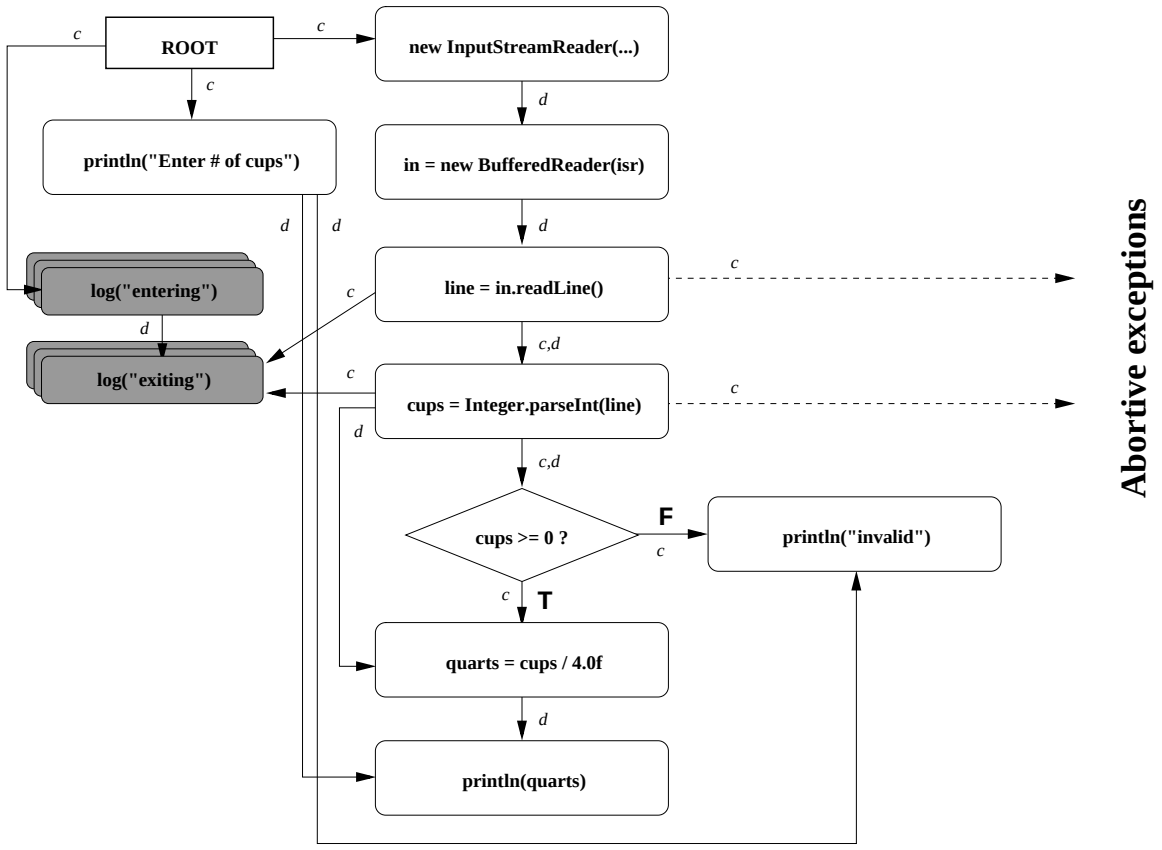


Figure 10.8: The program dependence graph of the program in Figure 10.2 after applying *and inlining* call-tracing advice with no exception footprint.

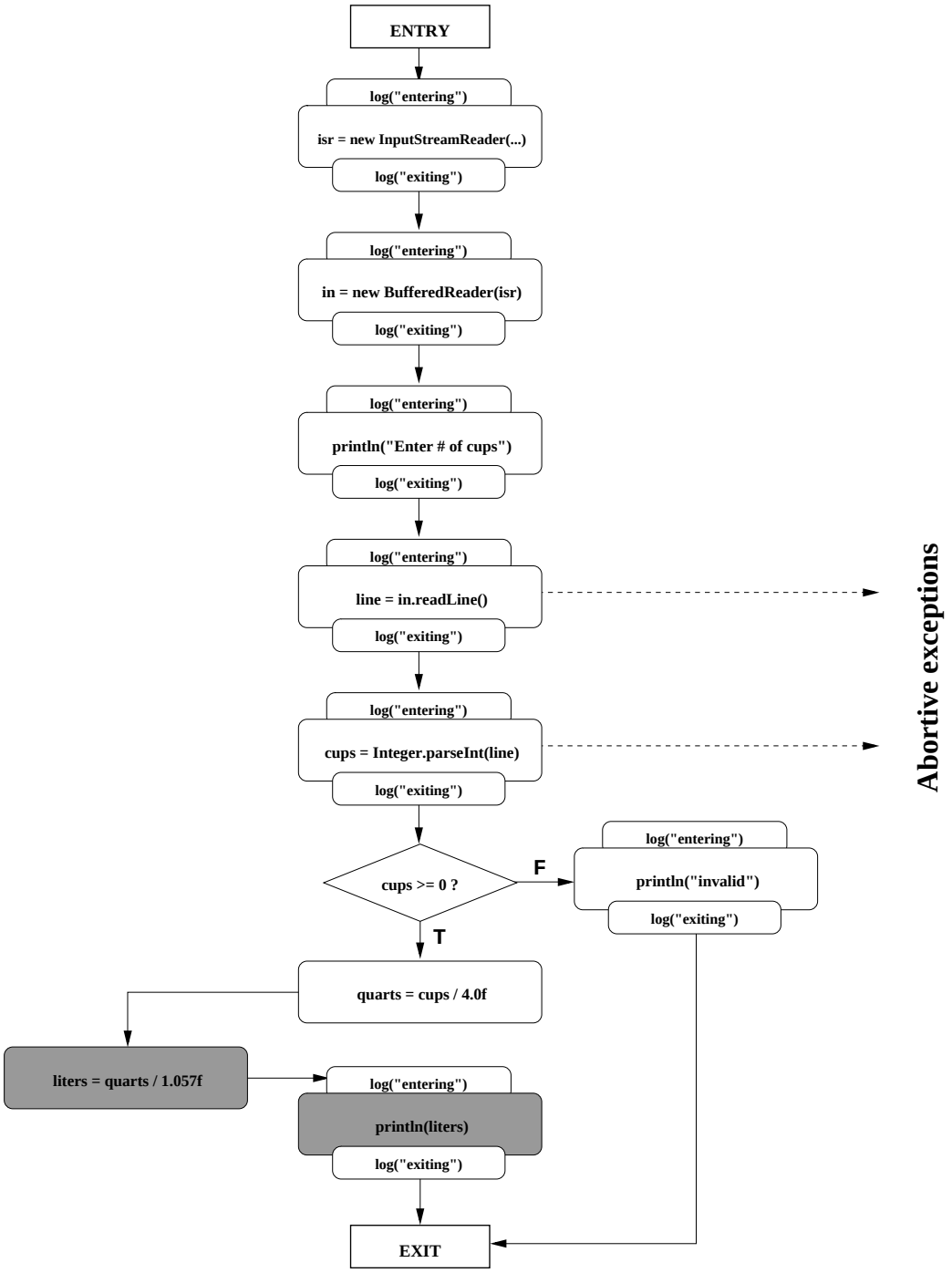


Figure 10.9: The control flow graph of the tracing- and metric-advised program of Figure 10.2 (inlined advice code). The tracing-advised program points are the same as in Figure 10.7; the shaded program points are those affected by the metric advice.

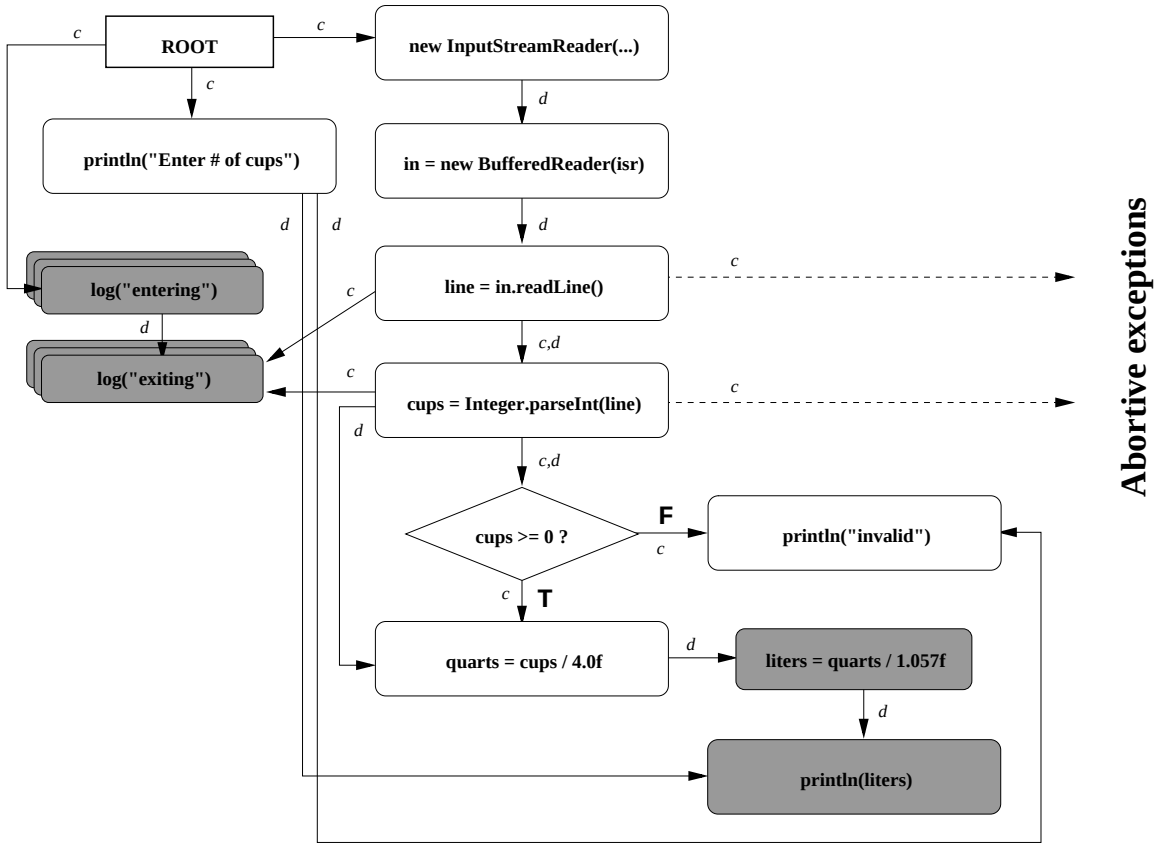


Figure 10.10: The program dependence graph of the tracing- and metric-advised base program in Figure 10.2 (inlined advice code). The shaded "log" boxes at the top left are from the tracing advice; the shaded "liters" boxes at the bottom right are from the metric advice. There is no control/data-flow path from one piece of advice to the other: these pieces of advice are independent.

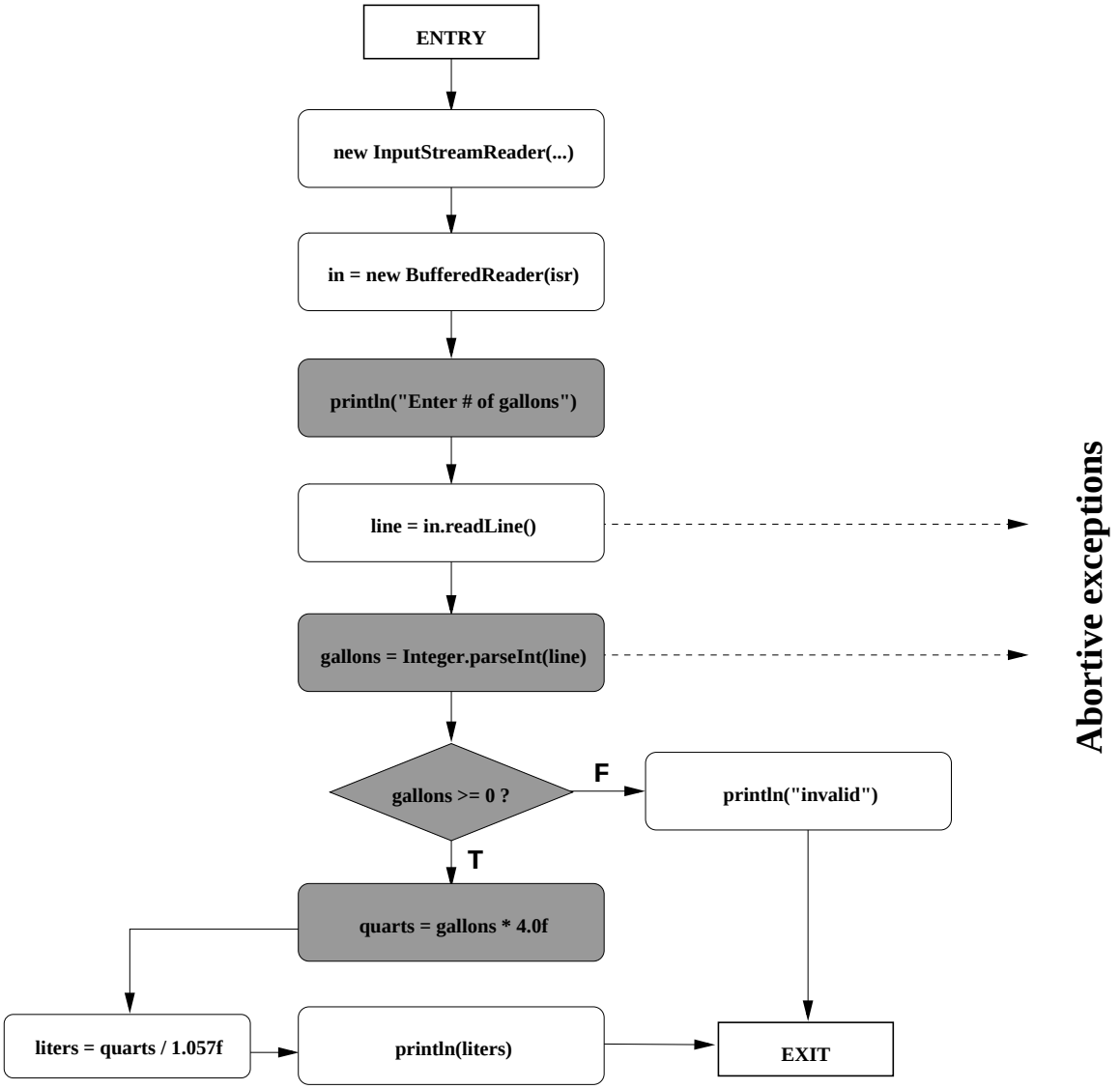


Figure 10.11: The control flow graph of the metric- and input-gallon-advised base program in Figure 10.2 (inlined advice code). Here, the shaded boxes represent those affected by the input-gallon advice.

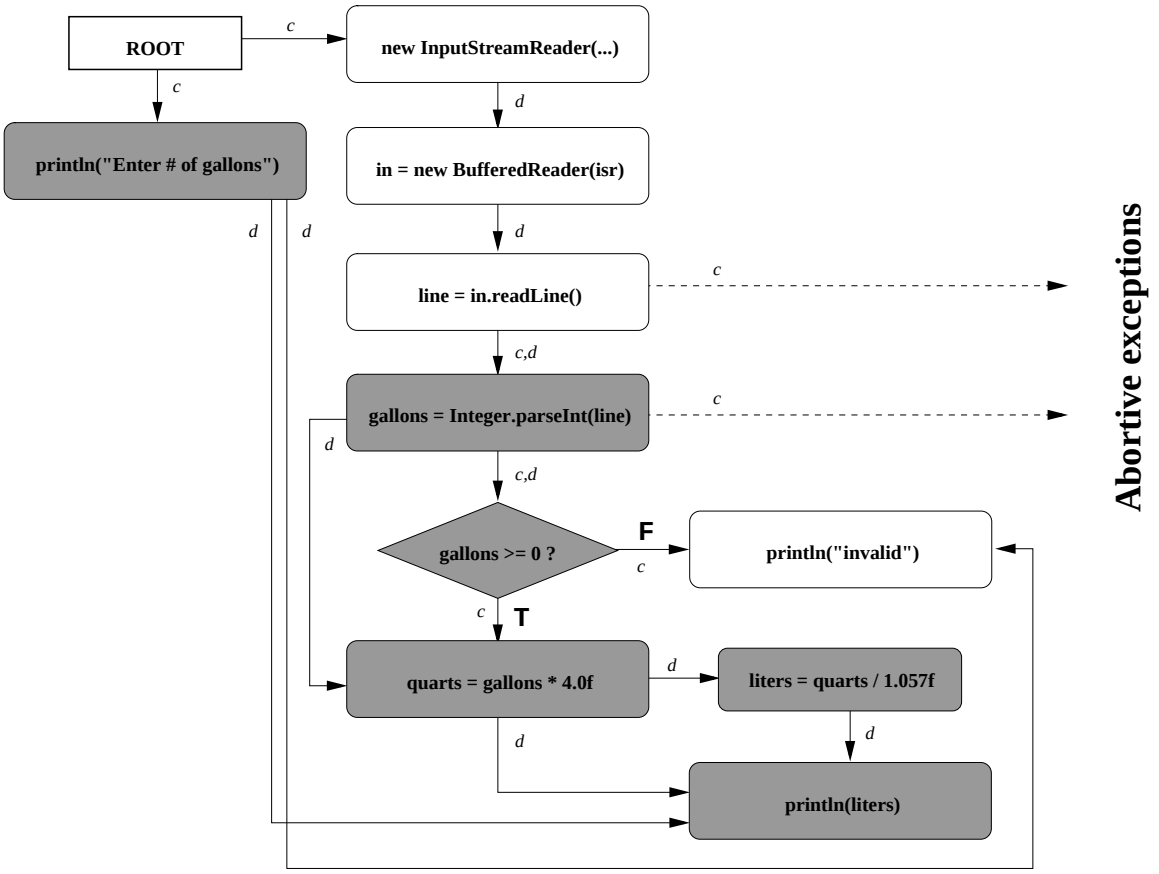


Figure 10.12: The program dependence graph of the metric- and input-gallon-advised base program in Figure 10.2 (inlined advice code). The two rightmost shaded boxes are from the metric advice; the others are from the input-gallon advice. Clearly there is a control/data-flow path from one piece of advice to another: these two pieces of advice are *not* independent.

10.3 FACET

FACET is a Java event channel developed at Washington University [55, 54, 53, 92, 79, 56]. FACET targets real-time, embedded systems and offers an array of features which may be independently omitted to lower the library's footprint; this type of software development is discussed in Chapter 8. These features are enabled at *compile time* rather than runtime: this approach reduces the on-disk footprint of the code itself and avoids the runtime detection and configuration requirements of other approaches (such as plugins for operating systems or web browsers). AspectJ is used to perform the instrumentation for the desired set of features.

With FACET's twenty-three selectable features, there are a considerable number of variations in a configured FACET library installation. Not all 2^{23} feature combinations are permitted, however—some have explicit dependencies on other features, and others are mutually-exclusive. This leads to significant work on the part of FACET maintainers to ensure the proper operation of FACET under all valid configurations. Indeed, this inspired the work of Section 8.3 to determine efficiently the legal combinations of features. FACET's *feature set specification,* encoding these relationships between features, is shown in Figure 10.13.[2]

FACET is designed as a collection of structural base code (a minimally-functional event channel) with the selectable features on the periphery. FACET features use aspects in three ways:

- to implement the feature's advertised functionality;
- to register the feature with the *feature registry* (which handles the explicit dependence and mutually-exclusive relations given in the feature set specification); and
- to add unit tests to the testing framework for the feature.

The use of aspects to implement the feature's advertised functionality can be further subdivided. These uses fall into three categories:

- Features include *advice* to implement the needed FACET behavior for the feature.

[2]The latest FACET release, version 2.2, is used for this work. Statistics, descriptions, and diagrams of FACET relate to this release version.

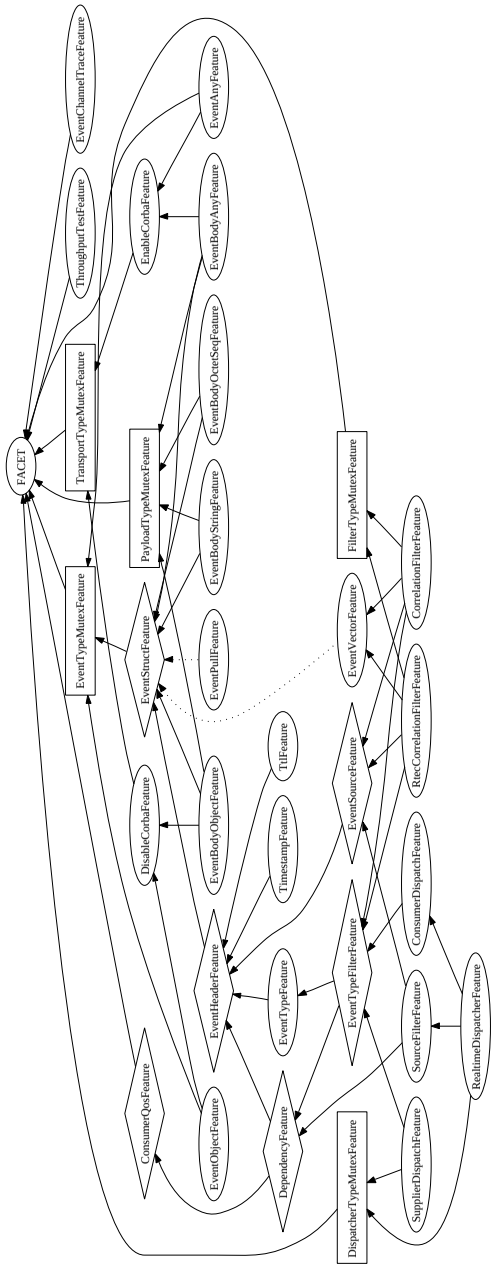


Figure 10.13: FACET's *feature set specification.* Normal features are represented as ovals; *abstract* features are diamonds; and *mutually-exclusive* feature relationships are rectangular. Solid lines indicate a normal dependency. Dotted lines indicate a *uses* relationship, which doesn't require that an abstract feature be realized as an object (in concrete form) at runtime.

- Some features must *introduce* needed fields into other classes with intertype declarations (Section 2.1.5), especially the routing and payload structures (the `EventHeader` and `EventCarrier` classes, respectively).
- Where multiple features apply to the same join point and a specific weaving order is necessary, aspect precedence is declared.

Features do not generally use each other's introduced fields:[3] introductions are either *private* to the introducing aspect or otherwise invisible to the features in other feature groups. So the main sites of aspect control/data interaction are:

- in the feature registry itself, and
- in the implementation behavior.

We consider each of these in turn.

10.3.1 The feature registry

All features explicitly depend upon the feature registrar, and the registrar itself depends upon the base class of all features. The registration mechanism for a feature consists of:

- the registrar's `AutoRegisterAspect`, which automatically registers all subaspects seen by the aspect weaver (see Figure 10.14);
- An interface that declares dependencies of the feature in its *extends* clause (see Figure 10.15);
- an aspect nested inside the interface that extends the registrar's `AutoRegisterAspect` and registers the feature (see Figure 10.15); and
- the registrar's `FeatureRegistry` class, which ultimately handles the feature registration and ensures that there are no feature combination violations (such as two mutually-exclusive features both being enabled).

It is important to emphasize that FACET's feature registry is decoupled from the implementation of its features. That is, a feature can still be active even though it fails

[3] Features often *do,* however, use fields introduced by their abstract super-features. See Section 2.1.5 for a discussion of *introductions* via intertype declarations.

to register. Registration ensures that proper initialization occurs and that explicit dependencies are satisfied; by that time the aspect weaver has already ensured that the feature's behavior is realized in the FACET library.

Analysis. When `FeatureRegistry.buildGraph()` is called during FACET startup, all the subaspects of `AutoRegisterAspect` are registered. This leads to several aspect control/data-dependences. First, any feature registration aspect (or the attempt to call into one) could potentially throw an exception and cause program abortion, though the other registrations would continue. Second (and even if exception behavior is ignored), there is still a control/data-dependence between two registration aspects as they ultimately access the same data structure in the feature registry. So there are control/data dependences between them.

```
package edu.wustl.doc.facet.feature;

/**
 * This aspect is intended to be inherited by any aspect that
 * needs to be registered in the FeatureRegistry.
 */

public abstract aspect AutoRegisterAspect {

        abstract protected void register (FeatureRegistry fr);

        private pointcut registry (FeatureRegistry fr) :
                execution (void FeatureRegistry.buildGraph ())
                && target (fr);

        after (FeatureRegistry fr) : registry (fr)
        {
                register (fr);
        }
}
```

Figure 10.14: FACET's `AutoRegisterAspect`, which registers the *time-to-live* filtering feature.

10.3.2 Payload contention

The *time-to-live* feature's main behavioral aspect is listed in Figure 10.16. This aspect provides public accessors `getTtl()` and `setTtl()` on `EventHeader`s and implements time-to-live functionality as a piece of `around()` advice.

```
package edu.wustl.doc.facet.feature_ttl;

import edu.wustl.doc.facet.feature.AutoRegisterAspect;
import edu.wustl.doc.facet.feature.FeatureRegistry;
import edu.wustl.doc.facet.feature_eventheader.EventHeaderFeature;

public interface TtlFeature extends EventHeaderFeature {

    static aspect Register extends AutoRegisterAspect {
        protected void register(FeatureRegistry fr) {
            fr.registerFeature(TtlFeature.class);
        }
    }
}
```

Figure 10.15: FACET's `TtlFeature` interface, with a nested aspect that registers the *time-to-live* filtering feature.

Analysis. Other "event header features" have advice on very similar pointcuts (compare to `TimeStampAspect` in Figure 10.17), deal with the same event channel structures, and therefore exhibit contention on these payload objects. The *time-to-live* feature is especially problematic for control/data dependence, since it proceeds conditionally.

10.4 Chapter summary

We have investigated control/data dependence in several aspects. As a general rule, features that inspect but don't modify state are easily control/data-independent of each other, while aspectual filters and aspects that involve modifications of state (like in FACET) often exhibit control/data-dependence on other aspects.

Control/data-independent, state-inspecting aspects are not necessarily as rare as they may seem at first; many important aspects are of this type. Many debugging aspects, including the tracing aspect of this chapter, are often control/data-independent of other aspects. Aspects that update a user interface based on current program state are also, and even aspects that manage sets of executing tasks or manage thread synchronization can fit into this category.

```
package edu.wustl.doc.facet.feature_ttl;

import edu.wustl.doc.facet.*;
import edu.wustl.doc.facet.EventChannelAdmin.*;
import edu.wustl.doc.facet.EventComm.*;

aspect TtlAspect {

        //
        // Update the time to live for the event.
        // If the TTL is still ok, return true.
        //
        boolean update_ttl (Event event)
        {
                event.getHeader().setTtl (event.getHeader ().getTtl () - 1);
                return event.getHeader().getTtl () >= 0;
        }

        //
        // Update the TTL, and possibly drop the event if it gets too low
        //
        void around (EventCarrier ec) :
                call (void EventChannelImpl.pushEvent (EventCarrier))
                && args (ec)
        {
                if (update_ttl (ec.getEvent ()))
                        proceed (ec);
        }

        //
        // Add appropriate accessors
        //
        public long EventHeader.getTtl ()
        {
                return this.ttl;
        }

        public void EventHeader.setTtl (long l)
        {
                this.ttl = l;
        }

}
```

Figure 10.16: FACET's `TtlAspect` aspect, with advice to update the *time-to-live* field and only proceed if the time has not expired.

```
package edu.wustl.doc.facet.feature_timestamp;

import edu.wustl.doc.facet.*;
import edu.wustl.doc.facet.EventComm.*;

aspect TimestampAspect {

        //
        // Mark a timestamp on the event as soon as it arrives
        //
        before (EventCarrier ec) :
                execution (void EventChannelImpl.pushEvent(EventCarrier))
                && args(ec)
        {
                Event event = ec.getEvent();
                event.getHeader ().setTimestamp (System.currentTimeMillis ());
        }

        //
        // Add appropriate accessors
        //

        public long EventHeader.getTimestamp ()
        {
                return this.timestamp;
        }

        public void EventHeader.setTimestamp (long tstamp)
        {
                this.timestamp = tstamp;
        }
}
```

Figure 10.17: FACET's `TimeStampAspect` aspect, with advice to assign the *timestamp* field when an event is pushed.

Chapter 11

Implementation

We implemented the dependence analysis of Section 9.4 by modifying the *GNU Compiler Collection* (GCC) version 4.1.1. GCC is a compiler suite widely for the compilation of C [64], C++ [99], Java [50], and other languages. GCC targets a wide array of processor families.

Precisely because it is so practical a compiler, we chose GCC as a research platform for implementing our work. Our changes can in principle be used to support our dependence analysis in different feature specification languages, and regardless of the architecture for which the code is being compiled. Optimizations enabled by our can similarly be applied to different languages and when compiling for a range of processor families. With full standard language support, a large development and larger user community, regular bug-fixes and feature enhancements, and debugging and profiling support, we feel GCC's benefits for research outweigh it's drawbacks.

The following sections detail our implementation in GCC of our dependence analysis. Section 11.1 logs the modifications we made to the compiler; specifically, Section 11.1.1 details the front-end modifications, Section 11.1.2 details the middle-end modifications, and Section 11.1.3 discusses consequences of our design decisions. Section 11.2 concludes and summarizes these implementation details.

11.1 Modifications to the compiler

GCC is designed to be highly modular to facilitate extension. New language *front-ends* can be incorporated to support additional programming languages, new *middle-end* analysis and transformation passes can be written in a language-independent and target-independent manner, and support for new language-independent processor and processor family *back-ends* can be provided. The modular design keeps these three "ends" from having undue dependencies on each other.

When a GCC front-end initially parses a program, it generates a tree representation of the program called GENERIC. Initially this representation includes language-specific constructs and is therefore not language-independent, but prior to passing the tree representation off to the middle-end, the representation is lowered to a simpler and language-independent tree representation called GIMPLE. GIMPLE trees are transformed in a variety of ways and are ultimately themselves lowered to an even simpler (and somewhat target-specific) intermediate representation used to produce the final assembly code output.

The modifications to the compiler in support of our dependence analysis work include modifications to its Java front-end and the addition of a middle-end compiler pass that operates on GIMPLE trees. Details of these modifications are described in the following sections.

11.1.1 Front-end modifications

GCC's stock implementation of Java aims for Java language compliance and does not support aspect-oriented constructs. We extended GCC's Java front-end to support a subset of AspectJ features for this work.

Language extensions

AspectJ-style aspect declarations are supported. However, aspects are all singletons: "per..." declarations are unsupported.

AspectJ-style pointcuts are supported with static and dynamic join point specifiers. The join point specifiers `cflow()`, `cflowbelow()`, and `handlers()` are not currently supported.

AspectJ-style advice is supported in *before*, *after*, and *around* varieties.

Aspect weaver

Once all the aspect definitions are read, they must be woven into the base program. We implemented an aspect weaver in GCC's Java front-end to perform this function. The weaver is flexible and can operate with the behavior of the nominal weaver of Section 2.2.1 or the AspectJ weaver as discussed in Section 2.2.2.

A note on implementation omissions

Not all AspectJ language constructs are supported. In particular, highly dynamic features of AspectJ (such as the family of `cflow` constructs) are unsupported. We judged that these features could be safely omitted for implementation simplicity, as it is our intent to determine the utility of aspect dependence analysis in a setting more broad than AspectJ. Many AspectJ programs do not use these features, and our set of AspectJ programs for experimentation use only our implemented subset of AspectJ features. Certainly a full AspectJ implementation in GCC would need to include these useful constructs.

Further, our implementation is in its infancy and should not at present be considered a well-tested aspect language implementation for serious use.

11.1.2 Middle-end modifications

To perform the aspect dependence analysis of Section 9.4, we implemented a middle-end GCC pass.

GCC detects data dependences for a variety of reasons in the middle-end, and it detects control dependences for a sophisticated *dead code elimination* pass. These analyses

are performed on SSA-transformed GIMPLE trees. The program dependence graph needed for aspect dependence analysis is merely the unification of these two dependence relations.

11.1.3 Consequences of implementation design

Because we implemented the weaver in the front-end, it cannot currently be used to support aspect-like facilities in languages other than our modified Java front-end. It would be possible to move our weaver from operating on Java-specific GENERIC trees to operating on language-independent, middle-end GIMPLE trees, but such is beyond the scope of this work. At the least it would require extending GIMPLE with notions of pointcuts, aspects, and intertype declarations. With such middle-end constructs, supporting aspect languages in additional GCC front-ends becomes fully a language matter—defining syntax for the aspect language forms and parsing them into this extended GIMPLE representation.

11.2 Chapter summary

We have implemented the dependence analysis introduced in Section 9.4 and studied in Chapter 10 inside GCC. This required adding aspect support to GCC. Weaving is performed early in compilation, so currently functions only for the Java language front-end of GCC; this limitation could be fixed in future.

Part III

Context of this Work

Someday we'll look back on this moment and plow into a parked car.

– *Evan Davis*

Chapter 12

Related Work

Related work falls into several categories corresponding to the different facets of this dissertation work. A related work discussion is present in many chapters; in particular, Chapters 4, 5, and 6 discuss work related to their specific topic areas. This chapter discusses related work in the areas of advanced separation of concerns, system aspects, and program dependence analysis.

12.1 Advanced separation of concerns

The general field of *Advanced Separation of Concerns* contains a variety of different technologies; most directly relevant to this dissertation are *aspect-oriented programming* and *multi-dimensional separation of concerns*.

12.1.1 Aspect-oriented programming

Aspect-oriented programming (discussed at length in Chapter 2) has received a lot of attention in the research community over the past ten years or more. It evolved from work on a metaobject protocol for Common Lisp [65], and an early formulation appears in [67].

12.1.2 Multi-dimensional separation of concerns

Multi-dimensional separation of concerns (MDSoC) approaches [102] focus more heavily on the *composition* of software features more than aspect-oriented approaches do. The same kinds of techniques are available in the two approaches, but the philosophy is somewhat different: software written as a subsettable collection of features (as described in Chapter 8) is a natural match for MDSoC design.

MDSoC tools, including HyperJ [57], allow you to write multiple views of the same class. These views are then composed, resulting in a final class with all of the functionality of each of the views. Where views conflict, resolution behavior can be specified, composing the contributions of the views or allowing one to dominate another.

12.1.3 Feature-oriented programming

Feature-Oriented Programming [17] is a somewhat less specific term that generally describes the style of programming that is the focus in this dissertation: an application is designed with a central core and a collection of modular features that are composed onto that core.

This can be achieved in a modelling tool [70], in an aspect-oriented language [67], in a language supporting multi-dimensional separation of concerns [102], and in other ways. Certain tools, like those of the AHEAD tool suite [18], are expressly intended for this purpose.

12.1.4 Relationship to this work

This dissertation identifies several problems and aspect-oriented solutions to them (Chapters 3–7). However, these solutions cannot be satisfactorily realized using AspectJ. This failure is analyzed and solutions to the problem suggested in Chapter 7.

This dissertation also introduces an aspect dependence analysis, described in aspect-oriented terms, but specifically motivated by and intended for feature-oriented software.

12.2 System aspects

Most uses of aspects are for application-level, rather than system-level, concerns. System-level aspects are application-agnostic and perform system functions—for example, garbage collection, task scheduling, or hardware support—in a manner transparent to the application.

Recent work has looked at implementing aspect-oriented design patterns [52]. Design patterns are not entirely system-level, but are similar in that they are often application-agnostic and reusable. However, they are also tied to the design of the application.

Aspects have been suggested for system security [110, 114], including for error checking, buffer overflow protection, and secure socket wrapping. Aspect socket wrapping is a common theme, on which we have prior work as well [29]. These concerns are on the level of the system.

System-level aspects often aim to change the semantics of the language or the runtime system; they thus represent a form of language extensibility. OpenJava [103] and OpenC++ [31] are examples of extensible languages. They use metaobject protocols [65] (precursors to aspect-oriented programming) to provide that extensibility.

12.3 Program dependence analysis

Program dependence analysis and the program dependence graph [46, 89] have been used widely in compiler analysis and optimization. They are used for code motion [68, 41], loop optimizations [14], and other optimizations [84].

This work applies program dependence to a new area: finding independence of modularly-specified features to be automatically composed with ("weaved into") a software core. With this dependence information, tests can be elided that otherwise may need to be executed.

Chapter 13

Conclusions

We start by restating the key technical contributions of this work, indicating from which chapters the various contributions come. We reiterate why these contributions are key. We then draw various conclusions about the direction of aspect-oriented language design from this work as a whole.

13.1 Key technical contributions

This dissertation makes several contributions. Roughly, they fall into five categories:

- system aspects;
- taxonomy of dependence relations;
- determining valid configurations;
- feature independence analysis; and
- future directions for aspect languages.

These contribution categories are each described below.

13.1.1 System aspects

This dissertation provides three examples of "system aspects"—aspectual mechanisms at the system-level and effect language and runtime concerns rather than application concerns. These system aspects are individual pieces of work that stand on their own. They are not simply pedagogical examples: Chapter 4 describes a metaprogramming mechanism to schedule task sets at compile time; Chapter 5 develops an automated memory management system for Java based on reference counting; and Chapter 6 describes software support for a hardware garbage collector.

These three chapters, together with Chapter 7, observe where these system aspects don't have an acceptable aspectual formulation today, and indicate what features a future aspect language could include to solve this problem.

13.1.2 Taxonomy of dependence relations

Chapter 9 introduces a taxonomy of dependence and independence relations for aspects, separating out different notions of dependence. The term "dependence" is used inconsistently in the literature.

13.1.3 Determining valid configurations

Chapter 8 proposes an algorithm for determining *valid configurations* based on a static feature set specification. This algorithm is proven correct and is proven to have an asymptotic computational complexity at least as good as any other algorithm for the purpose.

13.1.4 Feature independence analysis

Chapters 9 and 10 investigate a method of static analysis to determine a strong form of independence of aspects. Chapter 10 discusses benefits gained by programmers for aspects determined to be independent in this fashion.

Our implementation of this static analysis in a production compiler is described in Chapter 11.

13.1.5 Future directions for aspect languages

Chapter 7 indicates possible future directions for aspect languages to bring system aspects under the aspect purview. In particular, a generalized, extensible join point model!generalized and compile-time computation facilities are motivated.

13.2 This work in context

The preceding are key contributions.

System aspects, while studied before [110, 114], haven't been a focus for the aspect-oriented language community. This has had the tendency to drive aspect language development away from support for system-level concerns and away from fully extensible, flexible languages and runtime systems. This dissertation focuses on system aspects and the problem with implementing them with current tools. It makes observations that may be useful for future aspect language designers.

"Dependence" is used inconsistently in the aspect literature; this dissertation attempts to separate different notions associated to the term, discuss them directly, and provide them different names.

Subsettable software can have a significant testing burden. Insights made in this work can (1) reduce that burden for programmers of large software product lines, and (2) allow developers to enumerate their valid configurations efficiently.

13.3 Unwoven aspect analysis

Users of aspect-oriented languages enjoy the benefits of a higher level of abstraction for the software they write. They may be better able to separate the concerns of their software, and without liability: often aspect-oriented features are supported as an extension to a host language, as AspectJ is to Java, so programmers aren't trapped by the paradigm; instead they can choose not to apply aspect-oriented features when they deem those features unsuitable.

Aspect developers can benefit from the observations made in Chapter 7, which indicates improvements that future aspect languages could make to handle efficiently the kinds of systemic concerns of Part I of this dissertation.

Broadly speaking, Part I contributes an aspect usability study and Part II studies of aspect analysis; but usability and analyzability aren't separate, competing goals. Rather, the observations in Chapter 7 could improve aspect analysis as well. When programs are specified at a higher level, aspect compilers can do more to understand programmer intent, perform more accurate analyses, and offer better optimizations.

Appendix A

Source Listings

This appendix provides various source code listings for programs, functions, classes, and other artifacts referred to in the dissertation text. The listings are as follows:

Probe.java

```
package autoscope.probe;

import java.io.*;
import java.util.*;

/**
 * Contains some important pointcuts for aspect probes in this
 * package.
 */

abstract aspect Probe {
    /** A pointcut to exclude Probe aspects from analyzing
     * themselves */
    pointcut withinUs():
        within(autoscope.probe..*+) ||
        within(autoscope.runtime..*+);

    /** Convenience pointcut for method join points that interest us */
    pointcut methodExecution():
        !withinUs() && execution(* *.*(..));

    /** Convenience pointcut for constructor join points that interest
     * us */
    pointcut constructorCall():
        !withinUs() && call(*.new(..));

    /** A pointcut to select the execution of the <code>main()</code>
     * routine. */
    pointcut mainCut(String[] cmdline) :
        execution(public static void main(String[])) &&
        args(cmdline);

    /** The PrintWriter to which to send output. */
    protected PrintWriter out;

    /** Must be overridden by subaspects to specify the output file
     * extension.
     * @return the filename extension to use for the out
     * file.  The base part of the name is determined by the class
     * that executes the <code>main()</code> routine. */
    abstract protected String outExtension();

    /** A member aspect designed to cut around <code>main()</code>
     * before any Probe subaspect does.  Only thing is, the subaspects
     * of Probe have to have member aspects that dominate
     * Probe.MainCutter and force them (the subaspects of Probe) to be
     * loaded, adding to theProbes a reference to themself.  Really
     * ugly. */
```

```
static aspect MainCutter {
    declare precedence: MainCutter, Probe+;

    /** A Vector of Probe aspects.  This is added to with after(Probe)
     * advice on Probe initialization. */
    protected Vector theProbes = new Vector();
    /** Advice to grab onto <code>MakeMyPresenceKnown</code>
     * aspects in Probe subaspects.  It adds Probe
     * instances to the Vector theProbes as they are initialized. */
    after(Probe p): this(p) && initialization(Probe.new(..)) {
        theProbes.add(p);
    }
    /** Advice to set up the log file output PrintWriter (Probe.out).  The
     * filename extension is retrieved from Probe.outExtension(). */
    void around(String[] cmdline): mainCut(cmdline) {
        String packagedir = "";
        try {
            packagedir =
                thisJoinPointStaticPart.getSignature()
                .getDeclaringType().getPackage().getName()
                .replace('.','/') + '/';
        } catch(RuntimeException _) {}
        String filename =
            thisJoinPointStaticPart.getSourceLocation().getFileName();
        String reffilename = packagedir +
            filename.substring(0, filename.length() - 4);
        String extension = null;
        try {
            synchronized(theProbes) {
                for(Enumeration en = theProbes.elements();
                        en.hasMoreElements();) {
                    Probe p = (Probe)en.nextElement();
                    extension = p.outExtension();
                    if(extension == null)
                        throw new NullPointerException
                            ("The String returned by "
                             + p.getClass().getName() +
                             ".outExtension() cannot be null");
                    p.out = new PrintWriter
                        (new FileWriter(reffilename + extension));
                }
            }

            try {
                proceed(cmdline);
            } catch(Throwable e) {
                System.out.println("main() threw a "
                                   + e.getClass().getName());
                e.printStackTrace();
            }
```

```
                Thread[] thds = new Thread[32];
                Thread me = Thread.currentThread();
                ThreadGroup myGroup = me.getThreadGroup();
                int t;

                do {
                    t = myGroup.enumerate(thds, true);
                    try {
                        for(int i = 0; i < t; ++i)
                            if(thds[t] != null && thds[t] != me)
                                thds[t].join();
                    } catch(InterruptedException _) {}
                } while(t > 1);

                synchronized(theProbes) {
                    for(Enumeration en = theProbes.elements();
                            en.hasMoreElements();)
                        ((Probe)en.nextElement()).out.close();
                }
            } catch(IOException e) {
                System.err.println("while trying to work with "
                                   + reffilename + extension + ": " + e);
                e.printStackTrace();
            }
        }
    }
}
```

LivenessProbe.java

```
package autoscope.probe;

import java.util.*;

import org.aspectj.lang.*;
import org.aspectj.lang.reflect.SourceLocation;

/**
 * An abstract aspect to serve as the parent for aspect probes that
 * detect when objects become collectible.  This aspect contains no
 * liveness-probing implementation, but does handle some
 * implementation-independent stuff.
 */

abstract aspect LivenessProbe extends Probe {
    /** Returns our preferred filename extension "liv".  Returns the
     * String "liv", signalling that we want an output file with a
     * ".liv" extension. */
    protected String outExtension() { return "liv"; }

    // Ideally, we'd do this...
    //   private long Thread.frame = 0;
    // but for now, we have to do this...  :(

    /** The ThreadMap for the probed program. */
    protected final ThreadMap threads = new ThreadMap();

    /** This static field enforces the rule that you may not have more
     * than one LivenessProbe probe at once. */
    private static boolean alreadyConstructed;
    {
        if(alreadyConstructed)
            throw new ProbeError("Only one LivenessProbe probe "
                                 + "may be used at once.");
        alreadyConstructed = true;
    }

    protected pointcut frameAction(): constructorCall() || methodExecution();

    protected Hashtable newLocHash = new Hashtable();
    protected Hashtable newSiteThreadHash = new Hashtable();
    protected String getSrcLoc(Object o) throws NoSuchNewSiteException {
        SourceLocation loc = (SourceLocation)newLocHash.get(o);

        if(loc == null)
            throw new NoSuchNewSiteException();

        return o.getClass().getName() + "(" + loc.getFileName()
```

```
            + "<" + loc.getLine() + ":" + loc.getColumn() + ">)";
    }
    static protected class NoSuchNewSiteException extends Exception {}

    before(Object o): this(o) && execution(*.new(..)) && !withinUs() {
        if(newLocHash.get(o) == null) {
            /* this can be executed without a constructorCall() if this object
               is constructed reflectively, and x will be null */
            Object x = newSiteThreadHash.get(Thread.currentThread());
            if(x != null)
                newLocHash.put(o, x);
        }
    }

    Object around(): constructorCall() {
        newSiteThreadHash.put(Thread.currentThread(),
                              thisJoinPointStaticPart.getSourceLocation());
        Object o = proceed();
        if(newLocHash.get(o) == null)
            newLocHash.put(o, thisJoinPointStaticPart.getSourceLocation());
        Thread thr = Thread.currentThread();
        try {
            out.println("::new/" + (threads.getFrame(thr) - 1) + " " +
                        oid(o) + " " + getSrcLoc(o));
        } catch(NoSuchNewSiteException e) {
            e.printStackTrace();
        }
        return o;
    }

    Object around(): frameAction() {
        enterScope(thisJoinPointStaticPart);
        Object retval = proceed();
        exitScope(thisJoinPointStaticPart);
        return retval;
    }

    protected void enterScope(JoinPoint.StaticPart jp) {
        out.println(">>enter/" + threads.getFrame(Thread.currentThread()) +
                    " " + jp);
    }

    protected void exitScope(JoinPoint.StaticPart jp) {
        out.println("<<exit/" + threads.getFrame(Thread.currentThread()) +
                    " " + jp);
    }
}
```

DeathProbe.java

```
package autoscope.probe;

import java.lang.ref.*;
import java.util.*;

import org.aspectj.lang.reflect.SourceLocation;

/**
 * An aspect to detect when objects become collectible.  This aspect
 * uses a ReferenceQueue implementation.
 */

aspect DeathProbe extends LivenessProbe {
    /** A ReferenceQueue to use for object WeakReferences becoming
     * dead */
    protected final ReferenceQueue Q = new ReferenceQueue();

    /** A Vector to keep track of currently live object WeakReferences
     * in the system.  If we didn't keep a strong reference to these
     * WeakReferences, they would be collected and not show up on our
     * queue. */
    protected final Vector refs = new Vector();

    // Threads to Vectors of dead references
    protected final Hashtable deadRefHash = new Hashtable();

    /** Advice to capture a frame push or pop; frame counts are
     * maintained and collected objects are tracked. */
    Object around(): frameAction() {
        Thread thr = Thread.currentThread();
        Object retval = null;

        try {
            retval = proceed();
        } finally {
            DeathProbeReference r;

            // debug.println("::GC/" + threads.getFrame(thr)
            //     + " after " + thisJoinPointStaticPart);

            System.gc();
            System.runFinalization();

            // If using PhantomReferences, we must do it twice...
            // System.gc();
            // System.runFinalization();

            Vector deadRefs = (Vector)deadRefHash.get(thr);
```

```
            if(deadRefs == null)
                deadRefHash.put(thr, deadRefs = new Vector());

            int thisFrame = threads.getFrame(thr);

            while((r = (DeathProbeReference)Q.poll()) != null) {
                deadRefs.add(r);
                refs.remove(r);
            }

            for(Iterator it = deadRefs.iterator(); it.hasNext();) {
                r = (DeathProbeReference)it.next();
                int birthFrame = r.getBirthFrame();

                if(birthFrame >= thisFrame) {
                    out.println("D " + r + " "
                                /* + birthFrame + " " */
                              + (thisFrame - birthFrame)
                              + " after " + thisJoinPointStaticPart);
                    it.remove();
                }
            }
        }

        return retval;
    }

    after() returning(Object o): constructorCall() {
        Thread thr = Thread.currentThread();
        refs.add(new DeathProbeReference(o, Q,
                                         thr, threads.getFrame(thr) - 1));
    }

    /** Aspect containing simple advice that forces the DeathProbe
     * aspect to be class-loaded before Probe.MainCutter's advice
     * runs. */
    static aspect MakeMyPresenceKnown {
        declare precedence : MakeMyPresenceKnown, Probe.MainCutter;

        /** Simple advice to force the DeathProbe aspect to be
         * class-loaded before Probe.MainCutter's advice runs. */
        before(): mainCut(String[]) {
            DeathProbe.hasAspect();
        }
    }
}
```

ReferenceProbe.java

```
package autoscope.probe;

import java.util.*;

import org.aspectj.lang.reflect.*;

/**
 * An aspect to detect and store to a file which objects in a system
 * refer to which others.
 */

aspect ReferenceProbe extends Probe {
    /** Whether or not to print stuff verbosely. */
    protected boolean verbose = false;

    /** A hash of encountered new sites. */
    protected Hashtable newLocHash = new Hashtable();

    /** A Hashtable of Threads to the SourceLocation of new sites */
    protected Hashtable newSiteThreadHash = new Hashtable();

    /** Returns our preferred filename extension "ref".  Returns the
     * String "ref", signalling that we want an output file with a
     * ".ref" extension. */
    protected String outExtension() { return "ref"; }

    /** Sets a "currently executing" new site SourceLocation for the
     * current Thread. */
    before(): constructorCall() {
        newSiteThreadHash.put(Thread.currentThread(),
                              thisJoinPointStaticPart.getSourceLocation());
    }

    /** Links the SourceLocation of a new site to the object currently
     * under construction.  The SourceLocation is retrieved from
     * newSiteThreadHash, which contains a mapping of Threads to the
     * new sites they are currently executing. */
    before(Object o): this(o) && execution(*.new(..)) && !withinUs() {
        /* In super() constructor invocations, we'll run multiple
           times.  This explicit null check keeps us from unnecessary
           work (and garbage). */
        if(newLocHash.get(o) == null) {
           /* this can be executed without a constructorCall() if this object
              is constructed reflectively, and x will be null */
            Object x = newSiteThreadHash.get(Thread.currentThread());
            if(x != null)
                newLocHash.put(o, x);
        }
```

```
}

/** Links the SourceLocation of a new site to a constructed
 * object.  The* SourceLocation is retrieved from this
 * (constructor call) join point.  This advice is necessary for
 * objects instantiated by our system but whose constructor code
 * is outside our system (for example, instantiating a String or
 * some other standard class in the Java class library. */
after() returning(Object o): constructorCall() {
    if(newLocHash.get(o) == null)
        newLocHash.put(o, thisJoinPointStaticPart.getSourceLocation());
}

/** This advice, on assignments, tracks which objects refer to
 * which others and calls reference() appropriately to generate
 * reference output. */
before(Object a, Object x):
        target(a) && args(x) && set(* *.*) && !withinUs() {
    if(x == null) {
        if(verbose)
            out.println("=== PROBE: x is null in nonstatic probe at " +
                        thisJoinPoint);
        return;
    }
    if(a == null)
        reference(new StaticClass(thisJoinPointStaticPart.getSignature()
                                  .getDeclaringType()), x);
    else reference(a, x);
}

/** This routine registers (and prints to the PrintWriter
 * specified in out) that an object refers to another object.  The
 * String printed for each item is the String returned by
 * getSrcLoc(Object), unless parameter "a" is an instance of
 * StaticClass, in which case the name of the encapsulated class
 * is output.
 * @param a the object that refers to object x
 * @param x the object that object a refers to
protected void reference(Object a, Object x) {
    try {
        out.println(( (a instanceof StaticClass) ?
                      ((StaticClass)a).getReferent().getName() :
                      getSrcLoc(a))
                    + ": " + getSrcLoc(x));
    } catch(NoSuchNewSiteException _) {}
}

/** Get a String representing the new site for the given object.
 * @param o the object to get the new site for
 * @return a String of the form "Foo(Foo.java<1:3>)"
```

```
     * @exception a ReferenceProbe.NoSuchNewSiteException is thrown if
     * the object's new site hasn't been detected (ie., it corresponds
     * to a library object, which AspectJ cannot track, or a primitive
     * type that was promoted to a class-based equivalent (like int ->
     * Integer) by AspectJ. */
    protected String getSrcLoc(Object o) throws NoSuchNewSiteException {
        org.aspectj.lang.reflect.SourceLocation loc =
          (org.aspectj.lang.reflect.SourceLocation)newLocHash.get(o);

        if(loc == null)
            throw new NoSuchNewSiteException();

        return o.getClass().getName() + "(" + loc.getFileName()
            + "<" + loc.getLine() + ":" + loc.getColumn() + ">)";
    }

    /** A class to handle cases where no new site exists for an object
     * (for example, a primitive int that AspectJ promotes to an
     * Integer. */
    static protected class NoSuchNewSiteException extends Exception {}

    /** Class used to track static reference information by
     * encapsulating a java.lang.Class */
    static final protected class StaticClass {
        private Class c;
        public StaticClass(Class c) { this.c = c; }
        public Class getReferent() { return c; }
    }

    /** Aspect containing simple advice that forces the ReferenceProbe
     * aspect to be class-loaded before Probe.MainCutter's advice
     * runs. */
    static aspect MakeMyPresenceKnown {
        declare precedence : MakeMyPresenceKnown, Probe.MainCutter;

        /** Simple advice to force the ReferenceProbe aspect to be
         * class-loaded before Probe.MainCutter's advice runs. */
        before(): mainCut(String[]) {
            ReferenceProbe.hasAspect();
        }
    }
}
```

SimpleReferenceCounter.aj

```
public interface ReferenceCountable { }

aspect SimpleReferenceCounter {
  public int ReferenceCountable.refCount = 0;

  private void free(ReferenceCountable rc) {
    processDeadObject(rc);

    /* return space occupied by rc to the general-purpose allocator:
       this simple reference-counter assumes there's a way to do
       interface to the VM to do this... */
  }

  protected void resetField(Field field, Object onInstance) {
    /* This prevents static fields from being reset. Resetting
     * static fields royally screws AspectJ over. (it's not
     * pretty) */
    if ((field.getModifiers() & Modifier.STATIC) == 0)
      return;

    try {
      Class type = field.getType();
      if(!type.isPrimitive())
        field.set(onInstance, null); // object/array
      else if(type == Boolean.TYPE)
        field.setBoolean(onInstance, false); // boolean
      else if(type == Character.TYPE)
        field.setChar(onInstance, (char)0); // char
      else field.setByte(onInstance, (byte)0);
      // the last one works for int short long float double byte
    } catch (Exception e) {
      System.err.println("Could not reset field");
      e.printStackTrace();
    }
  }

  private void processDeadObject(ReferenceCountable rc) {
    if (rc instanceof brutil.NoncyclicReference)
      return;

    Class c = rc.getClass();

    do {
      Field fields[] = c.getDeclaredFields();
      int length = fields.length;
      for (int i = 0; i < length; ++i) {
        try {
          Field theField = fields[i];
```

```
        // We can get away with != because field names are interned --
        // I can't find a guarantee in the spec but in Sun's J2SDK
        // 1.4.1 it works...
        if (theField.getName() != "rc_next" &&
            (theField.getModifiers() & Modifier.STATIC) == 0) {
          theField.setAccessible(true);
          Object value = theField.get(rc);
          if (value instanceof ReferenceCountable && value != rc) {
            ReferenceCountable rcValue = (ReferenceCountable)value;
            //debug("   - Death of " + rc + " decrementing " +
            //      rcField + ": " + (rcField.refCount-1));
            rcValue.refCount--;
            if (isCollectable(rcValue))
              free(rcValue);
            // theField.set(rc, null); // The bare minimum reset
          } else {
            // Overzealous?  If so, should remove theField.set() above, too
            // resetField(fields[i], rc);
          }
        }
      } catch(Exception e) {
        System.err.println("Error decrementing fields of dead object");
        e.printStackTrace();
      }
    }
  } while ((c = c.getSuperclass()) != null);
}

private boolean isCollectable(ReferenceCountable rc) {
  if (rc.refCount < 0)
    throw new Error("Sub-zero refCount");
  return rc.refCount == 0;
}

/* Capture all putfields, so we can see how objects interact. */
pointcut reference(Object obj, ReferenceCountable newVal) :
  target(obj) &&
  !within(SimpleReferenceCounter+) &&
  args(newVal) &&
  set((Object || (ReferenceCountable+)) *);

/* Gives the old and new referent of a ReferenceCountable to
 * subclasses of this aspect so that they may take appropriate
 * action. */
Object around(Object obj, ReferenceCountable newVal) : reference(obj, newVal) {
  ReferenceCountable oldVal = null;
  if (!(obj instanceof brutil.NoncyclicReference)) {
    org.aspectj.lang.Signature sig = thisJoinPointStaticPart.getSignature();
    Field field = null;
    Class c = sig.getDeclaringType();
```

```
    do {
      try {
        field = c.getDeclaredField(sig.getName());
        field.setAccessible(true);
        oldVal = (ReferenceCountable)field.get(obj);
      } catch(NoSuchFieldException e) {
        if((c = c.getSuperclass()) == null)
          throw new InternalError();
      } catch (Exception e) {
        System.err.println("Could not retrieve old value of field: " + field);
        e.printStackTrace();
        break;
      }
    } while(field == null);
  }
  Object ans = proceed(obj, newVal);
  /* This seems a little hokey, I know, but the new value must
   * know it is being referenced first, or it might think it's
   * garbage. In instances such as ListItems, the oldVal could
   * have a reference to the newVal, and it would then try to
   * decrement it on death. */
  if (!(obj instanceof brutil.NoncyclicReference)) {
    if (newVal != null && newVal != obj) {
      //debug("Object " + obj + " pointing at " + newVal + ": " +
      //      (((ReferenceCountable)newVal).refCount+1));
      // takeactionafterreference
      ++newVal.refCount;
    }
    if (oldVal != null && oldVal != obj) {
      //debug("Object " + obj + " pointing away from " + oldVal +
      //      ": " + (oldVal.refCount-1));
      // takeactionbeforereference
      --oldVal.refCount;
      if (isCollectable(oldVal))
        free(oldVal);
    }
  }
  return ans;
}

/* Allow us to allocate from 'undead' list if we have objects to offer. */
ReferenceCountable around () : allocation() {
  Class c = thisJoinPointStaticPart.getSignature().getDeclaringType();
  ReferenceCountable ret = popList(c);
  if (ret == null) {
    ret = proceed();
    return ret;
  }

  ret.rc_next = null; // sane, yes...but necessary?
```

```
    ret.refCount = 0; // should be 0 anyway, but just in case. ;)

    return ret;
  }
}
```

HeapReferenceCounter.aj

```
public interface ReferenceCountable { }

aspect HeapReferenceCounter {
  public ReferenceCountable ReferenceCountable.rc_next;
  public static ReferenceCountable ReferenceCountable+.rc_freeList;
  public ReferenceCountable ReferenceCountable+.rc_getList() {
    return rc_freeList;
  }
  public void ReferenceCountable+.rc_setList(ReferenceCountable rc) {
    rc_freeList = rc;
  }


  public int ReferenceCountable.refCount = 0;

  private void free(ReferenceCountable rc) {
    processDeadObject(rc);
    rc.rc_next = rc.rc_getList();
    rc.rc_setList(rc);
  }

  /* popList() gets and returns the first element of the
   * recycled-object list for the given type, and "pops" it off the
   * "top" of the list */
  private ReferenceCountable popList(Class type) {
    try {
      Field listField = type.getField("rc_freeList");

      // If I don't do setAccessible(true), I get the following:
      //    java.lang.IllegalAccessException:
      //      Class brutil.aspects.HeapCounting can not access a
      //      member of class brutil.SinglyLinkedListItem with
      //      modifiers "public static"
      listField.setAccessible(true);

      ReferenceCountable item = (ReferenceCountable)listField.get(null);
      if (item == null)
        return null;
      item.rc_setList(item.rc_next); // which is faster - invokeinterface?...
      // listField.set(null, item.rc_next); // ...or reflective invoke(final)?
      return item;
    } catch (Exception e) {
      System.err.println("popList: Could not pop rc_freeList");
      e.printStackTrace();
      return null;
    }
  }
```

```
protected void resetField(Field field, Object onInstance) {
  /* This prevents static fields from being reset. Resetting
   * static fields royally screws AspectJ over. (it's not
   * pretty) */
  if ((field.getModifiers() & Modifier.STATIC) == 0)
    return;

  try {
    Class type = field.getType();
    if(!type.isPrimitive())
      field.set(onInstance, null); // object/array
    else if(type == Boolean.TYPE)
      field.setBoolean(onInstance, false); // boolean
    else if(type == Character.TYPE)
      field.setChar(onInstance, (char)0); // char
    else field.setByte(onInstance, (byte)0);
    // the last one works for int short long float double byte
  } catch (Exception e) {
    System.err.println("Could not reset field");
    e.printStackTrace();
  }
}

private void processDeadObject(ReferenceCountable rc) {
  if (rc instanceof brutil.NoncyclicReference)
    return;

  Class c = rc.getClass();

  // Climb class hierarchy..
  do {
    Field fields[] = c.getDeclaredFields();
    int length = fields.length;
    for (int i = 0; i < length; ++i) {
      try {
        Field theField = fields[i];
        // We can get away with != because field names are interned --
        // I can't find a guarantee in the spec but in Sun's J2SDK
        // 1.4.1 it works...
        if (theField.getName() != "rc_next" &&
            (theField.getModifiers() & Modifier.STATIC) == 0) {
          theField.setAccessible(true);
          Object value = theField.get(rc);
          if (value instanceof ReferenceCountable && value != rc) {
            ReferenceCountable rcValue = (ReferenceCountable)value;
            //debug("   - Death of " + rc + " decrementing " +
            //      rcField + ": " + (rcField.refCount-1));
            rcValue.refCount--;
            if (isCollectable(rcValue))
              free(rcValue);
```

```
            theField.set(rc, null); // The bare minimum reset
          } else {
            // Overzealous?  If so, should remove theField.set() above, too
            // resetField(fields[i], rc);
          }
        }
      } catch(Exception e) {
        System.err.println("Error decrementing fields of dead object");
        e.printStackTrace();
      }
    }
  } while ((c = c.getSuperclass()) != null);
}

private boolean isCollectable(ReferenceCountable rc) {
  if (rc.refCount < 0)
    throw new Error("Sub-zero refCount");
  return rc.refCount == 0;
}

/* Capture all putfields, so we can see how objects interact. */
pointcut reference(Object obj, ReferenceCountable newVal) :
  target(obj) &&
  !within(HeapReferenceCounter+) &&
  args(newVal) &&
  set((Object || (ReferenceCountable+)) *);

/* Gives the old and new referent of a ReferenceCountable to
 * subclasses of this aspect so that they may take appropriate
 * action. */
Object around(Object obj, ReferenceCountable newVal) : reference(obj, newVal) {
  ReferenceCountable oldVal = null;
  if (!(obj instanceof brutil.NoncyclicReference)) {
    org.aspectj.lang.Signature sig = thisJoinPointStaticPart.getSignature();
    Field field = null;
    Class c = sig.getDeclaringType();
    do {
      try {
        field = c.getDeclaredField(sig.getName());
        field.setAccessible(true);
        oldVal = (ReferenceCountable)field.get(obj);
      } catch(NoSuchFieldException e) {
        if((c = c.getSuperclass()) == null)
          throw new InternalError();
      } catch (Exception e) {
        System.err.println("Could not retrieve old value of field: " + field);
        e.printStackTrace();
        break;
      }
    } while(field == null);
```

```
    }
    Object ans = proceed(obj, newVal);
    /* This seems a little hokey, I know, but the new value must
     * know it is being referenced first, or it might think it's
     * garbage. In instances such as ListItems, the oldVal could
     * have a reference to the newVal, and it would then try to
     * decrement it on death. */
    if (!(obj instanceof brutil.NoncyclicReference)) {
      if (newVal != null && newVal != obj) {
        //debug("Object " + obj + " pointing at " + newVal + ": " +
        //      (((ReferenceCountable)newVal).refCount+1));
        // takeactionafterreference
        ++newVal.refCount;
      }
      if (oldVal != null && oldVal != obj) {
        //debug("Object " + obj + " pointing away from " + oldVal +
        //      ": " + (oldVal.refCount-1));
        // takeactionbeforereference
        --oldVal.refCount;
        if (isCollectable(oldVal))
          free(oldVal);
      }
    }
    return ans;
  }

  /* Allow us to allocate from 'undead' list if we have objects to offer. */
  ReferenceCountable around () : allocation() {
    Class c = thisJoinPointStaticPart.getSignature().getDeclaringType();
    ReferenceCountable ret = popList(c);
    if (ret == null) {
      ret = proceed();
      return ret;
    }

    ret.rc_next = null; // sane, yes...but necessary?
    ret.refCount = 0; // should be 0 anyway, but just in case. ;)

    return ret;
  }
}
```

StackReferenceCounter.aj

```
public interface ReferenceCountable { }

aspect StackReferenceCounter extends HeapReferenceCounter {
  /* A stack that recycles its 'cells' for frame simulation. This
   * could not be done with our brutil.LinkedStack because of
   * circularity concerns. We could not use ArrayStack because we need
   * an unbounded Stack. Dirty, but necessary, since usually stack
   * frames try to add themselves to their own frame list right before
   * they are created. */
  public static class RCStack {
    private StackFrame free;
    private StackFrame top;

    private StackFrame manufactureStackFrame() {
      if (free == null)
        return new StackFrame();
      else return freePop();
    }

    private void freePush(StackFrame li) {
      li.next = free;
      free = li;
    }

    private StackFrame freePop() {
      StackFrame ans = free;
      free = free.next;
      ans.refList = null;
      return ans;
    }

    public boolean isEmpty() {
      return top == null;
    }

    // if we never encounter EmptyStackExceptions during runs, let's
    // remove the checks...

    public StackFrame getTop() {
      if (isEmpty())
        throw new EmptyStackException();
      return top;
    }

    public ReferenceCountable peek() {
      if (isEmpty())
        throw new EmptyStackException();
      return top.refList;
```

```
  }

  public void push() {
    StackFrame temp = top;
    top = manufactureStackFrame();
    top.next = temp;
  }

  public ReferenceCountable pop() {
    if (isEmpty())
      throw new EmptyStackException();
    ReferenceCountable ret = top.refList;
    StackFrame temp = top.next;
    freePush(top);
    top = temp;
    return ret;
  }

  public void setTopList(ReferenceCountable rc) {
    top.refList = rc;
  }
}

static class StackFrame {
  StackFrame next;
  ReferenceCountable refList;
}

////////////////////
/* Introductions */
////////////////////

/* To recipients of this aspect's advice: */
public StackFrame ReferenceCountable.frame;

//////////////////
/* Aspect State */
//////////////////

private RCStack frameSim = new RCStack();
private StackFrame staticFrame = new StackFrame();
private int stack = 0;

////////////////////
/* Helper Methods */
////////////////////

public void threadSlam(ReferenceCountable rc) {
  /* Associate this object to the static set. */
  ReferenceCountable ref = ((ReferenceCountable)rc);
```

```
  if (ref.frame == null) ref.incr();
  ref.frame = staticFrame;
}

public void takeActionBeforeReference(Object obj, ReferenceCountable oldVal) {
  if (oldVal != null && oldVal.frame != null && !frameSim.isEmpty())
    addToCurrentFrame(oldVal);
  super.takeActionBeforeReference(obj, oldVal);
}

public void addToCurrentFrame(ReferenceCountable rc) {
  if (rc.frame != null) return;
  rc.rc_next = frameSim.peek();
  frameSim.setTopList(rc);
  ++rc.refCount;
  //debug(" (" + stack + ")- Adding to current frame: " + rc + ": " +
  //      ((ReferenceCountable)rc).refCount);
  rc.frame = frameSim.getTop();
  //debug("<< From addToCurrent: Stack List: >>");
  //printList(rc);
}

public void objectLeavingScope(ReferenceCountable rc) {
  //debug (" Stack Decrementing " + rc + ": " +
  //       (((ReferenceCountable)rc).refCount-1));
  --rc.refCount;
}

///////////////
/* Pointcuts */
///////////////

pointcut functionScope() :
  !within(HeapTracking+) && call(* (* && !(RCStack || StackFrame)).*(..))
  || execution((* && !(RCStack || StackFrame)).new(..));

////////////
/* Advice */
////////////

after() returning(ReferenceCountable rc) : allocation() {
  if (!frameSim.isEmpty())
    addToCurrentFrame(rc);
  return rc;
}

/* This piece of advice handles the pushing and poping of stack
 * frames, and all ReferenceCountable concerns involved. */
Object around() : functionScope() {
  frameSim.push();
```

```
    stack++;
    Object ret = proceed();
    StackFrame popped = frameSim.getTop();
    ReferenceCountable rc = frameSim.pop();
    stack--;
    while (rc != null) {
      ReferenceCountable temp = rc.rc_next;
      rc.rc_next = null;
      rc.frame = null;
      if (rc == ret) {
        if (rc.frame == popped) {
          /* If we found the returned object on this list, and this is
             not the initial frame (or a static frame), add the returned
             object to the next frame's list. */
          //debug(" Returning " + rc);
          if (!frameSim.isEmpty()) {
            addToCurrentFrame(rc);
            /* Messy, but addToCurrentFrame will incr, and we need this to
             * maintain correct refCount. */
            --rc.refCount;
          }
        }
      } else {
        /* If something other than this frame references the
           object, dissociate it from its frame. (it is now
           'orphaned') */
        objectLeavingScope(rc);
      }
      rc = temp;
    }
    return ret;
  }
}
```

Appendix B

Garbage Collector Software Support ChangeLog

This appendix provides the detailed jRate [34] log of changes for the software support described in Chapter 6. This is a complete log, giving the jRate and the GNU Compiler Collection (GCC) source files modified for each task.

- add "soft" nodes in Java front-end for `_Jv_SMM_getObjectData()`, `_Jv_SMM_putObjectData()`, and various flavors of `_Jv_SMM_bumpRefCount()` and `_Jv_SMM_decRefCount()` (gcc/java/decl.c) and also `JTI_SOFT_*` constants and #defines for them (gcc/java/java-tree.h)
- add "soft" nodes for `gnu::gcj::RawData` type and needed SMM functions in C++ front-end (gcc/cp/typeck.c, gcc/cp/cp-tree.h)
- add `build_object_in_smm_cond()` function to Java front-end: constructs a tree for a runtime conditional for a given tree of ptr type to determine if it's in SMM unit or not; optimized for ARTEC case (gcc/java/java-tree.h, gcc/java/expr.c)
- add `build_object_in_smm_cond()` function to C++ front-end; optimized for ARTEC case (gcc/cp/cp-tree.h, gcc/cp/expr.c)
- initialize C++ local variables of Java pointer type to *null* (gcc/cp/decl.c)
- in `expand_java_arraystore()`: support for arrays on bytecode-to-object compilation path (gcc/java/expr.c)

- in `expand_java_arrayload()`: support for arrays on bytecode-to-object compilation path (gcc/java/expr.c)
- in `expand_java_array_length()`: support for arrays on bytecode-to-object compilation path (gcc/java/expr.c)
- initialize Java non-parameter local variables/stack slots containing object references to null (gcc/java/decl.c)
- emit cleanup functions for Java local variables/stack slots to decrement their reference counts on function exit (gcc/java/decl.c)
- emit proper `bumpRefCount`/`decRefCount` calls when local variables/stack slots assigned to (gcc/java/decl.c, gcc/java/expr.c)
- emit cleanup functions for C++ local variables (gcc/java/decl.c)
- emit reference count bumps for C++ function parameters on function entry (gcc/cp/decl.c)
- emit cleanups to decrement reference counts for C++ function parameters on function exit (gcc/cp/decl.c)
- split RTL variable assignment for local variables/stack slots depending not only on `TYPE_MODE` but also on SMM-allocability (gcc/java/decl.c)
- increment SMM reference counts for C++ local variables of type *pointer-to-Java-object* when first initialized (gcc/cp/decl.c, gcc/cp/typeck.c, gcc/cp/cp-tree.h)
- increment and decrement SMM reference counts when C++ local variables of type *pointer-to-Java-object* are assigned (gcc/cp/typeck.c, gcc/cp/cp-tree.h)
- decrement SMM reference counts when C++ local variables of type *pointer-to-Java-object* go out of scope (gcc/cp/decl.c, gcc/cp/typeck.c, gcc/cp/cp-tree.h)
- in `expand_java_field_op()`: support for `getstatic`, `putstatic`, `getfield`, `putfield` SMM instrumentation on bytecode-to-object compilation path (gcc/java/expr.c)
- in `invoke_build_dtable()`: support for (virtual) method calls from Java on SMM-allocated objects (gcc/java/expr.c)

- added `-fsmm`/`-fno-smm` Java compiler options and `flag_smm` compiler global (gcc/java/java-tree.h) – also pass `-fsmm` to `jc1` (gcc/java/jvspec.c), document it (gcc/java/lang-options.h), and disallow it's use with `-femit-class-file[s]` (gcc/java/lang-specs.h). Define `flag_smm` global and hook it up to command-line processing (gcc/java/lang.c).
- added `-fsmm` C++ compiler option and `flag_smm` compiler global (gcc/c-opts.c, gcc/cp/lang-options.h, gcc/c-common.c, gcc/c-common.h)
- emit error when specifying `-fsmm` and compiling from Java source, for which SMM instrumentation isn't supported (gcc/java/parse.y)
- backport a 4.0.x-series fix for a gcj bug that breaks the source-to-bytecode compilation path for subsequent static calls (*e.g.* "`Runtime.getRuntime().runFinalizersOnExit(finalizeOnExit)`" from libjava/java/lang/System.java) (gcc/java/parse.y)
- alter `libjava`'s build process to generate `.o` files from `.class` rather than `.java` (so that `-fsmm` is effective for class library) and alter rules to make this possible (libjava/Makefile.am, libjava/Makefile.in)
- no longer build jRate demos on install—to work with SMM, the build process would need alteration to build `.class` files first (in jRate source: Makefile.am)
- include SMM function prototypes in standard library (libjava/include/jvm.h)
- new header file "smm.h" in standard library to declare SMM functions (libjava/include/smm.h)
- new source file "smm.cc" in standard library to implement SMM functions (libjava/smm.cc)
- make certain allocation routines friends of `java::lang::Class` (libjava/java/lang/Class.h)
- emit proper pointer mask for SMM-allocable classes (gcc/java/boehm.c)
- arrange for `_Jv_SMM_sendStructDef()` to be called at class loading time, using size (in words) of class and pointer mask, for classes that are SMM-allocable (libjava/java/lang/natClass.cc)

- add `smm_structID` and `gc_descr64` integer fields to `java.lang.Class` (gcc/java/class.c, gcc/java/decl.c, libjava/java/lang/Class.h)
- arrange for `_Jv_SMM_init()` to be called at JVM initialization (libjava/prims.cc)
- add new `artec_fpga_allocation` global to runtime (zero if SMM unit disabled, nonzero if SMM unit enabled)
- alter allocation routines to use SMM unit, when enabled, for objects that can be SMM-allocated and when SMM isn't already full (libjava/prims.cc)
- jRate's initial memory area is heap, instead of method area as in stock jRate (libjava/prims.cc; in jRate source: src/native/gcj-patches/jRate-gc.cc)
- fixed `libtool` to properly quote shell meta characters when compiling `.class` files (ltmain.sh in top-level GCC sources)
- fixed inner class usage error that gcj accepts but compiles erroneously (in jRate source: src/javax/realtime/PriorityQueue.java)
- added built-in defines `__JRATE_SMM_ADDR_TOP__` and `__JRATE_SMM_ADDR_BOTTOM__` for additional ease in making sure compiler and runtime use same values (gcc/c-common.c)
- added support for using the SMM `cloneObject()` routine if the object to be cloned is in SMM *and* the SMM unit is currently enabled for allocation (libjava/java/lang/natObject.cc)
- remove problematic piece of code (address-expression on member access) dealing with collection of references (libjava/java/lang/ref/natReference.cc)
- provide minimally-functional SMM function stubs for testing purposes (libjava/smm.cc)
- integrate with Boeing MemoryManager layer `HARDWARE` and `SOFTWARE` configurations, selectable via environment variable (libjava/smm.cc, libjava/configure, libjava/Makefile.am, libjava/Makefile.in, import of MemoryManager tree into libjava/MemoryManager; in jRate source: Makefile.am)
- fix `libgcj` build process to support MemoryManager without itself being SMM-instrumented (libjava/Makefile.am, libjava/Makefile.in)

- solve chicken-&-egg bootstrap problem in which the `SMM_Driver` class from the MemoryManager layer cannot be told to initialize until after it processes own `sendStructDef`! (libjava/smm.cc)
- C++ front-end changes for native code SMM field read/write (gcc/cp/typeck.c, gcc/cp/expr.c, gcc/cp/error.c, gcc/cp/cp-tree.h)
- C++ front-end changes for native code SMM static/global write (gcc/cp/typeck.c, gcc/cp/expr.c, gcc/cp/error.c, gcc/cp/cp-tree.h)
- C++ front-end changes for native code SMM vtable access to support C++-side calls of Java methods on SMM objects (gcc/cp/typeck.c, gcc/cp/call.c, gcc/cp/init.c, gcc/cp/expr.c, gcc/cp/class.c, gcc/cp/cp-tree.h)
- C++ front-end changes for native code bump/dec ref count (gcc/cp/typeck.c, gcc/cp/expr.c,gcc/cp/error.c, gcc/cp/cp-tree.h)
- new `IS_SMM_ACCESS` flag for `COND_EXPR`s in C++ front end to track if an SMM access is occurring in intermediate representation (gcc/cp/cp-tree.h)
- new `SMM_FIELD_OFFSET` macro in C++ front end to get the offset for a field for use in `getObjectData`/`putObjectData` calls (gcc/cp/cp-tree.h)
- new `JAVA_TYPE_IS_WIDE` macro in C++ front end to determine if a field has Java type double or long (gcc/cp/cp-tree.h)
- C++ native array read/write (libjava/gcj/array.h, numerous `libgcj` and `libjRateCore` library modifications across many files)
- minor modifications to 3.3.3 versions of `libiberty` and the C++ front end so that they can be compiled with (the stricter) gcc 4.0.x (include/obstack.h, gcc/cp/decl.c)
- fix error introduced by SMM instrumentation when building `gcjh` (gcc/java/gjavah.c).
- remove per-object `monitor_policy` from jRate (libjava/java/lang/Object.h, libjava/java/lang/natObject.cc; in jRate source: src/native/src/jrate/binding/java/ObjectInitializer.h, src/native/src/javax/realtime/MonitorControl.cc)
- make native `getClass()` method in `java.lang.Object` aware of SMM (libjava/java/lang/natObject.cc)

- fix bug in gcc 3.3.3 that issues an incorrect "value computed is not used" warning for `SAVE_EXPR` nodes (gcc/stmt.c)
- fix a bug in gcj 3.3.3 that doesn't look up methods properly in interface/abstract contexts when compiling from bytecode; import some method lookup code from gcj 3.4.4 (gcc/java/typeck.c)
- don't include jRate's `get_ticks()` function when building for PowerPC, else we get a multiply-defined conflict between it and the (identical) version in MemoryManager when linking `libgcj` (in jRate source: src/native/src/javax/realtime/HighResolutionClock.cc)
- add support for C++ template expansion of SMM local variable cleanup code (gcc/cp/pt.c)
- add support for `#pragma GCC SMM` for fine-grained control of SMM instrumentation in C++ source, also associated global and garbage collection root information (gcc/cp/lex.c, gcc/cp/Make-lang.in gcc/cp/config-lang.in)
- add `jRate/SMM` identification to the compiler (gcc/gcc.c, gcc/toplev.c, gcc/diagnostic.c)
- fix native `System.arraycopy()` to work with new C++-side array types and to support copying into and out of SMM (libjava/java/lang/natSystem.cc)

References

[1] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, Boston, Massachusetts, USA, 2001.

[2] Andrei Alexandrescu. Loki C++ library. `sourceforge.net/projects/loki-lib/`, 2006.

[3] Aonix, Inc. Aonix - PERC Products. `www.aonix.com/perc.html`, 2007.

[4] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. *ACM SIGPLAN Notices*, 23(7):11–20, July 1988.

[5] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, Boston, Massachusetts, USA, 2000.

[6] The AspectJ Organization. *Aspect-Oriented Programming for Java*, 2007. `www.aspectj.org`.

[7] The AspectJ Organization. *The AspectJ 5 Development Kit Developer's Notebook*, 2007. `www.eclipse.org/aspectj/doc/released/adk15notebook/`.

[8] The AspectJ Organization. *The AspectJ Programming Guide*, 2007. `www.eclipse.org/aspectj/doc/released/progguide/`.

[9] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: The deadline monotonic approach. In *Proceedings of the Eighth IEEE Workshop on Real-Time Operating Systems and Software (RTOSS 1991)*, Atlanta, Georgia, USA, May 1991.

[10] Matthew H. Austern. *Generic Programming and the STL*. Addison-Wesley, Reading, Massachusetts, USA, 1999.

[11] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and

Julian Tibble. Optimising AspectJ. *ACM SIGPLAN Notices*, 40(6):117–128, 2005.

[12] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithmic language ALGOL 60. *Communications of the ACM (CACM)*, 6(1):1–17, January 1963.

[13] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2003)*, pages 285–298. ACM Press, 2003.

[14] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[15] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM (CACM)*, 21(4):280–294, April 1978.

[16] Henry G. Baker. The Treadmill: real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27(3):66–70, March 1992.

[17] Don Batory, Roberto E. Lopez-Herrejon, and Jean-Philippe Martin. Generating product-lines of product-families. In *Proceedings of the 17th IEEE international conference on Automated software engineering (ASE 2002)*, pages 81–92, Washington, DC, USA, 2002. IEEE Computer Society Press.

[18] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.

[19] William S. Beebee, Jr. and Martin Rinard. An implementation of scoped memory for real-time Java. *Lecture Notes in Computer Science*, 2211:289–305, October 2001.

[20] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9), September 1988. `www.hpl.hp.com/personal/Hans_Boehm/spe_gc_paper/`.

[21] Greg Bollella, Ben Brosgol, Peter Dibble, Steve Furr, James Gosling, David Hardin, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, Boston, Massachusetts, USA, 2000.

[22] Daniel Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, second edition, 2002.

[23] Claus Brabrand and Michael I. Schwartzbach. Growing languages with metamorphic syntax macros. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2002)*, pages 31–40. ACM Press, 2002.

[24] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of the thirteenth annual conference on Object-oriented programming, systems, languages, and applications (OOPSLA 1998)*, pages 183–200, Vancouver, British Columbia, Canada, 1998. ACM Press.

[25] Rodney A. Brooks. Trading data space for reduced time and code space in real-time collection on stock hardware. In *ACM Symposium on LISP and Functional Programming (LFP 1984)*, pages 256–262, Austin, Texas, USA, 1984. ACM Press.

[26] R. Brown and L. Hendren. Automatic recycling of Java objects. Technical report, McGill University, 2000.

[27] Adam L. Buchsbaum, Haim Kaplan, Anne Rogers, and Jeffery R. Westbrook. A new, simpler linear-time dominators algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(6):1265–1296, 1998.

[28] Dante J. Cannarozzi, Michael P. Plezbert, and Ron K. Cytron. Contaminated garbage collection. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI 2000)*, pages 264–273, Vancouver, British Columbia, Canada, June 2000. ACM Press.

[29] Dante Cannarozzi and Morgan Deters. Secure, distributed whiteboard. `www.cs.wustl.edu/~mdeters/whiteboard/`, 2002.

[30] Perry Cheng and Guy Belloch. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI 2001)*, pages 125–136, Snowbird, Utah, USA, June 2001. ACM Press.

[31] Shigeru Chiba. A metaobject protocol for C++. In *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications (OOPSLA 1995)*, pages 285–299, Austin, Texas, USA, October 1995. ACM Press.

[32] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill Higher Education, second edition, 2001.

[33] Angelo Corsaro. Techniques and patterns for safe and efficient real-time middleware. Technical Report WUCSE–2004–54, Washington University in St. Louis, Department

of Computer Science and Engineering, September 2004. Doctoral dissertation.

[34] Angelo Corsaro and Morgan Deters. jRate: A Real-Time Java ahead-of-time compiler. `jrate.sourceforge.net/`, 2006.

[35] Ole-Johan Dahl and Kristen Nygaard. SIMULA: an ALGOL-based simulation language. *Communications of the ACM (CACM)*, 9(9):671–678, 1966.

[36] Morgan Deters. Dynamic assignment of scoped memory regions in the translation of Java to Real-Time Java. Technical Report WUCSE–03–27, Washington University in St. Louis, Department of Computer Science and Engineering, May 2003. M.S. Thesis.

[37] Morgan Deters. The tortured history of real-time garbage collection. `www.cs.wustl.edu/~mdeters/doc/slides/rtgc-history.pdf`, October 2003. Presented in the Fall 2003 Seminar on Programming Languages, Department of Computer Science, Washington University.

[38] Morgan Deters and Ron K. Cytron. Automated discovery of scoped memory regions for real-time Java. In *Proceedings of the third international symposium on Memory management (ISMM)*, pages 25–35, Berlin, Germany, June 2002. ACM Press.

[39] Morgan Deters, Christopher Gill, and Ron Cytron. Rate-monotonic analysis in the C++ typesystem. In *Proceedings of the RTAS 2003 Workshop on Model-Driven Embedded Systems (MDES)*, Washington, DC, USA, May 2003.

[40] Morgan Deters, Nicholas A. Leidenfrost, Matthew P. Hampton, James C. Brodman, and Ron K. Cytron. Automated reference-counted object recycling for Real-Time Java. In *Proceedings of the Tenth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2004)*, pages 424–433, Toronto, Canada, May 2004.

[41] D. M. Dhamdhere. A fast algorithm for code movement optimisation. *ACM SIGPLAN Notices*, 23(10):172–180, 1988.

[42] Distributed Object Computing Group. nORB - special purpose middleware for networked embedded systems. `deuce.doc.wustl.edu/nORB/`, 2007.

[43] Robert Dyer and Hridesh Rajan. Modular compilation strategies for aspect-oriented constructs. Technical Report 06–30, Iowa State University, Department of Computer Science, 2006. `archives.cs.iastate.edu/documents/disk0/00/00/04/90/00000490-00/main.pd%f`.

[44] Tzilla Elrad, Robert E. Filman, and Atef Bader, editors. *Communications of the ACM (CACM)*, 44(10)*: Special Issue on Aspect-Oriented Programming*. ACM Press, October 2001.

[45] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Communications of the ACM (CACM)*, 44(10):29–32, 2001.

[46] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[47] Xavier Franch. Systematic formulation of non-functional characteristics of software. In *Proceedings of the International Conference on Requirements Engineering (ICRE 1998)*, pages 174–181, Colorado Springs, Colorado, USA, 1998.

[48] Free Software Foundation. GCC Home Page. `gcc.gnu.org`, 2006.

[49] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, USA, 1995.

[50] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Massachusetts, USA, 2000.

[51] Stefan Hanenberg and Rainer Unland. Parametric introductions. In *Proceedings of the 2nd International Conference On Aspect-Oriented Software Development (AOSD '03)*, pages 80–89, 2003.

[52] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the seventeenth annual conference on Object-oriented programming, systems, languages, and applications (OOPSLA 2002)*, pages 161–173, Seattle, Washington, USA, November 2002. ACM Press.

[53] Frank Hunleth. Building customizable middleware using aspect-oriented programming. Technical Report WUCS–02–07, Washington University in St. Louis, Department of Computer Science, May 2002. M.S. Thesis.

[54] Frank Hunleth and Ron K. Cytron. Footprint and feature management using aspect-oriented programming techniques. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 38–45. ACM Press, 2002.

[55] Frank Hunleth, Ron Cytron, and Christopher Gill. Building customizable middleware using aspect oriented programming. In *The OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa Bay, Florida, USA, October 2001. ACM Press. `www.cs.ubc.ca/~kdvolder/Workshops/OOPSLA2001/ASoC.html`.

[56] Frank Hunleth, Ravi Pratap Maddimsetty, Ron K. Cytron, and Christopher D. Gill. FACET—Framework for Aspect Composition for an EvenT channel. `www.cs.wustl.edu/~doc/RandD/PCES/facet/`, 2007.

[57] IBM Research. *HyperJ(TM): Multi-Dimensional Separation of Concerns for Java(TM)*, 2007. `www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm`.

[58] IBM Research. Jikes^{tm} RVM home page. `jikesrvm.sourceforge.net/`, 2007.

[59] Kathleen Jensen and Niklaus Wirth. *Pascal: User Manual and Report*. Springer-Verlag, Secaucus, New Jersey, USA, 1975.

[60] Ralph E. Johnson. Reducing the latency of a real-time garbage collector. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(1):46–58, March 1992.

[61] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[62] Richard Jones and Rafael Lins. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley & Sons, Inc., New York, New York, USA, 1996.

[63] Alan C. Kay. The early history of Smalltalk. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 69–95, New York, New York, USA, 1993. ACM Press.

[64] Brian Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1978.

[65] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Massachusetts, USA, 1991.

[66] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with AspectJ. *Communications of the ACM (CACM)*, 44(10):59–65, 2001.

[67] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 1997. Springer-Verlag.

[68] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI 1992)*, pages 224–234, San Francisco, California, USA, June 1992.

[69] Donald E. Knuth. *The Art of Computer Programming, volume I: Fundamental Algorithms*. Addison-Wesley, second edition, 1973. Chapter 2.

[70] Ákos Lédeczi, Miklos Maróti, and Péter Völgyesi. The generic modeling environment. Technical report, Institute for Software Integrated Systems, Vanderbilt University, 2001. `www.isis.vanderbilt.edu/projects/gme/GMEReport.pdf`.

[71] J. P. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm—exact characterization and average-case behaviour. *IEEE Real-Time Systems Symposium*, pages 166–171, December 1989.

[72] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141, 1979.

[73] Henry Lieberman and Carl E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM (CACM)*, 26(6):419–429, June 1983.

[74] Tom Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, USA, 1997.

[75] Martin Linenweber. A study in Java bytecode engineering with PCESjava. Technical Report WUCSE–03–17, Washington University in St. Louis, Department of Computer Science and Engineering, May 2003. M.S. Thesis.

[76] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, January 1973.

[77] Jane W. S. Liu. *Real-Time Systems*. Prentice-Hall, Upper Saddle River, NJ, USA, 2000.

[78] Jane W. S. Liu, Juan-Luis Redondo, Zhong Deng, Too-Seng Tia, Riccardo Bettati, Ami Silberman, Matthew Storch, Rhan Ha, and Wei-Kuan Shih. PERTS: A prototyping environment for real-time systems. Technical Report UIUCDCS-R-93-1802, University of Illinois at Urbana-Champaign, May 1993.

[79] Ravi Pratap Maddimsetty. Efficient customizable middleware. Technical Report WUCSE–2003–78, Washington University in St. Louis, Department of Computer Science, December 2003. M.S. Thesis.

[80] Pattie Maes. Concepts and experiments in computational reflection. In *Conference proceedings on Object-oriented programming systems, languages, and applications*

(OOPSLA), pages 147–155, Orlando, Florida, USA, October 1987. ACM Press.

[81] Tobias Mann, Morgan Deters, Rob LeGrand, and Ron K. Cytron. Static determination of allocation rates to support real-time garbage collection. In *Proceedings of the 2005 ACM Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2005)*, pages 193–202, Chicago, Illinois, USA, June 2005. ACM Press.

[82] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM (CACM)*, 3(4):184–195, 1960.

[83] Michael Metcalf and John Reid. *FORTRAN 90/95 Explained, second edition*. Oxford University Press, Oxford, United Kingdom, 1999.

[84] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., 1997.

[85] Scott Nettles and James O'Toole. Real-time replication garbage collection. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI 1993)*, pages 217–226, Albuquerque, New Mexico, USA, June 1993. ACM Press.

[86] Kelvin D. Nilsen. Garbage collection of strings and linked data-structures in real-time. *Software Practice and Experience*, 18(7):613–640, July 1988.

[87] Kelvin D. Nilsen and William J. Schmidt. Hardware support for garbage collection of linked objects and arrays in real-time. In *Proceedings of the OOPSLA/ECOOP 1990 Workshop on Garbage Collection in Object-Oriented Systems*, Ottawa, Canada, October 1990.

[88] Kelvin Nilsen. Issues in the design and implementation of real-time Java. *Java Developer's Journal*, 1(1):44, 1996.

[89] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on practical software development environments (SDE 1984)*, pages 177–184, New York, New York, USA, 1984. ACM Press.

[90] Oxford University Programming Tools Group and the McGill University Sable Research Group. abc: The AspectBench Compiler for AspectJ. `www.aspectjbench.org/`, 2007.

[91] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*

(PLDI 1988), pages 199–208, Atlanta, Georgia, USA, 1988.

[92] R. M. Pratap, F. Hunleth, and R. K. Cytron. Building fully customisable middleware using an aspect-oriented approach. *Software*, 151(4):199–218, 2004.

[93] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture, volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, Inc., New York, New York, USA, 2000.

[94] William J. Schmidt and Kelvin D. Nilsen. Performance of a hardware-assisted real-time garbage collector. In *ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 76–85, New York, New York, USA, 1994. ACM Press.

[95] Lui Sha and John B. Goodenough. Real-time scheduling theory and Ada. *IEEE Computer*, 23(4):53–62, 1990.

[96] Yefim Shuf, Manish Gupta, Rajesh Bordawekar, and Jaswinder Pal Singh. Exploiting prolific types for memory management and optimizations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 295–306. ACM Press, 2002.

[97] Fridtjof Siebert. Hard real-time garbage-collection in the Jamaica Virtual Machine. In *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA 1999)*, Hong Kong, December 1999. IEEE Computer Society Press.

[98] SPEC Corporation. Java SPEC benchmarks. Technical report, SPEC, 1999. Available by purchase from SPEC.

[99] Bjarne Stroustrup. *The C++ Programming Language, Special Edition*. Addison-Wesley, Boston, Massachusetts, USA, 2000.

[100] Venkita Subramonian, Christopher Gill, and Doug Stuart. Design and implementation of nORB. Technical report, Washington University in St. Louis, Department of Computer Science and Engineering. `www.cs.wustl.edu/~venkita/nORB/norbtechreport.pdf`.

[101] Sun Microsystems, Inc. Tuning garbage collection with the 1.4.2 java(tm) virtual machine. `java.sun.com/docs/hotspot/gc1.4.2/`, 2003.

[102] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering*, pages 107–119, Los Angeles, California,

USA, May 1999. IEEE Computer Society Press.

[103] Michiaki Tatsubori, Shigeru Chiba, Kozo Itano, and Marc-Olivier Killijian. OpenJava: A class-based macro system for Java. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Reflection and Software Engineering*, volume 1826 of *Lecture Notes in Computer Science*, pages 117–133, Heidelberg, Germany, 2000. Springer-Verlag.

[104] TimeSys Corporation. TimeSys TimeWiz. `www.timesys.com/index.cfm?hdr=tools_header.cfm&bdy=tools_bdy_model.cfm`, 2003.

[105] Mads Tofte. A brief introduction to regions. In *Proceedings of the first international symposium on Memory management (ISMM)*, pages 186–195, Vancouver, British Columbia, Canada, October 1998. ACM Press.

[106] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, February 1997.

[107] Erwin Unruh. Primzahlen. `www.erwin-unruh.de/primorig.html`, 2007.

[108] Todd Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[109] Todd Veldhuizen. Techniques for scientific C++. Technical Report TR542, Indiana University, Department of Computer Science, August 2000.

[110] John Viega, J. T. Bloch, and Pravir Chandra. Applying aspect-oriented programming to security. *Cutter IT Journal*, 14(2):31–39, February 1994.

[111] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming (ICFP 2003)*, pages 127–139, Uppsala, Sweden, August 2003.

[112] Mark Weiser. Program slicing. In *Proceedings of the fifth international conference on software engineering (ICSE 1981)*, pages 439–449, San Diego, California, USA, 1981. IEEE Computer Society Press.

[113] Paul R. Wilson. Uniprocessor garbage collection techniques (long version). Submitted to ACM Computing Surveys. ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps, 1994.

[114] Bart De Win, Frank Piessens, Wouter Joosen, and Tine Verhanneman. On the importance of the separation-of-concerns principle in secure software engineering. In *Proceedings of the workshop on the Application of Engineering Principles to System Security Design*, Boston, Massachusetts, USA, November 2002. Available at

http://www.acsac.org/waepssd/papers/02-piessens.pdf.

[115] Niklaus Wirth. On the design of programming languages. In *IFIP Congress*, pages 386–393, 1974.

[116] Victor Fay Wolfe, Russell Johnston, Peter Kortmann, Ben Watson, Steven Wohlever, Lisa C. DiPippo, Rama Bethmagalkar, and Gregory Cooper. RapidSched: Static scheduling and analysis for Real-Time CORBA. In *Proceedings of the 5th International Workshop on Real-Time Object-Oriented Dependable Systems*. IEEE, January 1999.

Index

$\mathcal{A}$

$\mathcal{B}$

C

D

E

F

G

H

I

J

K

$\mathcal{P}$

$\mathcal{R}$

S

T

Revision History

Brief revision notes for this printing and previous ones are listed below, along with doctoral program milestones and dates. The most recent revision and a full list of thesis errata, corresponding to all versions ever in print, will be made available online as they become available:

`http://www.morgandeters.com/dissertation/`

Washington University's Department of Computer Science and Engineering publishes technical reports online dating back to 1996:

`http://cse.seas.wustl.edu/research-techreports.asp`

---

30 Apr 2007 The copy you are now reading was typeset (TEX job *main* at 12:33pm).

30 Apr 2007 Committee suggestions incorporated. Submitted to the Graduate School of Arts & Sciences and published as technical report WUCSE–2007–29.

24 Apr 2007 Public dissertation defense.

06 Apr 2007 Dissertation submitted to the Sever Institute for format review.

07 Nov 2005 Public proposal defense.

27 Jun 2003 Accepted as candidate in doctoral program.

Curriculum Vitæ

Morgan G. Deters

Degrees Held

Doctor of Philosophy, Computer Science
Washington University in St. Louis, May 2007

Master of Science, Computer Science
Washington University in St. Louis, May 2003

Bachelor of Science, Cum Laude, Computer Science
Bowling Green State University, May 2000

Refereed Publications

Tobias Mann, Morgan Deters, Rob LeGrand, and Ron K. Cytron. Static determination of allocation rates to support real-time garbage collection. In *Proceedings of the 2005 ACM Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2005)*, pp. 193–202, Chicago, Illinois, June 2005. ACM Press.

Morgan Deters, Nicholas A. Leidenfrost, Matthew P. Hampton, James C. Brodman, and Ron K. Cytron. Automated reference-counted object recycling for Real-Time Java. In *Proceedings of the Tenth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2004)*, pp. 424–433, Toronto, Canada, May 2004. IEEE Computer Society.

Morgan Deters, Christopher Gill, and Ron Cytron. Rate-monotonic analysis in the C++ type system. In *Proceedings of the RTAS 2003 Workshop on Model-Driven Embedded Systems (MDES)*, Washington, DC, May 2003.

Morgan Deters and Ron K. Cytron. Automated discovery of scoped memory regions for Real-Time Java. In *Proceedings of the 2002 International Symposium on Memory Management (ISMM 2002)*, pp. 25–35, Berlin, Germany, June 2002. ACM Press.

Steven M. Donahue, Matthew P. Hampton, Morgan Deters, Jonathan M. Nye, Ron K. Cytron, and Krishna M. Kavi. Storage allocation for real-time, embedded systems. In *Embedded Software: Proceedings of the First International Workshop*, volume 2211 of *Lecture Notes in Computer Science*, pp. 131–147, Tahoe City, California, USA, October 2001. Springer-Verlag.

Morgan Deters and Ron K. Cytron. Introduction of program instrumentation using aspects. In *Proceedings of the ACM OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa Bay, Florida, USA, October 2001. `http://www.cs.ubc.ca/~kdvolder/Workshops/OOPSLA2001/ASoC.html`.

Morgan Deters, Nicholas Leidenfrost, and Ron K. Cytron. Translation of Java to Real-Time Java using aspects. In *Proceedings of the International Workshop on Aspect-Oriented Programming and Separation of Concerns*, pp. 25–30, Lancaster, United Kingdom, August 2001. Proceedings published as technical report CSEG/03/01 by the Computing Department, Lancaster University, United Kingdom.

Publications in Preparation

Tobias Mann, Morgan Deters, Robert LeGrand, and Ron K. Cytron. Static Determination of Allocation Rates to Support Real-Time Garbage Collection. Extended version of LCTES 2005 paper.

Delvin Defoe, Morgan Deters, and Ron K. Cytron. An Efficient On-the-fly Reference-Counting Garbage Collector (working title).

Teaching Experience

Organizer and lecturer, *Demystifying GCC: Under the Hood of the GNU Compiler Collection*
Half-day tutorial offered at the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Portland, Oregon, 2006.

Organizer and moderator, *Doctoral Programming Language Seminar: Internals of the GNU Compiler Collection* (CSE 7201)
Washington University in St. Louis, Fall 2005.

Guest lecturer, *Embedded Computing Systems* (CSE 467)
Washington University in St. Louis, April 2005. Delivered one lecture.

Guest lecturer, *Advanced Multi-Paradigm Software Development* (CSE 562)
Washington University in St. Louis, January 2003. Delivered one lecture.

Lecturer, *Real-Time Java*
Center for the Application of Information Technology (CAIT), St. Louis, September 2002–July 2004. Taught three classes with Ron Cytron.

Lecturer, *Aspect-Oriented Programming with AspectJ* (CSE 6783)
Washington University in St. Louis, Fall 2002. Delivered 12 lectures.

Organizer and moderator, *Doctoral Programming Language Seminar: Features of Programming Languages* (CS 6782)
Washington University in St. Louis, Spring 2002.

Professional Societies

Association for Computing Machinery (ACM), ACM Special Interest Group on Programming Languages (SIGPLAN), Institute of Electrical and Electronics Engineers (IEEE), IEEE Computer Society

May 2007