WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

DYNAMIC ASSIGNMENT OF SCOPED MEMORY REGIONS IN THE TRANSLATION OF JAVA TO REAL-TIME JAVA

by

Morgan G. Deters

Prepared under the direction of Ron K. Cytron

A thesis presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of
Master of Science

May, 2003

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

DYNAMIC ASSIGNMENT OF SCOPED MEMORY REGIONS IN THE TRANSLATION OF JAVA TO REAL-TIME JAVA

by Morgan G. Deters

ADVISOR: Ron K. Cytron

May, 2003
Saint Louis, Missouri

Advances in middleware, operating systems, and popular, general-purpose languages have brought the ideal of reasonably-bound execution time closer to developers who need such assurances for real-time and embedded systems applications. Extensions to the Java libraries and virtual machine have been proposed in a real-time Java standard, which provides for specification of release times, execution costs, and deadlines for a restricted class of threads. To use such features, the programmer is required to use unwieldy code constructs to create region-like areas of storage, associate them with execution scopes, and allocate objects from them. Further, the developer must ensure that they do not violate strict inter-region reference rules.

Unfortunately, it is difficult to determine manually how to map object instantiations to execution scopes. Moreover, if ordinary Java code is modified to effect instantiations in scopes, the resulting code is difficult to read, maintain, and reuse. We present a dynamic approach to determining proper placement of objects within scope-bounded regions, and we employ a procedure that utilizes aspect-oriented programming to instrument the original program, realizing the program's scoped memory concerns in a modular fashion. Using this approach, Java programs can be converted into region-aware Java programs automatically.

copyright by

Morgan G. Deters

2003

In fond memory of Jamie,
whose short life
will be long remembered

and for my parents,
without whom, indirectly, there would be no thesis,
and you would quietly ponder why anyone bothered
to bind blank pages together

Contents

List of Tables vii

List of Figures viii

Acknowledgments x

Preface xi

1 Introduction to Real-Time Java 1

1.1 Threading model 3

1.1.1 Synchronization 4

1.2 Features 5

1.2.1 Keeping time 5

1.2.2 Software access to execution platform 5

1.3 Tasks and event handlers 7

1.4 Memory model 7

1.5 Comments 8

2 Real-Time Java Memory Model 9

2.1 The RTSJ meets garbage collection 9

2.2 Uncollected storage areas 10

2.3 Nested storage scopes and instantiations 13

2.4 Issues in using scoped storage 14

3 Advanced Separation of Concerns **16**

3.1 Approaches to Separation of Concerns 16

3.1.1 Composition Filters 17

3.1.2 Adaptive Programming 18

3.1.3 Meta-Object Programming 19

3.1.4 Subject-Oriented Programming 20

3.1.5 Hyperspaces and Multi-Dimensional Separation of Concerns 21

3.1.6 Aspect-Oriented Programming 21

3.1.7 Intentional Programming 22

3.2 Applications of ASoC 22

3.3 Choice of AOP 24

3.4 AspectJ: a general-purpose aspect extension to Java 24

4 Motivation for Automatic Scope Assignment **25**

5 Scope Detection and Assignment **28**

5.1 Parameterization of Scope Analysis 28

5.2 Approach 29

5.3 Analysis techniques 30

5.4 Vertex granularity of *doesReference* 32

5.4.1 Scope determination 33

5.4.2 Propagation of liveness through *doesReference* graph 36

5.4.3 Target program instrumentation 38

5.4.4 Multithreaded target programs 39

6 Aspects **40**

6.1 Analysis aspects 41

6.1.1 Reference-probing aspect 42

6.1.2 Object death-probing aspect 43

6.1.3 Offline analysis 44

6.2 Runtime enforcement 44

6.3 Other aspect analysis techniques 45

6.4 Remarks . . . 46

7 Experimentation . . . 47

7.1 Experimental method . . . 47

7.2 Benchmarks . . . 48

7.3 Results . . . 49

8 Related Work . . . 55

8.1 Region-based memory systems . . . 55

8.2 Analysis Techniques . . . 56

9 Conclusion . . . 57

9.1 Future work . . . 57

Appendix A Source for Analysis Tools . . . 59

Appendix B Source for Instrumentation . . . 82

References . . . 90

Revision History . . . 98

Vita . . . 99

List of Tables

1 Java programming vs. real-time programming xiii

2.1 Legal references between storage areas as specified by the RTSJ 11

5.1 Constraint system size for benchmarks (size 10) using *allocation site* granularity to construct the *doesReference* graph. Section 7.2 contains more information on these benchmarks 38

7.1 Benchmark sizes: the number of objects and distinct allocation sites encountered during execution 48

List of Figures

1.1 Sample notation used for Java and RTSJ class diagrams 2

1.2 Java and RTSJ thread classes 3

1.3 RTSJ thread parameter classes 4

1.4 RTSJ time classes 6

2.1 (a) A Java program and (b) its RTSJ version 12

3.1 Overview of the Composition Filters method 18

3.2 An example runtime MOP event 20

4.1 The *doesReference graph* for the program of Figure 2.1(a), vertices augmented with object liveness information 26

4.2 Optimal RTSJ scope assignment for the program of Figure 2.1(a) with inter-object references superimposed 27

5.1 Overview of approach 30

5.2 Worklist algorithm for scope assignment 38

6.1 Advice implementing scopes in the example program of Figure 2.1 41

6.2 A sample enforcement aspect 45

7.1 Scoping assignments for objects in `mpegaudio` 51

7.2 Scoping assignments for objects in `raytrace` 51

7.3 Scoping assignments for objects in `javac` 52

7.4 Scoping assignments for objects in `jess` 52

7.5 Extra longevity of objects in `raytrace` using scoped memory 53

7.6 Extra longevity of objects in `mpegaudio` using scoped memory 53

7.7 Extra longevity of objects in `javac` using scoped memory 54

7.8 Extra longevity of objects in `jess` 54

Acknowledgments

I would like to thank my advisor, Dr. Ron K. Cytron, for his guidance on this research, for interesting conversations, and for my graduate education in general, and the other members of my committee, Dr. Christopher D. Gill and Dr. Aaron Stump, without whom, according to the dean, I would not graduate. I was very fortunate to receive their invaluable suggestions for revising this text.

I would like to thank Nicholas A. Leidenfrost, Matthew P. Hampton, Dante J. Cannarozzi, Conrad E. Warmbold, and Steven M. Donahue for assisting in the instrumentation of a Java Virtual Machine (JVM) to provide results herein presented, and I would like to thank Guy Steele, Jr. and Sun Microsystems, Inc. for providing our research group with that JVM. I should also thank all past and present members and affiliates of the Distributed Object Computing Group at Washington University for their friendship and stimulating discussions and/or arguments. Besides those mentioned above, this includes but is not necessarily limited to, in alphabetical order, Luther Baker, Kitty Balasubramanian, Sharath Cholleti, Angelo Corsaro, Delvin Defoe, Phil DiCorpo, Lucas Fox, Scott Gelb, Mike Henrichs, Chris Hill, Joe Hoffert, Frank Hunleth, Ramaprabhu Janakiraman, Yamuna Krishnamurthy, Anand Krishnan, Victor Lai, Dr. David Levine, Martin Linenweber, Balachandran Natarajan, Kirthika Parameswaran, Jeff Parsons, Michael Plezbert, Ravi Pratap, Rooparani Pundaleeka, Dr. Irfan Pyarali, Dr. Douglas C. Schmidt, Venkita Subramonian, Stephen Torri, Nanbor Wang, and Guoliang Xing. I must also acknowledge Peggy Fuller, Jean Grothe, Myrna Harbison, and Sharon Matlock, without whom the department would not function, and the staff of Computing Technology and Services, without whom the department would not compute.

I should dearly thank (in reverse alphabetical order this time) Derrick Moseley, Justin Levine, Jake LaBombarbe, Ravindra Das, and others for their insightful, interesting, and/or distracting discussions through the years. (I *should* thank them, so I am. Thanks to you all.)

And I would especially like to thank my parents, Dr. Donald and Lynn Deters, for their lifetime of support and encouragement. (And because if I did not they would be mad.)

The research represented in this thesis was sponsored by the Defense Advanced Research Projects Agency (DARPA) under contract F33615–00–C–1697.

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

Morgan G. Deters

Washington University in Saint Louis

May 2003

Preface

Traditionally, developers of software for real-time and embedded systems have resorted to low-level programming tools and implementation mechanisms during the development cycle. It is routinely argued that such techniques are needed due to several constraints and requirements on real-time and embedded systems development, including the following:

- The resulting software must have a small memory footprint. Both the code size and the amount of memory used during execution must be considered. In designing software for general-purpose platforms (desktop computers, for example), this concern is generally not regarded as significant next to the features, quality, and overall correctness of the resulting software. In an embedded system or other "special-purpose" platform, however, a small memory footprint may be a first-class engineering requirement, considered a vital part of the *correctness* of the system.

- In real-time systems, some (and perhaps all) logical code segments must be *reasonably bounded* in execution time; optimization of the worst case, rather than the average case, is essential. Latency in response time to user, network, and internally-generated events must be bounded to guarantee that a critical operation will complete at a particular time or will operate at a particular rate. Notably, this implies that any storage requests made from a code segment with real-time constraints must guarantee satisfactory completion within a particular execution window. This timeliness constraint is a matter of *correctness* in real-time systems.

- If the target platform is a device operating on a battery, it must budget its power consumption, and ideally it should adapt to low-power conditions by suspending non-critical, power-expensive operations. Software running on such systems thus has an additional responsibility to reason logically about very low-level, hardware-specific constraints, features not present in most of today's general-purpose software (excepting, of course, operating systems, some graphics applications, and similar modules).

Such adaptive logic can naturally be quite complex for advanced, flexible, and upgradeable embedded systems, such as recent Personal Digital Assistants (PDAs) on the market.

Software developers targeting real-time and embedded platforms are therefore forced to reason about these software concerns throughout their code, whereas developers of general-purpose software may relegate such matters to an operating system, a utility library, or, more generally, a layer of Distributed Object Computing (DOC) middleware [17] or virtual machine [48] capable of scheduling and managing resources. Several advantages are associated with handling such concerns in middleware:

- Portability is ensured across platforms supported by the middleware layer. Higher-level abstractions are available to developers, providing robust solutions designed and coded by experts and proven empirically.
- Code reuse is realized. This includes several benefits:
  - *Ad hoc* solutions provided within a group or company or even tailored for a specific project are often unnecessary and are better handled by using available implementations of design and architectural patterns [30, 13, 64].
  - Development time is reduced. The use of already-engineered, high-level abstractions frees developers from constantly dealing with low-level technical concerns.
  - Reuse of standard, high-level middleware abstractions has a large impact on code maintainability.
  - Commercial Off The Shelf (COTS) components built using the standard middleware layer may be utilized.
- Communications are standardized. Code that targets a layer of middleware can communicate with other software written by different developers but for the same middleware platform, and inter-vendor middleware communication is often specified by higher-level standards [56, 59].
- Bugs in production code are reduced. By targeting a well-designed, well-tested middleware package, the number of bugs in shipped software can be reduced. In critical real-time and embedded systems, this can be a serious matter of human safety and system security.

A layer of middleware that supports real-time and embedded concerns is thus needed. Much research and development effort has been and continues to be directed toward this goal [17, 56], and has brought high-level constructs and strong portability to real-time and embedded developers. Java[1] [33, 2] provides high-level constructs at the language level, supported by the JVM, and higher-level, distributed concerns are available in class libraries [16, 69, 70, 45]. Java has indeed enjoyed wide acceptance, from applications deployed on large servers to those deployed on small, hand-held devices; however, the real-time community has largely ignored Java for a variety of reasons.

Table 1 coarsely juxtaposes Java and real-time programming. Naturally, this is an oversimplification, as the truth about system requirements depends upon the particular software in question. Still, it is useful to note that while Java supports high-level programming constructs largely taken for granted in much of the programming world, including automatic storage management, objects, and retargetable bytecode, real-time programmers have concerns that have traditionally kept them from enjoying these language and runtime features and have coded tailored memory allocators and hard-coded program logic and concurrency scheduling in an attempt to keep memory footprint small and execution time predictable. As research continues in this area, however, high-level features will find their way into real-time programmers' toolkits because of their enormous benefits when building large, complex systems that must scale, must be quickly and economically developed, must be maintainable, and must be retargetable.

Table 1: Java programming vs. real-time programming.

Java Programming	**Real-Time Programming**
High-level	Low-level
Popular	Specialized art
Object-Oriented	Manually-optimized
Automatic memory management	Specialized memory allocators
Reusable components	Tailored components
Flexible, adaptable	Hard-coded behavior
Scalable	Doesn't scale
Portable	Hardware-dependent
Optimize the average-case	Reasonably-bounded worst-case is critical
Memory use varies	Memory request latency is critical
Coarse control over concurrency	Fine control over concurrency

Recently, an expert group issued the Real-Time Specification for Java [10] (RTSJ) in an attempt to reconcile Java and real-time programming differences. This document describes a semantic addendum and supplemental class library for Java designed to support real-time applications. There are no changes to the

[1] Java is a registered trademark of Sun Microsystems, Inc.

base language; thus, compilers and other program-development tools are largely unaffected by the RTSJ's mechanisms.

But real-time and embedded systems developers face other challenges as well. Besides the advantages to high-level reasoning about their programs, simplicity, and reusability afforded by traditional, standardized middleware, there are "concern complexities" to be addressed. Chapter 3 introduces Advanced Separation of Concerns (ASoC) technologies and their applicability to real-time and embedded systems software. In particular, proper use of these technologies on top of standard middleware can afford code reuse and design simplicity beyond that which is attainable with object middleware alone.

The RTSJ touches many areas of the Java programming language, including threads, task prioritization, event handling, synchronization, and memory management. We introduce these different areas in Chapter 1, but will focus primarily on the aspects of the RTSJ concerned with storage management, described in depth in Chapter 2.

After the aforementioned discussion of ASoC in Chapter 3, we identify a difficulty in using the RTSJ memory system in Chapter 4 and propose a solution in Chapter 5. We tie together this solution and ASoC in Chapter 6 and describe our implementation. Experimental results for automatically assigning objects to RTSJ scopes for ordinary Java programs are given in Chapter 7. Related work and concluding remarks are provided in Chapter 8 and Chapter 9, respectively.

The main contributions of this thesis are to describe the automated harvesting of potential RTSJ scoped memory areas in ordinary, single-threaded Java programs with the flat Java memory model, to provide some early evaluation of the objects that can take advantage of these memory areas instead of relying upon the standard garbage collected heap, and to argue the relevance of ASoC for real-time and embedded systems software development. This thesis and the research it describes are concerned primarily with Java programs. However, we believe our technique is applicable to the more general problem of assigning objects to stack frame-associated memory regions and, therefore, applicable to other systems in which region-based memory management has been deployed (see Chapter 8 for a brief discussion of these systems).

Chapter 1

Introduction to Real-Time Java

The Real-Time Specification for Java [10] (RTSJ) adds various real-time components to the Java[1] [33, 2] programming language. In particular, it specifies:

- A new set of classes constituting a real-time Application Programming Interface (API), comprising the `javax.realtime` package;
- Time and scheduling guarantees of an RTSJ-conforming platform;
- Programmatic mechanisms that must be explicitly used by a programmer for the platform to provide these guarantees; and
- Those implementation decisions that are left to implementation vendors

This chapter introduces each of these areas, discusses its relationship to similar constructs in core Java, surveys the relevant API, and briefly comments on the approach. The specification's suitability for real-time programming is stressed. The RTSJ proposes no modification or extension to Java syntax.[2] This approach is evaluated in a concluding section.

[1] Java is a registered trademark of Sun Microsystems, Inc.

[2] Other real-time proposals for Java have included syntactic modifications to the core language; see, for example, the Real-Time Core Extensions [61] (RTCE).

This chapter uses class diagram similar to the Unified Modeling Language [57] (UML) for indicating Java and RTSJ class relationships. In particular, names of interfaces, abstract classes, and abstract methods are shown in an *oblique* font, public methods are denoted with a *plus* (+), and protected methods are denoted with a *hash* (#). Java's `implements` relation is shown with a dotted line with a closed arrow in the direction of the relation, and Java's `extends` relation is shown with a solid line and an closed arrow in the direction of the relation. RTSJ interfaces and classes are distinguished from those available in core Java by a thick outline. A visual summary of this notation is shown in Figure 1.1. Please see [57] or [62] for further discussion of modeling object designs.

Note that these diagrams and this chapter are *intentionally incomplete*. The class diagrams and associated text are intended only to provide a general overview of the RTSJ's object architecture and may exclude significant portions of the API. See [10] for additional information about the RTSJ and for complete API documentation.

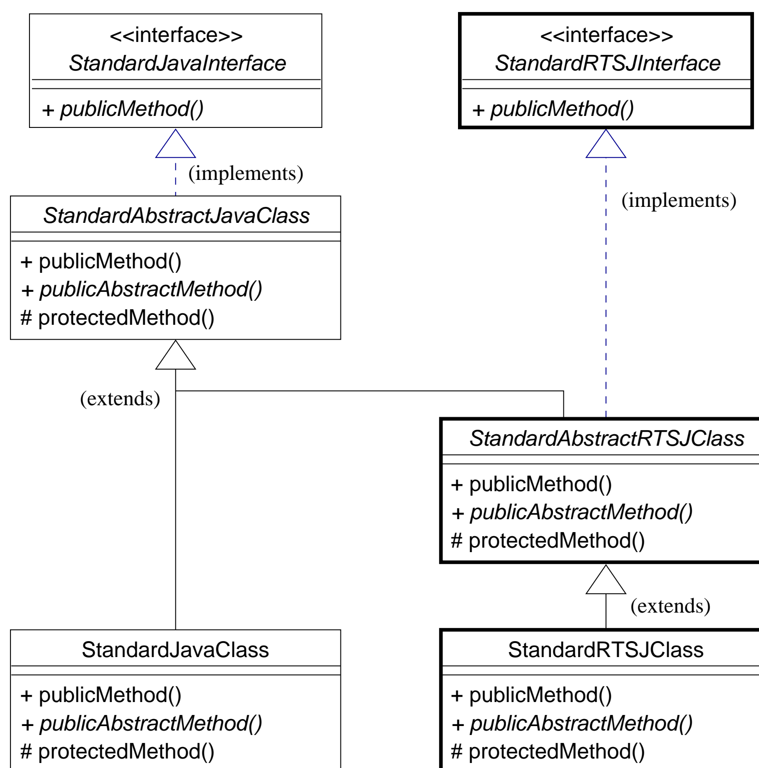

Figure 1.1: Sample notation used for Java and RTSJ class diagrams.

An editorial note on RTSJ versions: The Real-Time Specification for Java was first published in 2000 as a book [10], and this has been called the "first edition" of the specification. A revision, the "final specification

version 1.0," was completed in November 2001, and it is available online [11]. The RTSJ reference implementation from TimeSys [73], development of which took place between the release of the two specifications, implements some of each but neither in full. Except where noted, we refer to the original, published version found in [10].

1.1 Threading model

In Java, the creation and management of threads is a relatively simple task: a programmer needs only to subclass the `java.lang.Thread` object and provide code for concurrent execution in a `run()` method.[3] Concurrency is then triggered via the `start()` method. In the RTSJ, this simplicity is largely kept, yet additional parameters may be specified to control the way in which a given thread is scheduled.

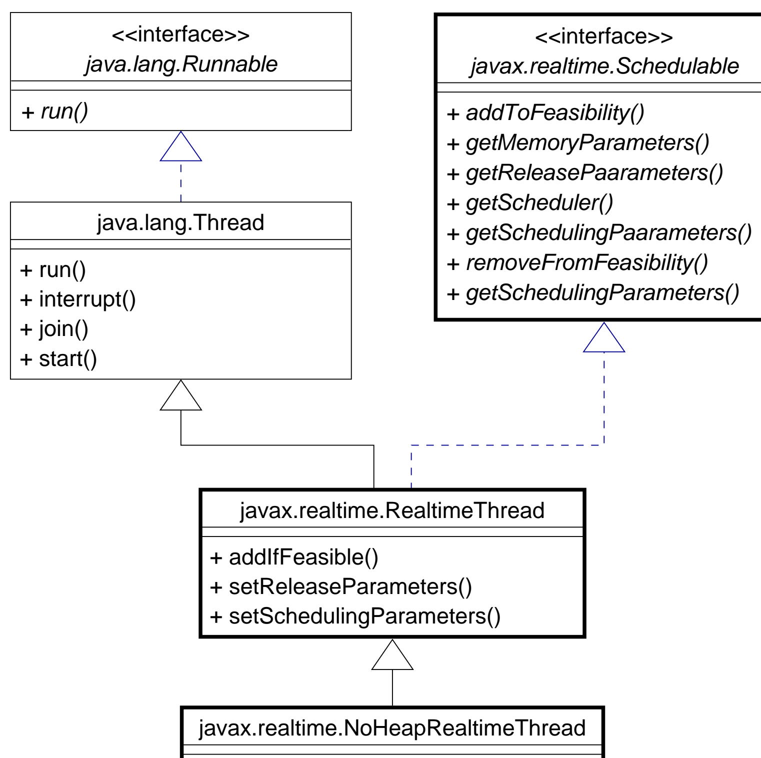

Figure 1.2: Java and RTSJ thread classes.

Figure 1.2 shows Java and RTSJ thread classes. There are three parameter kinds associated with real-time threads: the `SchedulingParameters` type specifies information about how a thread should be

[3]This is not the only way to spawn a thread in Java; one can, for example, subclass `java.util.Timer` instead, or provide the `Thread` constructor with a `Runnable` object intended to specify the execution behavior of the thread.

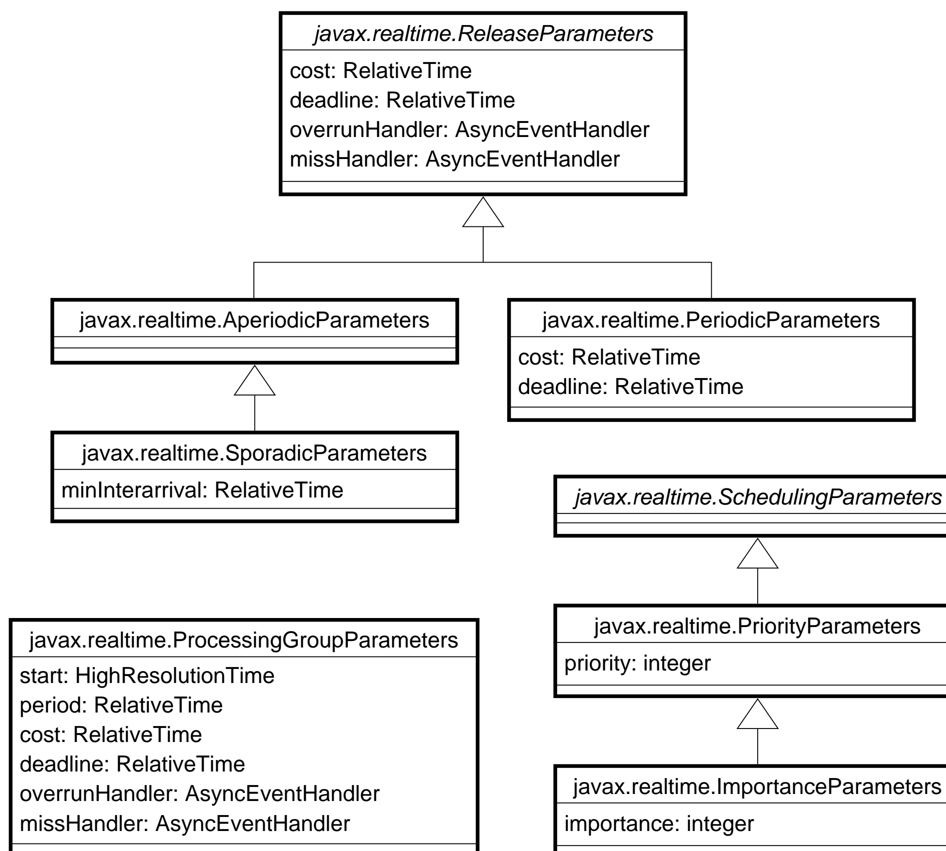

Figure 1.3: RTSJ thread parameter classes.

scheduled relative to other schedulable tasks; the `ReleaseParameters` type specifies how the a thread should be scheduled with respect to time; and the `ProcessingGroupParameters` type specifies a limit to the amount of resources that a group of threads may consume. These parameter types are diagrammed in Figure 1.3 along with all of their subclasses.

1.1.1 Synchronization

Like threads, Java synchronization is built into the core language instead of being included as part of an add-on library or provided as an operating system-dependent feature. Each object in Java has an associated recursive lock that can be acquired and released in a scoped manner. Synchronization is specified by Java's `synchronized` construct. At first this seems particularly suited to real-time software development that needs this functionality; unfortunately, however, this built-in locking mechanism can be inappropriate for real-time software that requires the ability to attempt to acquire the lock in a nonblocking fashion.

Doug Lea has provided additional locking primitives not present in standard Java that can help in this regard [45]. (These add-on classes are coded in pure Java and can operate on any compliant Java2 v1.2 Java Virtual Machine (JVM), including RTSJ-compliant JVMs.)

Every Java object can be used as a monitor. That is, in addition to locks, Java objects have condition variables. Condition variables provide a facility for simple condition notification between threads without polling (busy-waiting).

For additional information about issues in Java concurrent programming, see [44]; for additional information about RTSJ synchronization, see [10] chapters 3, 4, and 6.

1.2 Features

The RTSJ also contains some "utility" features that are not part of any particular subsystem. These features include time measurement and environment access.

1.2.1 Keeping time

The RTSJ provides time classes for reflecting about time. They thus perform a similar function to Java's standard `java.util.Date` type, with the exception that these APIs are capable of nanosecond precision (`java.util.Date` is limited to millisecond precision), can more easily represent relative times through the `RelativeTime` class, and can more easily represent frequencies through the `RationalTime` class.

The class hierarchy of RTSJ time classes is shown in Figure 1.4. The `Clock` class is not shown; this class provides a real-time clock interface capable of scheduling timers.

1.2.2 Software access to execution platform

The RTSJ provides a `POSIXSignalHandler` class capable of scheduling programmatic responses to standard POSIX signals (as SIGINT or SIGSEGV) on platforms that support such signals. By calling the static

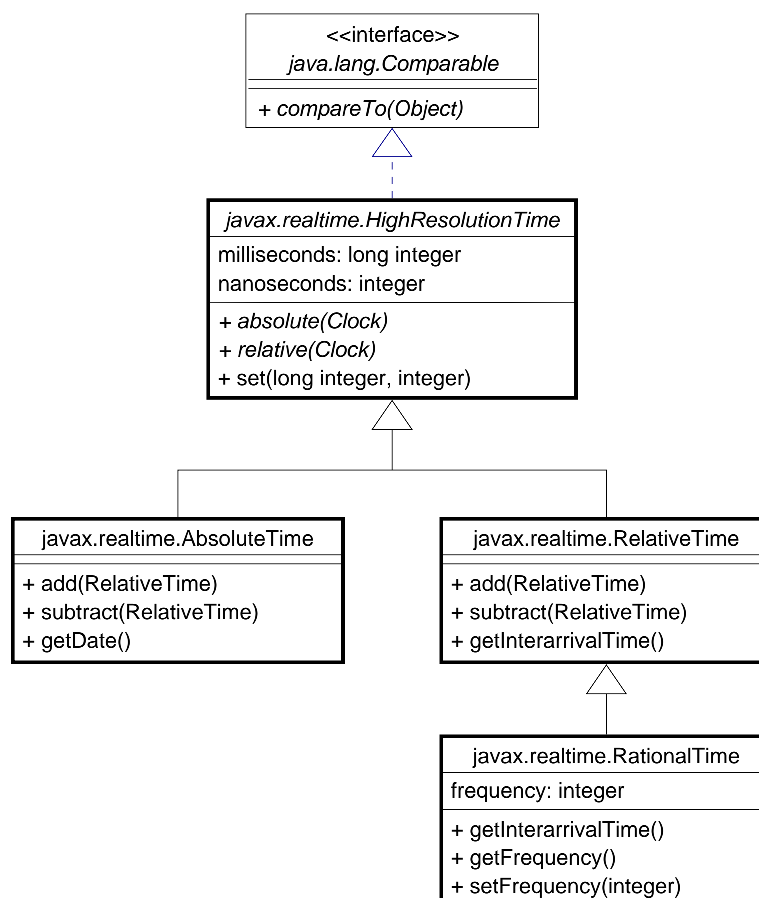

Figure 1.4: RTSJ time classes.

method `addHandler()` on `POSIXSignalHandler`, one can associate a list of asynchronous event handlers to a particular signal; a POSIX signal-handling thread runs in the RTSJ-compliant JVM and dispatches to the appropriate handlers when the underlying platform delivers a signal.

The RTSJ unfortunately under-specifies the handling of POSIX signals by an RTSJ-compliant JVM. It is unclear whether the `addHandler()`, `removeHandler()`, and `setHandler()` routines, which are all synchronized static methods of the `POSIXSignalHandler` class, could block for a period of time if the internal list of handlers is currently locked (as it might be, for example, when another thread is in the process of adding a signal handler).[4] The use of this feature in real-time software is thus limited.

[4]This comment is based on [10]. The version of the specification found in [11] corrects this issue—in part—by removing the keyword "`synchronized`" from the methods' signatures. (In this case, the reference implementation [73] conforms to [10].) Still, no statement is made on synchronization within each method's implementation, which would appear to defer to the RTSJ JVM implementor any decision about thread synchronization and implementation in this case. Such an implementor could either handle the dispatching mechanism largely inside the JVM, or rely more heavily upon the host operating system to dispatch to user code as appropriate. In either case, however, an RTSJ-compliant JVM running on a host operating system that supports POSIX signals must support multiple dispatches for a single POSIX signal, in some form, regardless of the host operating system's support for such a feature.

1.3 Tasks and event handlers

The RTSJ allows the developer to specify tasks run at a particular time (RTSJ Timers) and define new types of asynchronous events that can be handled. Once an event has been identified and an `AsyncEvent` instance describing it has been discovered or generated, it can be associated with `AsyncEventHandler` objects. Each `AsyncEventHandler` represents a particular programmatic response to the event and has an associated runtime cost and scheduling parameters, rather like a `RealtimeThread`.[5] However, RTSJ-compliant JVMs are designed with the expectation that large numbers of `AsyncEventHandler` objects may exist in the system at any one time, yet few of them are active simultaneously.[6] Violating this assumption can, then, be expected to result in a weakening of real-time guarantees.

1.4 Memory model

In Java, memory is segregated into the *stack*, a set of *registers* ("local variables"), and the *heap*. Stack elements and registers contain 32- and 64-bit floating points, object references, and return addresses. The JVM performs nearly all computation on the stack.[7] Objects only exist in the heap; the Java model does not allow for stack-allocated objects. Object storage is returned to the system with the aid of a *garbage collector*, which can detect objects that are no longer in use.

In making no changes to the Java language, the RTSJ keeps the standard Java memory heap. However, the real-time development community as a whole is unwilling to accept garbage collection due to its runtime cost. Even more troublesome is the unpredictability of garbage collection—garbage collection schemes cannot, in general, guarantee that they will reclaim space or how long it will take to find space that may be reclaimed. If a program executes a `new` operation to allocate space for a new object, the garbage collector may need to be invoked at that point to reclaim enough space for the allocation to proceed. Thus garbage collection can be infeasible for real-time systems that must absolutely guarantee that they will meet their deadlines.

---

[5] This similarity is structurally represented in the class structure: `RealtimeThread` and `AsyncEventHandler` both implement the `Schedulable` interface.

[6] This expectation is stated explicitly in [10], page 129, and also in [11], page 183.

[7] The IINC instruction is a notable exception; it performs computation in a register.

The RTSJ does not mandate real-time garbage collection and, in fact, dismisses it. The expert group argued that, as of the time of finalization of the specification (November 2001), no garbage collector was suitable for all real-time applications.[8] Numerous proposals exist for real-time garbage collection schemes, including , and these choices are certainly open to RTSJ JVM implementors.

Though the specification makes no strict guarantees for threads that can access garbage-collected storage, RTSJ-compliant JVM must support a garbage-collected heap. It also provides other memory areas that reduce the need for incurring garbage collection overhead. Specifically, it provides an *immortal memory area* for objects that are known never to become dead and *scoped memory areas*—the cousin of stack allocation, scoped memory allocation allows that the program free itself of the need for garbage collection by meticulously placing objects into scopes outside of which they are known to be dead. However, as demonstrated in Chapter 2, this approach has serious consequences in the readability and understandability of code, and the clean separation of design goals in general. Chapter 2 also gives an in-depth description of the RTSJ memory model.

1.5 Comments

The Real-Time Specification for Java [10] brings real-time programming to Java. Since it makes no modifications to the underlying, popular Java language, it affords real-time developers the full range of high-level language features taken for granted in the programming world at-large, including strong type safety, an object-oriented style of programming, and automatic store management. However, without enhanced syntax to specifically handle the concerns of real-time and embedded systems, RTSJ code can easily become verbose, unwieldy, and unmanageable.

[8] See [10] page 59 or [11] pages 76–77.

Chapter 2

Real-Time Java Memory Model

The work presented here is motivated partly by the RTSJ specification, which itself was motivated by the desire to make Java attractive to developers of real-time applications. While the RTSJ could itself be subject to criticism, for the purposes of this work we adopt its views concerning how Java can be used while still providing real-time guarantees for critical tasks. In this chapter we discuss the relevant portions of the RTSJ for our work.

2.1 The RTSJ meets garbage collection

As discussed in Chapter 1, the RTSJ [10] covers many issues related to real-time programming. We will now limit ourselves to matters of storage and threads and their relationship to the garbage collector.

Although research has addressed the viability of real-time garbage collection to some extent [54, 18], it is still questionable [55] whether a collector and allocator can *always* provide storage for an allocation request (*i.e.*, a `new` command) in time proportional to the size of the request. Consider the case where the heap is full except for a few dead objects. All exact collectors require some form of marking phase to discover objects that are still live. Generational collection [79] can limit the extent of marking but at the expense of ignoring potentially dead objects in older generations. The result is that the time taken to perform a `new`

command cannot be reasonably bounded if garbage collection is required to liberate storage to satisfy the request.

Such bounds are essential if the thread requesting the storage must adhere to an execution-cost budget. In the RTSJ, tasks' budgets are submitted to the scheduler so that the *feasibility* of scheduling the tasks can be determined. Currently, such decisions are based on Rate-Monotonic Analysis (RMA) [50, 65]. If the garbage collector causes a thread to experience unbudgeted delay, then that task *as well as others* could miss their deadlines.

As a result, the RTSJ insists that threads for which scheduling criteria are taken seriously—the appropriately-named `NoHeapRealTimeThreads` (NHRTTs)—cannot allocate storage in the garbage-collected heap. In fact, such threads cannot even *reference* an object in the garbage-collected heap, since it is possible that heap objects could be locked during a garbage-collection cycle, causing the referencing thread to experience unbudgeted delay.

2.2 Uncollected storage areas

The RTSJ defines two kinds of storage areas for use by NHRTTs:[1]

`ImmortalMemory` is a singleton memory area all of whose component objects' lifetimes are that of the entire program.

`ScopedMemory` is an area of storage whose constituent objects' lifetimes are collectively tied to a particular execution scope.

Both of the above storage areas are completely ignored by garbage collection in an RTSJ-compliant JVM; objects allocated in the singleton immortal memory area are not collectible, and objects allocated within a `ScopedMemory` area are collected *en masse* when the threads that could access the memory area exit the associated scope.

[1] We do not discuss the `PhysicalMemory` classes of the RTSJ here; such classes permit access to specific locations of memory and can also be scoped or immortal, but they are not used in our work.

There are two types of scoped memory areas defined in the RTSJ. Each derives from the class `ScopedMemory`:

`LTMemory` offers a *linear-time memory allocator*. That is, allocation requests in `LTMemory` scoped memory areas take time proportional to the size requested.

`VTMemory` offers a *variable-time memory allocator*. No guarantee is made of such an allocator's typical or worst-case timing characteristics.

jRate [20], a `gcj`-based ahead-of-time compiler implementation of the RTSJ, offers an additional type of `ScopedMemory` area:

`CTMemory` offers a *constant-time memory allocator* from the point of view of the `new` operation; the memory is initialized to a 0 bit-pattern when the `CTMemory` is first created, rather than incrementally as objects are instantiated inside of it.

To associate a `ScopedMemory` area to a particular execution scope and to be permitted to allocate from it, an RTSJ program calls the `enter` method on a `ScopedMemory` instance, as in the code of Figure 2.1(b). The `enter` method takes a single parameter—a `Runnable` object whose `run` method becomes the execution scope for the scoped memory area. A reference count of threads currently accessing a given scoped memory area is kept by the JVM, and when all of these threads exit their scopes' top-level `run` methods, the objects within the memory area are known to be collectible—no live objects in the system can reference them because of particular reference constraints established by the RTSJ and shown here in Table 2.1.

Table 2.1: Legal references between storage areas as specified by the RTSJ.

	Reference to Heap	Reference to Immortal	Reference to Scoped
Heap	Yes	Yes	No
Immortal	Yes	Yes	No
Scoped	Yes	Yes	Yes*

* If to an object in the same or outer scope

In accordance with Table 2.1, code executing in an RTSJ thread may not access any storage that could be reclaimed while the thread is still accessing it. It is safe for any storage area to access immortal memory. Also, a thread can access any scope that it has *entered*—any scope whose reference count was bumped on behalf of that particular thread. To guard against potential "dangling references" to objects that

```java
class Example {
  void zero() {
    RefObject E = new RefObject();     *
    one(E);
  }

  void one(RefObject E) {
    Object D = two(E);
  }

  Object two(RefObject E) {
    RefObject D = new RefObject();     *
    Object F = new Object();           *

    RefObject A = new RefObject(D),    *
              B = new RefObject(E),    *
              C = new RefObject(F);    *

    D.ref = E;

    refTemporarily(A, B);
    refTemporarily(B, C);
    refTemporarily(C, A);

    Object G = new Object();           *
    refTemporarily(E, G);

    return D;
  }

  void refTemporarily(RefObject source,
                      Object target) {
    source.ref = target;
    source.ref = null;
  }
}

class RefObject {
  Object ref;
  RefObject() { }
  RefObject(Object o) { ref = o; }
}
```

(a)

```java
import javax.realtime.*;

class RTExample {
  void zero() {
    final RefObject E = new RefObject();
    // memory area of 1 kilobyte
    new LTMemory(1024,1024).enter(
      new Runnable() {
        public void run() {
          one(E);
        }
      });
  }

  void one(final RefObject E) {
    final ScopeReturnValue returnObj =
      new ScopeReturnValue();
    // memory area of 1 kilobyte
    new LTMemory(1024,1024).enter(
      new Runnable() {
        public void run() {
          returnObj.set(two(E));
        }
      });
    Object D = returnObj.get();
  }

  Object two(RefObject E) {
    RealtimeThread thisThread =
      RealtimeThread.getRealtimeThread();
    ScopedMemory thisScope =
      (ScopedMemory)thisThread.getMemoryArea();
    MemoryArea
      outerScope = thisScope.getOuterScope(),
      outerOuterScope =
        ((ScopedMemory)outerScope).getOuterScope();
    RefObject D = (RefObject)
      outerScope.newInstance(RefObject.class);

    Object F = new Object();

    RefObject A = new RefObject(D),
              B = new RefObject(E),
              C = new RefObject(F);

    D.ref = E;

    refTemporarily(A, B);
    refTemporarily(B, C);
    refTemporarily(C, A);

    Object G =
      outerOuterScope.newInstance(Object.class);
    refTemporarily(E, G);

    return D;
  }

  void refTemporarily(RefObject source,
                      Object target) {
    source.ref = target;
    source.ref = null;
  }
}

class ScopeReturnValue {
  Object obj;
  void set(Object o) { obj = o; }
  Object get() { return obj; }
}

// RefObject definition remains the same
```

(b)

Figure 2.1: **(a)** A Java program and **(b)** its RTSJ version. Allocation sites are marked by an asterisk (`*`) in **(a)**.

have been reclaimed, inter-scope references are regulated. Scoped storage can contain references to the garbage-collected heap, but if an NHRTT tries to access any cells in the heap, an exception is thrown.[2]

2.3 Nested storage scopes and instantiations

While such acquisition by a thread is the mechanism for bumping the reference count of a `ScopedMemory` area, it is important to understand the intent of `ScopedMemory` in the RTSJ. Garbage collection is avoided in such areas, so it is essentially left to the programmer to determine the effective lifetime of all instantiated objects and to allocate such objects in the appropriate scope (respecting criteria 1 and 2 discussed in Chapter 1). If `ScopedMemory` creation and deletion were free, then ideally a program would allocate them to hold objects with coinciding lifetimes: such objects are then freed collectively when the scope is exited and its memory area is deleted.

This bias toward "many scopes" may encounter performance difficulties if `ScopedMemory` creation, deletion, or other necessary accounting is expensive. In this work, we take the view that more, precise scopes are better than fewer, conservative scopes.[3] We draw from research on *regions* [75, 74, 31] and tie a scoped area's life to the pushing and popping of stack frames in response to method invocations and exits. Some regions work is more general, as the beginning and ending of a frame could correspond to any reasonable pair of stack frames. For our present work, we tie the life of a scope to that of a single stack frame. Thus, when any method is entered, we can open a scope, and that scope exists for the lifetime of that method's frame. When the method returns, and its associated frame is popped, the storage scope is deleted.

In our automated method for using scopes, we rewrite object allocation sites (`new` commands) to cause the instantiated object to be allocated at a particular location on the call stack, where this location is determined by the analysis we detail in Chapter 5. Returning to the example in Chapter 1, variables `A`, `B`, `C`, and `F` would be allocated in their own allocating method's associated `ScopedMemory` area. Since object `D` is returned to its allocating method's caller, however, the instantiation of `D` must be moved back one frame, so its allocation site can be rewritten as described in Chapter 5 to allocate `D` in method `one`'s stack frame. Similarly, object `G` must be moved back two frames due to the reference of `G` by `E`.

[2] The RTSJ does offer a weaker form of a real-time thread, `RealtimeThread`, that can access such cells, but for which no scheduling guarantees can be made because of the issues discussed in Section 2.1.

[3] Generally speaking, this is true for runtime memory footprint. However, as mentioned, the generation of many `ScopedMemory` areas can potentially increase the footprint and execution time of the program.

Our instrumented `new` commands still execute where they were originally specified, but code is inserted to cause the storage for each allocation to be obtained from the appropriate scope. Object `E` is allocated in frame 0, as specified in the original program.

2.4 Issues in using scoped storage

Considering the rules of Table 2.1, we observe that:

- Marking up the source code for a class with explicit scope-allocation gestures (as in Figure 2.1(b)) will hamper readability and reuse of that class.
- Library classes provided by different vendors often coexist in a system in which they must interoperate; to promote separation of concerns, portability, and reusability in object-oriented systems, these separate products should not be required to know of and react to each other's individual memory requirements.

Furthermore, it is often difficult to determine the best scopes for object instantiation for the following reasons:

- The answer depends on how objects can reference each other and the effective lifetime of objects. As shown in Figure 2.1(a), object references need not persist to affect object allocation. The references from `A`, `B`, and `C` are all temporary, but the programmer who uses the RTSJ must ensure that each such access is safe according to the rules given in Table 2.1.
- Similarly-lived objects often unrelated in design *accidentally* become collectible at the same time due to their placement in other, aggregate structures or the behavior of a common thread that uses them.
- Java code is generally not written with object deallocation in mind (as code in a non-garbage collected language like C++ may be), and it is therefore not always reasonable to pick out places in the code where finalization and deallocation can occur. Stated another way, it is often difficult to determine object ownership. While this is a problem shared with C++ development, C++ code is typically written to handle such memory concerns, while legacy Java code is not.
- Allocation and deallocation events may not be clearly or inherently scoped; two objects of a given type or instantiated in a similar way may be live a very different amount of time. Standard library utilities (*e.g.*, Java's `Collection` classes) may be used in different ways with different associated liveness.

- It is unclear how best to communicate scoped-memory requirements between class libraries from different vendors, or between a third-party library and the library user.

Based on these observations, we will present our solutions for determining scopes for objects in Java programs and ideas for transforming them into scoped memory-aware RTSJ programs. First, however, we survey Advanced Separation of Concerns (ASoC) techniques and how, in general, they apply to real-time and embedded systems software development.

Chapter 3

Advanced Separation of Concerns

Since the popularization of object-oriented programming (OOP), a number of "post-object" technologies have been considered, including *generic programming* [5, 1, 12], *generative programming* [21], *computational reflection*[1] [51], and the broad category of *Advanced Separation of Concerns (ASoC)*. Informally, ASoC provides a software developer with a method of system decomposition more powerful than OOP provides alone. Many different technologies make up ASoC.

Section 3.1 attempts to demystify the large collection of technologies grouped under the banner of ASoC. Section 3.2 gives a brief discussion of the wide array of applications for this technology. Section 3.3 defends our selection of Aspect-Oriented Programming (AOP) for our research in particular, and Section 3.4 serves as a tutorial for the AspectJ suitable for understanding the later content of this work.

3.1 Approaches to Separation of Concerns

Different ASoC techniques define different programmatic abilities to reflect upon and modify the behavior of a system; in this way, ASoC can be viewed as a "re-abstraction" of reflection, allowing a programmer to apply reflection at a higher-level in a refined way. To make this concept more concrete, we offer the following sample of ASoC techniques.

[1] Of course, reflection is not a "post-object" technology *per se*, as it existed in various forms before the rise of OOP. Its use (and misuse) in object-based languages, however, has been the subject of much study [22, 33].

- The *Composition Filters (CF) model* [8] is designed to allow a program to respond to inter-object message passing in a modular fashion.
- *Adaptive Programming (AP)* [46, 58] allows a runtime system to reflect upon class relationships and conveniently keep a loose coupling between different classes.
- *Meta-Object Protocols (MOPs)* [41, 32] offer a refined form of reflection that focuses on modifying and reacting to object behavior at runtime or structurally reflecting upon code at compile time.
- *Subject-Oriented Programming (SOP)* [36] is a method of object decomposition inspired by the perspective of the consumer of a class rather than the class's notion of its own place in the system.
- *Hyperspaces* [71] facilitate the coherent composition of separately-designed or implemented object classes.
- *AOP* is a category of related techniques providing facilities to structurally modify object classes and respond to events at nominated "join points" in code execution or dataflow.
- *Intentional Programming (IP)* [67] provides a modular, natural way for a programmer or set of programmers to manufacture a software system.

The following sections briefly discuss each of these techniques in turn, discussing the relevance of each in the larger world of ASoC and providing references for further study. This survey focuses on the *technique*, not any particular *language* implementing the technique; yet where it serves for purposes of clarity both will be described.

3.1.1 Composition Filters

The Composition Filters (CF) model allows input and output message filters to be placed on objects in a flexible manner. In this view, each object has an *interface*, consisting of *input* and *output filters*, with which it interacts with the outside world, and an *implementation*, which encapsulates object behavior but is subject to filters when interacting with other objects.

Figure 3.1 depicts this object architecture. Each incoming message must pass a bank of *input filters*, each of which could cause the message to be blocked, diverted to another object, or modified in some way.

Any message sent from the object must pass through a bank of *output filters*, with similar functionality. Composition filters are available for Java using ComposeJ [76], which implements composition filters using inlining.

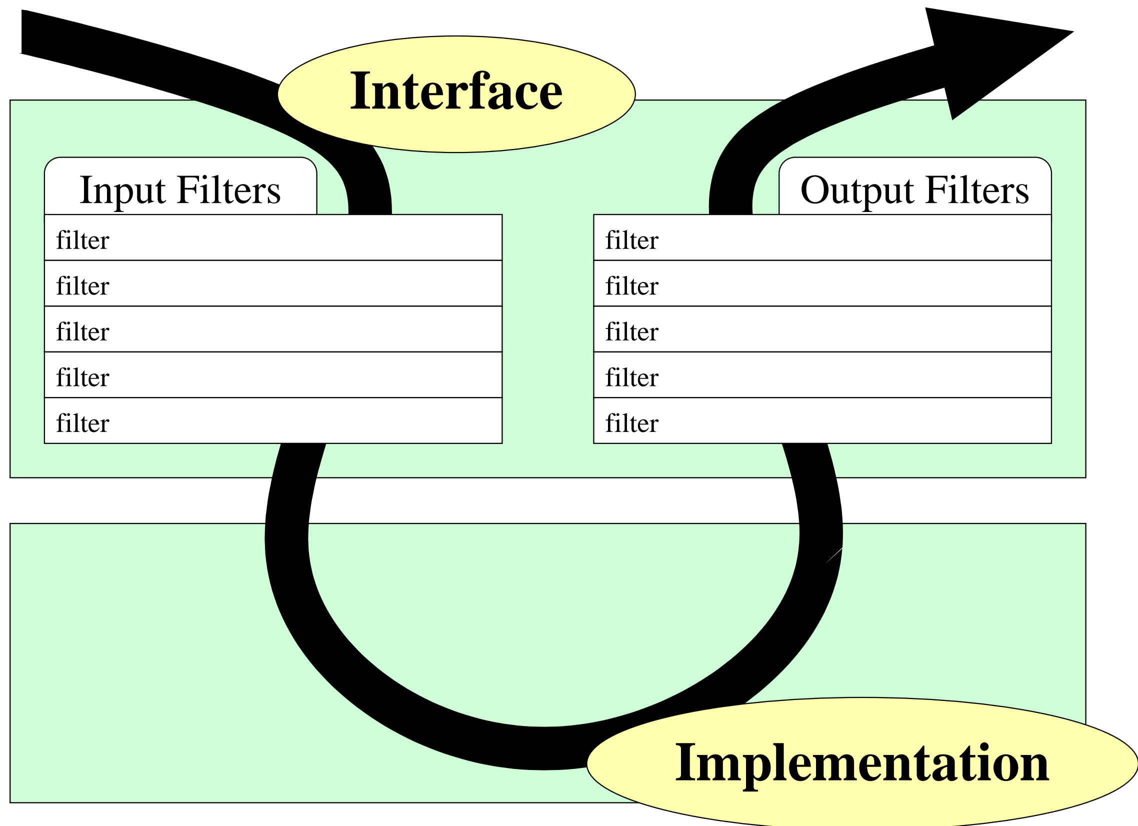

Figure 3.1: Overview of the Composition Filters method.

3.1.2 Adaptive Programming

According to the *Law of Demeter* [47], a programming style rule for object-oriented systems, a class should not rely upon the structure of other classes in the system except for an immediate set of neighbors, called its *preferred suppliers*. This is, in general, a good rule of thumb; experience in object-oriented code maintenance overwhelmingly demonstrates that large, highly-coupled object systems are often brittle and small modifications or extensions are often subject to compilation complaints from the type system or exotic run-time failures.

However, programming strictly according to the Law of Demeter can lead to many tiny accessor methods to navigate the class hierarchy. In some systems, it also leads to long accessor chains; for example, one might write the following in Java:

```
( (Expression)getTree().getRoot().getLeftChild() ).getRHS().getConstantValue();
```

Unfortunately, writing code as above leads to an increased coupling between classes, the exact problem the style rule is attempting to avoid; in effect we have only shifted the problem with no substantial improvement.[2]

Adaptive Programming (AP) [58] gives the developer the ability to decouple their object designs cleanly according to the Law of Demeter. The overall structure of the class hierarchy is discovered by the runtime system in the form of a UML-like *class graph*, and programmatic traversal of this structure is permitted using the *adaptive visitor* design pattern [46] and a *traversal strategy*. The traversal strategy specifies what traversals are "interesting" to the program at a particular point and notifies the adaptive visitor for each of these nominated points. Thus, program code that implements such operations as collecting together the leaves of a complex tree structure or navigating a complex reference graph to implement object persistence is completely decoupled from the class structure and handled—adaptively—at runtime. Thus, additions of new classes or changes to the program's class inheritance or composition structure does not invalidate the traversal code.

3.1.3 Meta-Object Programming

Meta-Object Protocols (MOPs) provide the ability for a program to reason about itself. Like reflection, there are additional levels of interpretation, each of which operates on the levels below. While MOPs vary in their scope, power, and usability, MOPs for object-oriented systems provide *meta-objects* attached to other objects or control or data flows and provide some ability to *intercept* base operations in the code and "jump" to the relevant meta-level meta-object.

Figure 3.2 shows an example of such a MOP interception for a received message on an object. In this case, the meta-object allows the message to be passed through, possibly after altering it. A meta-object can also block such messages, handle them itself, or dispatch them to another object entirely.

[2] Although direct application of the style rule may provide some benefit, the problem with such accessor chains is their brittle nature: any change to any of the classes along the change could require large-scale code refactorization elsewhere in the system.

Meta-object programming can, in general, be seen as a foundational technique with which many other ASoC techniques can be implemented.

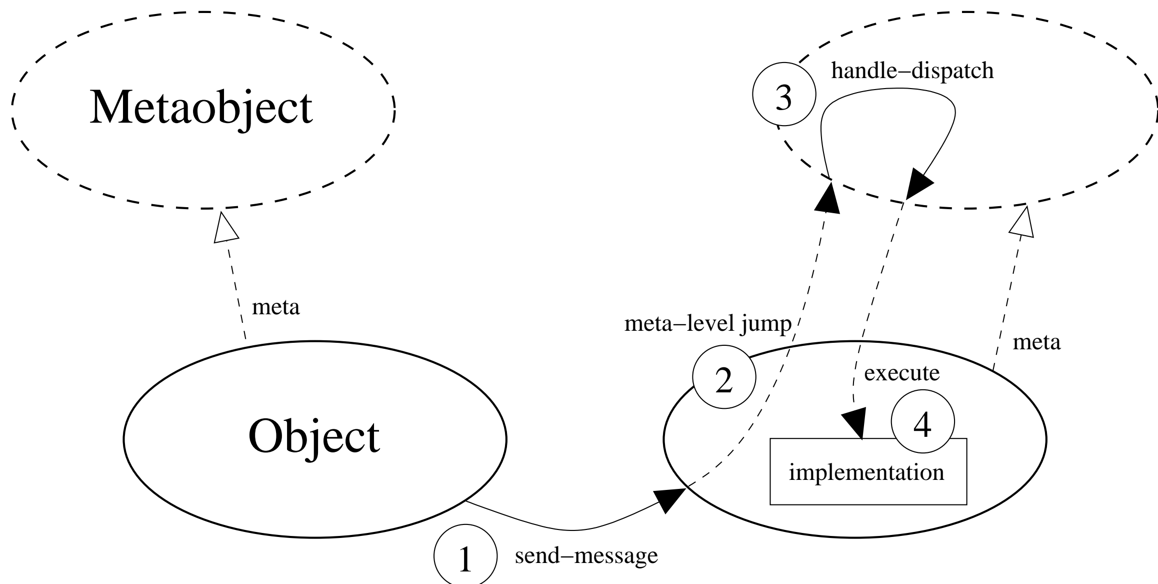

Figure 3.2: An example runtime MOP event. Closed arrows indicate flow of control; open arrows indicate inter-object references. First, a message is sent between objects. This causes a jump to the meta-level to accept the message; the meta-object of the receiver then routes the message appropriately, modifying it or blocking it as necessary.

There are also compile-time MOPs that allow structural access to the source code as it is compiled and add new language functionality to the interpretation and translation mechanism; see, *e.g.*, OpenJava [72]. In a sense, these compile-time MOPs cooperatively compile source code alongside their compilers.

3.1.4 Subject-Oriented Programming

IBM's initial work with ASoC involved the concept of Subject-Oriented Programming (SOP). This work is based on the observation that many objects in a software system play different roles during their lifetimes. Because of this, objects tend to collect additional responsibilities that really implement different concerns in the overall design of the system. If unchecked, this increased integration of concerns quickly leads to increased complexity and bug-prone code copy-and-pasting; the only general solution for an active software project is to ref actor the code base regularly. This is a problem inherent in many object-oriented software systems, and is not necessarily the fault of designers; a perfectly good object-oriented design can lead to these type of problems.

In an example taken from [36], consider a *Tree* object. The *Tree* may be seen as a home to a *Bird* object; thus, the *Tree* has such properties as *shelter-value* and *branch-height*. But the *Tree* may fulfill a different value role to a *Logger* object; here, the *Tree* has such properties as *time-to-cut* and *type-of-wood*. Still another object in the system, the logging company's *Accountant* object, is concerned with market price. Here, the *Tree* has a *monetary-value*. All of these properties are inherent to the *Tree*, yet placing the code for determining each (or placing the store for holding each) inside of the *Tree* object's code quickly yields a maintainability issue; in which other objects in the world are *Bird*, *Logger*, and *Accountant* objects interested?

SOP acknowledges that this sort of issue arises in object software and seeks to provide a method of composition allowing a clean separation of an object's different roles at development time. SOP has a relation to the *Role Object* design pattern [6], in which many views can be accumulated on an object, and the very similar *Extension Interface* design pattern [64], which allows for multiple interfaces to be flexibly applied to an object.

3.1.5 Hyperspaces and Multi-Dimensional Separation of Concerns

Another approach forwarded by the same group at IBM involves identifying different dimensions of concerns in the design of a software system. In the Multi-Dimensional Separation of Concerns (MDSOC) approach, each constituent part of a program (methods, fields, and entire classes and packages) are assigned a position along each dimension. An MDSOC compiler then composes a program consisting of the proper pieces along each dimension, merging code as specified by rules provided by the developer. IBM's Hyper/J language [39] implements their technique. The compiler is freely available, and while it is currently undergoing development, it is quite stable.

3.1.6 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) allows special-purpose types called *aspects* to be *woven* into an object software system with the end result being a coherent system. Each aspect exerts a particular concern upon the software; for example, an aspect might modify a system's behavior in acknowledgment of a low power condition or implement synchronized access to a particular data structure.

Two major features define an aspect-oriented system, *quantification* and *obliviousness* [29]. *Quantification* refers to the fact that a piece of code may affect another, completely separate piece of code somewhere else in the system. *Obliviousness* refers to the fact that the affected piece of code has not been specially prepared to receive this modification. In practice, this essentially makes the *Interceptor* design pattern [64] a first-class language concern: when a particular language construct occurs, intercepting code may be spliced with it to form a new construct at that point, enacting a new behavior which may be the union of the two or a wholesale replacement of the initial code gesture.

While it is the intent of AOP to allow programmers to *separate* concerns readily, letting the language resolve the code integration dependencies, it is certainly possible to complicate object systems by porting them to AOP systems badly. As all post-object programming techniques discussed in this chapter, AOP does not seek to replace OOP, but to extend it. Thus, while there is a temptation to over-separate program concerns and refactor them into aspects, it is wise to consider the aspects and objects of a system together in order to achieve a robust design capable of being properly and easily maintained.

3.1.7 Intentional Programming

Intentional Programming (IP) [67, 40] is a proposal allowing the programmer to specify a language-neutral *intent* through an easy-to-manipulate interface. Similar in many ways to work on Model-Driven Architectures (MDAs) and code-generation tools, IP can (potentially) generate code in any target language, effecting the programmer's intents within a larger software system. IP often falls under the broad category of ASoC because it can separate the design considerations and implementation details of the target program. This separation can assist software development considerably by binding various views and rules to the system in easily configurable ways.

3.2 Applications of ASoC

ASoC technologies have been successfully applied in a large variety of areas. These are a small sampling of the resulting benefits:

- improvement, simplification, and reusability of design pattern implementations [35, 23]

- clean, modular accumulation of feature sets [38]
- detailed, online analysis of executing code in separate modules
- noninvasive support for logging and debugging purposes [4]
- simple and powerful pre- and postcondition annotation
- modular and consistent cryptography settings
- separation of business rules from functional code
- accessibility of multiple object views [49]
- increased ability to reason about object persistence
- separation of optimization, memoization, and caching code [24]
- enforcement of global behavior rules

Our interest in ASoC technologies comes out of the power and expressiveness of this family of programming paradigms for composing large-scale, adaptable software systems.[3] We have found these technologies—and the mindset of extensive concern-separation that they imply—to be very useful in both everyday programming practice and in our specialized research and development cycles. ASoC can reduce development time, simplify programs, and improve reusability.

In addition to its broad applicability to general-purpose software engineering, ASoC has important benefits for special-purpose domains as well. Power, memory, and footprint concerns can be separated cleanly into "meta-concerns," distinct from the body of "functional" code. This keeps non-functional pieces of the system generic and easier to debug, and it eliminates the chances of "missing a spot" during a (manual) global search and replace operation to change configuration of such behaviors; indeed, it can eliminate such "search and replace" code maintenance altogether. In this way, ASoC, when applied appropriately, can significantly cut the accidental complexity of a software system.

[3] The term "adaptability" often refers to *runtime* adaptability alone. However, *compile-time* adaptability can be incredibly useful in targeting software constraints; see, *e.g.*, [26]

3.3 Choice of AOP

For our work, we chose to use aspect-oriented programming. In part, our decision was based on tool support—AspectJ, described in the next section, is well-supported, stable, and quite usable. The problems we address are also particularly well-suited to design in an aspect system; in Chapter 6 we describe the aspects we implemented and the ways in which they simplified our task.

3.4 AspectJ: a general-purpose aspect extension to Java

AOP has been realized in a general-purpose aspect language extension to Java called AspectJ [3]. AspectJ identifies *join points* in the execution flow of a Java program, and provides four new language constructs: *pointcuts* specify a (possibly empty) set of join points; *lexical introduction* allows an aspect type to declare new members in other types and modify the type hierarchy; *advice* represents an AOP *quantification* that may be applied on the join points in a particular pointcut; and *aspects* provide an object type for containing these constructs. The AspectJ compiler *weaves* these aspects together with the affected classes, resulting in standard Java bytecode that may be run on any compliant JVM and that contains the behavioral properties specified both by the classes and the relevant aspects.

With AspectJ, applications can be designed and implemented more quickly, contain fewer bugs when shipped, and software development projects can increase code reuse and maintainability. AspectJ can be used throughout the development process, for small and large programs alike; it can even be used to aid in refactoring an existing code base. Several companies are using AspectJ and the research community is quite active. Recently, Hanneman and Kiczales implemented "Gang of Four" design patterns [30] using AspectJ [35], building upon prior work by Gregor Kiczales and the AspectJ team on a cleanly concern-separated implementation of the Observer design pattern. AspectJ is growing in popularity, has an active user community, and the compiler and associated tools are based on an open-source development methodology. Its tools are easy to use, freely available, and mature. It is for these reasons that we have chosen to use AspectJ in our research of concern-separation in real-time and embedded software.

Chapter 4

Motivation for Automatic Scope Assignment

We motivate the problem we study and preview our analysis using the example introduced previously in Figure 2.1(a) (page 12) which defines methods `zero`, `one`, and `two`—methods that are invoked at depth 0, 1, and 2 in the call-stack, respectively. The example also contains 7 *allocation sites* (`new` statements). The problem at hand is to map each object instance o to the "highest" call-stack frame $F \in \{0, 1, 2\}$ that satisfies the following constraints:

1. Object o is known to be dead after F is popped;

2. Any object that could refer to object o is mapped to a frame popped no sooner than F.

With regard to constraint 1, our problem is analogous to escape analysis [78, 19, 9] or to the use of "regions" [75, 74]. As explained in Chapter 2, constraint 2 is introduced to ensure safe pointer-accesses in accordance with the RTSJ.

The ultimate outcome of our analysis is a program conforming to the RTSJ; Figure 2.1(b) shows this intended result. Each instantiation site is modified to allocate its object from a storage area, in a manner consistent with the RTSJ.

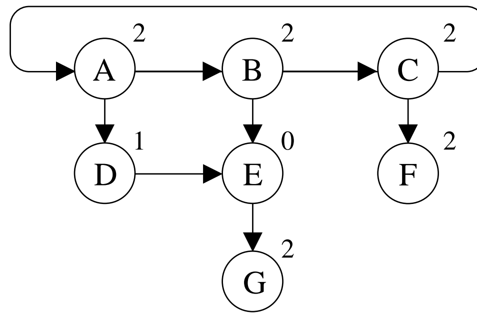

Figure 4.1: The *doesReference graph* for the program of Figure 2.1(a), vertices augmented with object liveness information.

To respect constraint 1, we first run the program in Figure 2.1(a) and observe the following object behavior:

Object	Instantiated in frame	Becomes collectible
A	2	when frame 2 pops
B	2	2
C	2	2
D	2	1
E	0	0
F	2	2
G	2	2

For the purposes of scope-creation for RTSJ, it suffices to assume that an object cannot die in a frame above its birth-frame. Analysis has been considered elsewhere to move allocation sites like these up the call-stack to decrease the lifetime of the instantiated object [66].

Next, to ensure legal references (constraint 2 above), we maintain a graph of inter-object references observed during runs of the program; in this *doesReference graph*, shown in Figure 4.1, each object is represented as a vertex, and an edge is placed between v and w if object v stores a reference to object w at some point when the program is run. We augment the graph with the "collectibility" information from the above table, as shown in the figure.

The following observations are key to solving our problem.

- Objects A, B, and C are involved in a *cycle* in the *doesReference* graph. Based on constraint 2 above, these objects must be mapped to the same frame.

- Object A references D, implying by constraint 2 that D must be mapped to a frame at or below the frame associated with A. The same can be said for objects B $\rightarrow$ E, C $\rightarrow$ F, and also D $\rightarrow$ E and E $\rightarrow$ G.

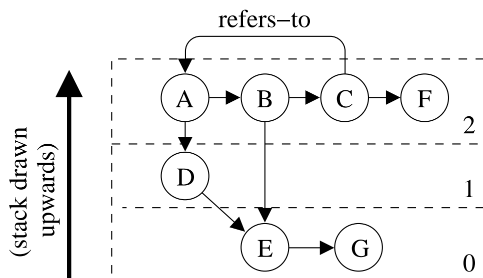

Figure 4.2: Optimal RTSJ scope assignment for the program of Figure 2.1(a) with inter-object references superimposed.

- Object `D` is instantiated in method `two` (at frame 2) but does not actually become dead until frame 1 pops, because it is returned to the calling method.
- Object `E` is allocated in frame 0, and must therefore live forever.

One possible solution to the above problem is to allocate *all* objects in frame 0's memory region—this is the last frame to be popped. This clearly satisfies the above constraints, because all objects will live for the duration of the entire program. However, we seek a solution with the objective of minimizing the lifetime of each object, so as to reduce the program's storage footprint.

The best RTSJ solution for this problem is depicted in Figure 4.2, which corresponds to the scopes used in the code of Figure 2.1(b). Inter-object references have been superimposed in the figure; when visualized in this manner, object references are required by constraint 2 to always point down (toward the base of) the stack.

Object `E` is around for the entire lifetime of the program and thus is not dead until frame 0 is popped. Object `D` cannot die until frame 1 is popped. While objects `A`, `B`, `C`, and `F` could be allocated in frame 1's memory region, that would unnecessarily increase their lifetimes. Instead, they can be allocated in the region at frame 2. Object `G` is instantiated during the execution of frame 2, but must be allocated in frame 0 because of the reference from object `E`, even though the reference was temporary.

Chapter 5

Scope Detection and Assignment

Before proceeding with a detailed account of our approach to the problem of automatically determining scopes in Java programs, we need to understand the specific scope assignment problem we address in relation to other potential models for scope assignment.

5.1 Parameterization of Scope Analysis

Some assumptions about typical object behavior are made in our scope assignment solution; these design decisions are detailed here with other potential choices we did not pursue.

For information collected during the analysis phase to be utilized when generating RTSJ-compliant code, we must be able to "recognize" an object over distinct runs of program P. This is important both to compare data from different analysis runs of P and also to calculate which particular points of P need to contain RTSJ memory scope instrumentation.

We therefore define four object lifetime behavior categories that aid in making this determination. For our purposes, "object behavior" is the lifetime behavior of an object or group of objects—specifically, birth and death events. We define the following:

- A *class* C and all object instances of runtime type C are said to exhibit *class lifetime behavior* if all of those objects behave similarly.[1]

- An instantiation site (`new` instruction) s and all object instances instantiated at site s are said to exhibit *instantiation site behavior* if all of those objects behave similarly.

- An instantiation site s and all object instances instantiated at that site are said to exhibit *call stack behavior* with respect to a function $f : S \to m$ where S is the set of all call stacks at points of instantiation and m is the set of all stack frames. This is a generalization of the previous class of object behavior.

- Object instances that do not fit into the above categories (for the purposes of our analysis, they behave only like themselves) are said to exhibit *instance behavior*.

Because we are concerned primarily with upper bounds on the lifetime of objects, and we can create an ordering, any one of these four categories is sufficient to characterize all observed object behavior in P. (For example, objects of a given type `Foo` may have dissimilar liveness characteristics, but they are all live *at most as long* as the longest-lived object among them.)

5.2 Approach

Our approach for translating a Java program P into a RTSJ-compliant, scoped memory-aware program P', is based on two kinds of analysis. In particular, for a given object instance x, we seek to answer these two questions, analogous to the two criteria introduced in Chapter 1:

1. What is the lifetime of x?

   Based on this information, we can map x's lifetime to the stack frames F_P of program P. In this way, we ensure that x's allocation in P' occurs prior to its use and that x is collected (together with other objects in the same scoped memory area) only after it is dead.

[1] By *runtime type* we intend those objects whose *most derived* type is C, rather than those objects determined by a more inclusive *instance-of* test.

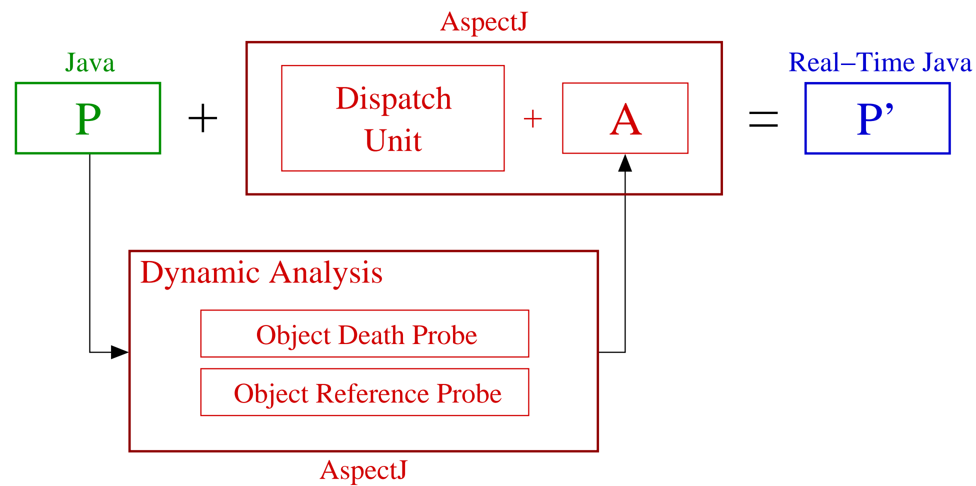

Figure 5.1: Overview of approach.

2. What other objects must have access to x during program P?

   With knowledge of those objects' lifetimes, we can ensure that x is allocated in a scope that does not violate the scope-referencing rules of the RTSJ, as explained in Chapter 1.

The results of the above analyses are represented in the *doesReference* graph, similar to the one shown in Figure 4.1. Once that graph has been determined, the algorithm presented later in this section can be applied to obtain the optimal scope assignments of Figure 4.2. Before discussing our scope-assignment algorithm, we explore possible choices of analysis and the effect on the structure and contents of a *doesReference* graph.

5.3 Analysis techniques

Analysis of object lifetime and object references can be carried out using one of the following techniques:

Static: Compile- or load-time analysis could examine the instructions of a program to determine lifetime [78] and potential storage-referencing behavior [43, 37, 28, 77].

The information obtained in this manner would hold over *any* execution of the program. Unfortunately, such information is necessarily conservative, as the problems defined above are undecidable in general.

Dynamic tracing: Traces from one or more runs of the program could be examined to profile the lifetime of objects and their storage-referencing behavior.

While this information is precise for a given set of runs, it may not represent the program's behavior in general. Thus, such information is necessarily imprecise; if programs are adapted to RTSJ based only on dynamic trace information, then the criteria listed above and those introduced in Chapter 1 could potentially be violated.

As there is no "escape" mechanism for objects once they have been allocated from a region, an adaptive, runtime approach to scoped region assignment would result in the collection of those objects to which outstanding references exist or in objects unnecessarily existing for the entire execution of the program.

The *truth* concerning a program's behavior lies somewhere between the static answer (which holds over *all* executions) and the dynamic tracing answer (which holds only over an observed set of executions). For this reason it is potentially useful to consider an approach combining the two into a single, unified approach; the research represented here pursues answers based on *dynamic, trace-based* analysis rather than static analysis. Our reasons are as follows.

- Static analysis may turn out to be overly conservative, to the point of not providing useful results for the real-time applications developer. For example, if all objects appear to be immortal, then the resulting RTSJ program will essentially preallocate all objects in one global scope.

- Although the answers don't hold for all executions, the results based on dynamic analysis can inform static analysis on the distance between its solution and some solutions that hold for particular executions. If that distance is small, then static analysis has worked well.

- If static analysis turns out to be overly conservative, then the developer will inevitably be involved in the ultimate decision about the program's scopes [7]. Our work can serve to validate or invalidate such a manual scope assignment based on actual executions of a program. Further, if the developer can provide guarantees of test coverage, stronger statements may be made about the results obtained by dynamic analysis.

The unifying principle between static and dynamic analysis for the problem we consider is the *does-Reference graph*, an example of which is shown in Figure 4.1. For dynamic analysis, this graph shows reference behavior observed in an actual run of a program. Static analysis could at best produce a "may

reference" graph, which would indicate potential referencing behavior, but would suffer imprecision because of the uncertainty of aliasing and other phenomena that must be estimated in a static analysis approach to the problem. Further, with delayed class loading and reflective calls, it can be difficult to make strict static guarantees for production Java code, since the conservative estimate reached is often far too conservative: a static analysis tool may not be able to determine the set of methods called by a reflective invocation, for example, and thus must either assume that any method in the universe may be called or relax its guarantees of accuracy.

For these reasons we use dynamic analysis in determining the interconnections between elements of the *doesReference* graph. If each vertex of the graph is a set of objects (or a characteristic that defines a set of objects), as we will discuss below, then we admit an edge $[u, v]$ to the *doesReference* graph whenever an object belonging to u stores a reference to an object belonging to v.

5.4 Vertex granularity of *doesReference*

When fully built, the *doesReference* graph will influence object instantiation so that the safe memory accesses required by the RTSJ (and shown here in Table 2.1) are respected. Before building this graph, though, we must define the granularity of the vertices of the *doesReference* graph, which could be one of the following, corresponding to the behavior categories introduced in Section 5.1:

- A vertex could represent all objects instantiated directly by a given method. This would simplify expressing the results of our analysis, since each method could enter the appropriate scope upon invocation and exit the scope prior to returning. The resulting graph will merge the behavior of objects of different types and instances. We would like to obtain a more precise result.
- A vertex could represent all objects of a given type (class) in the program. If so, then the edges of the *doesReference* graph would indicate referencing behavior over all instances of objects of that type. While such a graph might contain relatively few vertices, the resulting graph may be unnecessarily conservative. For example, if any `String` object must live forever, then so would all instances of `String` according to such a graph.

- A vertex could represent a dynamic instance of an object; this was the approach used to construct the *doesReference* graph of Figure 4.1. This way, each `String` object instantiated at runtime would be represented by its own vertex. The resulting graph would have many vertices, but the behavior of each instantiation is nicely distinguished. Unfortunately, there is then the problem of recognizing such dynamic objects across multiple analysis runs of the program and eventually on instrumentation of the program to place objects into scopes.

- As a compromise of these three ideas, each vertex could represent all instances of objects created at a particular *allocation site* (`new` command) of the program.

Because the ultimate result of our analysis is the instrumentation of allocation sites, we choose each vertex to represent a unique allocation site. If each object instance is represented separately, we would require context-sensitive information as to how the appropriate instantiation statement was reached.

To summarize, the vertices of the *doesReference* graph correspond to object-instantiation sites of the program and the edges are determined dynamically by instruction traces. The edges of the *doesReference* graph reflect inter-object references; each vertex represents all objects instantiated at a given site.[2]

In addition to the referencing information, we require per-vertex knowledge concerning when the objects associated with that vertex become collectible. Such information can be recorded as a frame number (or frame offset, as discussed below) like that shown in Figure 4.1. For the remainder of this section, we assume that the *doesReference* graph has been faithfully constructed. Chapter 7 explains the mechanics of how the relevant information can be collected.

5.4.1 Scope determination

In the observations listed in Chapter 1, and in Figure 4.2, we stored frame information based on the *absolute depth* of the call stack. We diverge from that approach here to introduce the more precise concept of *relative depth* that our analysis actually produces.

Relative depth is measured as a call-stack frame offset between the instantiation of an object and the point at which it is first determined to be dead (collectible). If an object is instantiated at frame 10 and

[2] With this definition of the *doesReference* graph, Figure 4.1 remains the same, as there is a one-to-one mapping from the 7 instantiated objects to the 7 instantiation sites.

becomes collectible when frame 4 pops, its lifetime is said to be 6 frames (it is live until the seventh frame pop). A relative liveness metric is much more suitable for our purposes than an absolute liveness metric; any method that acts as a factory (*e.g.*, the commonly-used `iterator()` method of Java's `Collection` classes) likely generates objects with very different absolute liveness characteristics, since the constructed object's liveness is heavily based upon the caller, for whom the object is being instantiated. Relative frame information alleviates this problem somewhat by reducing the penalty for a call to a factory method deep in the call stack.[3]

We now present our algorithm for determining scope assignments for a Java program P. We seek a mapping

$$Scope : V_P \rightarrow \{0, 1, 2, \ldots, maxstack(P) - 1\}$$

from a vertex $v \in V_P$ in the *doesReference* graph for P to an integer indicating the lifetime of objects aggregated by that vertex.

To find all legal, potential scopes in a Java program, we first construct the *doesReference* graph of observed inter-object references in both user and library code as previewed in Chapter 4. Vertices in this graph are groups of objects belonging to the same behavior category as described in Section 5.1. Edges in this graph represent inter-object references.

During an analysis run of program P, three types of events are distinguished and processed:

- An *object instantiation event* indicates the instantiation (allocation) of an object.
- An *object collection event* indicates the collection of an object by the garbage collector.
- An *inter-object reference event* indicates that one object is referring to another.

We presently assume objects fall under the category of *call stack behavior*, although our analysis could be extended to determine which type is most appropriate. We maintain an instance mapping M from each object instance to its presently assigned category (a vertex in the *doesReference* graph), a mapping B of objects to their birth frame index, and a mapping F of *doesReference* vertices to integers specifying the maximum *relative lifetime* on the call stack (measured as the number stack frames) observed for an object in the set represented by the vertex.

[3]Early experiments with absolute frame liveness information showed little promise, as expected, compared to the relative frame liveness results presented in Section 7.3, for the purposes of precise scope detection.

An *instantiation event* (which occurs when executing a `new` instruction) is a quadruple (o, C, s, S_b), representing the instantiation of an object o of class C at an instantiation site s and current depth S_b of the call stack (the object's birth frame stack index). To process such an event, we first determine if s is in the *doesReference* graph—if it is not, we create a new vertex v and set $F(v) := 0$. Then we add $o \to v$ to M and set $B(o) := S_b$.

A *collection event* (which occurs immediately after garbage collection[4]) is a tuple (o, S_d), indicating the automatic reclamation of object o by the JVM with current depth S_d of the call stack.[5] We respond to this type of event by setting $F(M(o)) := \max\{F(M(o)), B(o) - S_d\}$.

An *inter-object reference event* (which occurs when executing a `putfield` or `aastore` instruction) is a tuple (a, b), indicating that a is storing a reference to b. To process an inter-object reference event, we add an edge $[M(a), M(b)]$ to the *doesReference* graph if one does not already exist.

Any cyclic chain of references indicates a set of objects that must reside in the same scope. We thus decompose the *doesReference* graph into its strongly connected components upon completion of analysis runs of P. At this point, we have a directed acyclic graph indicating all legal scoping hierarchies.

Assignment of memory scopes

In an RTSJ-instrumented program, we must place objects into their correct scopes, and we must handle the opening of scopes—an RTSJ compliant JVM, as discussed in Chapter 2, will handle the closing of scopes when their top-level `run()` methods exit. Our approach is to associate a memory scope with every stack frame. Thus, if we know that a particular object x is unreachable after a particular frame f pops, we can simply place x into the scope S_f associated with f when x is instantiated. Thus, we determine the stack frame at which a given object becomes dead.

[4]It is important to note that our analysis runs rely on a garbage collector to determine when objects become collectible. However, the instrumented, scope-aware program does not make use of a garbage collector, since objects in scoped memory regions may not be individually collected (see Chapter 2).

[5]S_b and S_d are, then, the "birth" and "death" frame indexes, respectively, associated to the object o of the event in which they are present.

5.4.2 Propagation of liveness through *doesReference* graph

To see how our construction proceeds, let us first consider the case of *absolute*, rather than *relative*, frame assignment.

At the point of instantiation of object o, a nonnegative frame number o_i represents the method whose stack frame occurs at absolute depth o_i in the current call stack. We also define o_c to be the absolute frame number which, when popped, causes o to become collectible; if o is not collectible until the program exits, $o_c = 0$. For our purposes an object cannot be collected until at least the call-stack frame from which it was instantiated pops; thus $o_c \leq o_i$.

The output of scope determination, then, is the $Scope$ mapping such that $Scope(v)$ is the expected lifetime $o_i - o_c$ of objects described by vertex v, expressed as an integer.

Once the *doesReference* graph has been constructed and validated over a set of runs, we proceed to discover the $Scope$ map as follows:

1. Lifetime information associated with a vertex v indicates the frame by which all objects associated with v have become dead. This is easily computed by retaining the maximum relative lifetime observed of any object associated with v. We then have constraints of the following form:

$$Scope(v) \geq \max_{o \in Objects(v)} o_i - o_c$$

   where $Objects(v)$ is the set of all objects instantiated at the site represented in the *doesReference* graph by v. As constructed, the *doesReference* graph already provides us this information.

   For the program shown in Figure 2.1(a), liveness information imposes the following constraints:

Vertex	$Scope(v)$
A, B, C, F, and G	≥ 2
D	≥ 1
E	≥ 0

2. Referencing information constrains the relative scope allocation for the related references. According to the RTSJ, object x can reference object y only if y's associated scope fully encloses x's scope in

a nested-scope relationship for the currently executing thread. In terms of the *doesReference* graph, reference information is mapped to the vertices of that graph.

Thus, a reference from an object x to an object y is mapped to an edge from X to Y in the graph, where X and Y are the vertices corresponding to the allocation sites of x and y, respectively. Each reference contributes such an edge to the *doesReference* graph if one is not already present to represent the reference. Each edge then imposes the following constraint between all pairs (x, y) where $x \in X, y \in Y$:

$$y_c \leq x_c$$

Letting $Vertex$ be a mapping from objects to *doesReference* vertices, we can obtain

$$Scope(Vertex(y)) \geq y_i - x_c$$

and since $x_i - x_c = Scope(Vertex(x))$ in the instrumented program, we have

$$Scope(Vertex(y)) \geq Scope(Vertex(x)) + y_i - x_i$$

We can then use the observed values for y_i and x_i during analysis.

To summarize, once the *doesReference* graph is constructed, assignment of scopes can be reduced to solving a relatively simple constraint system. Each vertex is constrained by liveness information to be allocated at or below a certain stack depth; referencing behavior places edges between vertices to ensure that the relative stack-depth allocation of objects respects the rules of the RTSJ.

The constraint system that arises in this fashion has at least the following solution

$$\forall\, v, \quad Scope(v) = \max_{x \in Objects(v)} x_i$$

which effectively assigns all objects to the primordial scoped memory area. At present, ignoring the cost of scope creation and destruction, the solution we seek *maximizes* the solution for each scope, so as to minimize object lifetime. This solution minimizes storage footprint at the expense of creating (perhaps too) many scopes.

Table 5.1: Constraint system size for benchmarks (size 10) using *allocation site* granularity to construct the *doesReference* graph. Section 7.2 contains more information on these benchmarks.

benchmark	vertices	edges
`jess`	1,072	2,068
`raytrace`	879	1,721
`javac`	1,212	3,592
`mpegaudio`	818	1,605

The solution that minimizes footprint but respects all constraints can be obtained via a worklist approach, shown in Figure 5.2.

generate *doesReference* graph $G = (V, E)$
fill in $Scope(v)\ \forall v \in V$ from collected liveness information
initialize $worklist := V$
while $worklist \neq \emptyset$ **do**
  get and remove u from $worklist$
  foreach $v \in Successors(u)$ **do**
    apply reference constraint (u ref v)
    if $Changed(v)$ **then**
      $worklist := worklist \cup \{v\}$
    end if
  end foreach
end while

Figure 5.2: Worklist algorithm for scope assignment.

When calculating *relative* frame depth scopes, the application of a reference constraint (u ref v) is performed by assigning:

$$Scope(v) := \max_{w \in \{u,v\}} Scope(w)$$

The result is a scope index $Scope(v)$ for all vertices v of the *doesReference* graph (and thus for all instantiation sites of program P).

5.4.3 Target program instrumentation

With the above analysis, we can complete our goal of instrumenting the original program P to take advantage of discovered memory scopes. We instrument each method and constructor invocation to allocate and enter a new `ScopedMemory` area and each allocation site (that is, each vertex v of the *doesReference* graph) to allocate its object in the appropriate stack frame. Thus, an instantiation site with an assigned *relative depth* of 4 would allocate its object in the `ScopedMemory` area associated with the call-stack frame that is buried 4 frames under the presently active one. *Code splitting* and other techniques can be performed at this

step to reduce the overhead and number of `ScopedMemory` areas in the instrumented program; we have not implemented such optimizations.

5.4.4 Multithreaded target programs

So far we have considered only single-threaded programs. It is possible, however, to extend this approach to operate with similar liveness and reference traces collected from multiple threads executing concurrently.

In a multithreaded environment, objects that are not shared between threads can be handled in a similar way as in the single-threaded case; such objects receive a scope assignment with respect to their respective owning threads. Objects shared between two or more threads[6] must have a scope assignment in each of those threads. When all threads sharing an object have popped the stack frame associated with the object's lifetime, the object is dead.

With those observations, an extension to our approach to allow for multiple threads may be constructed. A particular scoped memory area corresponds not to a single stack frame but to a particular cut across all participating threads' call stacks.[7] Each thread is instrumented to enter this shared scope at its own appropriate call-stack frame.

[6]An object x can be shared either *directly* between threads (multiple threads access the object directly), or *indirectly* through a reference from a shared object. In either case, the combined behavior of all participating threads affects the scope assignment of x, so it constitutes a shared object for our purposes.

[7]This notion of participation is difficult to define in a satisfying manner under the RTSJ as originally published in [10]. The text here assumes the "single parent rule," a constraint introduced with the "final v1.0" version of the specification [11] that addresses a significant pointer safety problem identified in the original.

Chapter 6

Aspects

Once the analysis described in previous sections has been performed, each object instantiation site has been assigned a particular stack frame from which to allocate. We need a way of transforming the program P so that it takes advantage of this information by creating memory scopes at proper times and placing objects into their assigned scopes. We find it beneficial to use the notion of Aspect-Oriented Programming to encapsulate this procedure in an *aspect*. The aspect contains *advice* to ensure the following:[1]

1. When a method or constructor is called, advice is patched in to associate it to a scoped memory area. This new `ScopedMemory` area is pushed onto a stack of memory regions. Then, the actual method is dispatched within this memory area. After it returns, the advice pops the memory scope off the stack.

2. Allocation sites (`new` instructions) are advised to place the objects they instantiate into the correct memory scope.

Advice similar to that of Figure 6.1 is generated, with implementations of the `makeScope` and `getScope` methods referred to in that example. Recognition of objects (`isObjectD` and `isObjectG` methods in that example) is easily performed by looking up the currently executing instantiation site (using the `thisJoinPoint` construct of AspectJ) in the table generated by the analysis.[2] The output of the

[1] As mentioned in Section 3.4, each piece of AspectJ *advice* represents an AOP *quantification*. Concretely, a piece of advice is a piece of code that executes in the execution context of a join point somewhere else in the program.

[2] Briefly, `thisJoinPoint` offers reflective contextual information about the current execution point (*join point*). For example, if the join point being advised is a "method call" join point, `thisJoinPoint` offers access to its name, defining class, attributes,

```
aspect ScopedMemoryAspect {
  ScopeStack scopes = new ScopeStack();
  Object around(): call(void Example.one())
              || call(Object Example.two(Object)) {
    Object retval = null;
    ScopedMemory sm = makeScope(thisJoinPoint);
    scopes.push(sm);
    sm.enter(
      new Runnable() {
        public void run() { retval = proceed(); }
      } );
    scopes.pop();
    return retval;
  }
  Object around(): call(RefObject.new(..)) {
    // Object D is allocated one frame back
    if(isObjectD(thisJoinPoint.getSourceLocation()))
      return scopes.getScope(1) // region for frame 1
        .newInstance(RefObject.class);
    if(isObjectG(thisJoinPoint.getSourceLocation()))
      return scopes.getScope(0) // region for frame 0
        .newInstance(Object.class);
    // otherwise allocate from current memory area
    return proceed();
  }
  /* ... */
}
```

Figure 6.1: Advice implementing scopes in the example program of Figure 2.1.

analysis tells us the stack frame F at which an object at that site becomes dead, and we retrieve associated memory region from the stack of `ScopeMemory` objects to perform the actual allocation.

6.1 Analysis aspects

By determining the execution-flow join points of interest and then advising those points, we can collect and act on analysis data. We have written a *reference-probing aspect* to keep track of potential inter-references between different objects and a *death-probing aspect* to determine when objects become unreachable (and therefore eligible for collection by the garbage collector).

Both analysis aspects inherit from a simple, abstract aspect (see the `Probe` class in Appendix A) that defines a few useful definitions for sets of join points. These are called *pointcuts* in AspectJ and can (informally) be considered a definition of a “net” with which to “catch” certain join points when they occur. For example, the `methodExecutions()` and `constructorCalls()` join points specify such “nets” to “catch” method executions and constructor calls.

signature, and parameters. For a “reference” join point, the field name and type and the objects involved in the reference are available through `thisJoinPoint`.

The `withinUs()` pointcut is used in a different manner. It is defined similarly to the others, and it "catches" join points that occur within the analysis aspects themselves. However, its negation is applied—that is, it is used to inhibit our aspect advice from acting on the aspects themselves. This behavior is generally not desired, and leaving this out is a common AspectJ programming error. Such a bug often results in an infinite loop, with a piece of advice recursively advising itself.

There are three general kinds of advice in AspectJ—"before," "after," and "around." As its name suggests, "before" advice is triggered *before* a join point is executed—that is, if it is advising a *method-call* join point, it executes *before* the method call does. "After" advice is the reverse: *first* the join point is executed, *then* the advice is triggered.[3]

The third kind, "around" advice, runs *in place of* its target join point. Thus, if "around" advice is advising a method call join point, the method is *not* called before *or* after the advice is triggered. In such a case, the advice code has complete control over whether the method is called at all. If the advice decides to execute the join point, it can do so through AspectJ's `proceed()` construct.

6.1.1 Reference-probing aspect

As discussed in Chapter 2, it is an error for an RTSJ object to reference an object that is not in an ancestor's scope. We have developed a reference-probing aspect to harvest object-referencing activity from a set of program executions. Appendix A shows the reference-probing aspect's implementation.

As described in Chapter 5, we are presently concerned with grouping objects together based on the location of their instantiation—that is, the source code location of their associated `new` operation. To track the objects that are created in our system, we use "before" advice on constructor executions. This advice sets up an unresolved `SourceLocationImpl` object. When the constructor call is completed, our "after" advice on constructor calls resolves the source location of the instantiation.

We also have advice to detect when one object references another. "Around" advice captures *field-assignment* join points and calls the `reference()` method, registering the reference. One or both of the objects involved in the reference may have an unresolved instantiation source location—its constructor may

[3]There are further classes of "after" advice in AspectJ as well, corresponding to whether a normal or exceptional path was taken to exit the join point. Further discussion may be found in the AspectJ Programming Guide [4].

still be executing and, thus, our resolution advice has not yet run. In this case, the reference is kept in a stack of references that are not yet resolved. A subsequent call to `reference()` will detect that the instantiation locations have since been resolved and output the reference data.

The information thus collected can then be analyzed to determine legal scoping hierarchies for RTSJ. Section 6.1.3 describes the scope-forming process in more detail.

6.1.2 Object death-probing aspect

In Java, one can use the `java.lang.ref.WeakReference` class to keep references to objects without inhibiting their collection. Further, one can use a `WeakReference` together with an instance of Java's `java.lang.ref.ReferenceQueue` class to carry out processing when the object becomes collectible in a more flexible way than is available with object finalization.

Our *death-probing aspect* uses Java's `ReferenceQueue` and `WeakReference` types to receive notification of each object's "collectibility." Each method- and constructor-exit is advised with a collection cycle, and the set of such collectible objects can thus be determined. A sketch of the death-probing aspect is shown in Appendix A.

A *frame* field is introduced into `Threads` to track birth and death "times" of objects created by the thread, as well as the execution join points at which they become collectible. Using this information, we can determine how many stack frames particular objects remain live after being instantiated.

The `DeathProbe` aspect also maintains a list of references to outstanding live objects, `refs`,[4] and the `ReferenceQueue` instance. "Around" advice captures all method executions and constructor calls, maintains the frame number, and detects dead objects. "After" advice is used on constructor calls to capture the birth frame of instantiated objects and store them in a `DeathProbeReference`. The type `DeathProbeReference`, defined as a member class, extends `WeakReference` and provides for the storage of objects' birth information.

[4]These references are necessary. If a `WeakReference` object becomes itself collectible, it will never be placed on the `ReferenceQueue` upon its referent's collection.

6.1.3 Offline analysis

In keeping with the approach outlined in Chapter 5, we must determine two critical pieces of information from the data collected by the analysis aspects:

- The scoping hierarchy to use in the RTSJ translation
- The execution join points of the program that should enter memory scopes.

To determine the scoping hierarchy, we construct our *doesReference* graph from the reference information obtained by the reference-probing aspect, where each node of the *doesReference* graph corresponds to a well-defined set of objects in the original program.[5] We collapse the strongly-connected components of the *doesReference* graph, as any strongly-connected object-reference chain implies that all of the participating objects must be in the same memory scope. Collapsing strongly-connected components results in a directed acyclic graph (DAG). Because legal references may only point to objects in the same scope or longer-lived scopes, the transpose of this collapsed *doesReference* graph is a DAG representing parent-to-child scope nesting relationships.

We then select a tree over this DAG to be the memory scope hierarchy of the translated program. All references observed in dynamic-analysis runs of the program point upward in this generated scoping tree and are therefore consistent with RTSJ's reference rules. Having decided on the particular memory scoping hierarchy to use, we can take the information gleaned from the death-probing aspect to determine the execution scopes—methods and constructors—on which we wish to root the discovered memory scopes.

6.2 Runtime enforcement

Once a particular scoping hierarchy has been selected, and the exact join points that should signal the entrance into scopes have been determined, we must enforce the scoping strategy on the running program. We can achieve this using AOP as well. We advise join points that should enter memory scopes to do so. "Around" advice is used, as shown in Figure 6.2.[6] The scope is entered by a `RealtimeThread`, which dispatches

[5] As mentioned in Section 6.1.1, we are currently grouping objects together into *doesReference* nodes based on the source code location of their construction—their associated `new` operation. In the future, we will be investigating more flexible and adaptive ways of grouping objects into *doesReference* nodes.

[6] Of course, the actual structure of the generated memory aspects will depend on the analyzed program.

control back to the advised join point. No special treatment is required to close the scope, as that will automatically follow the execution scope's exit as guaranteed by RTSJ.

The enforcement aspect makes use of our `ScopeMap` class. `ScopeMap` implements Java's `Map` interface and maps AspectJ `JoinPoints` to `ScopeMemory` instances.

```
package autoscope.runtime;
aspect EnforcerAspect {
  ScopeMap scopes = new ScopeMap();
  void enterScope(JoinPoint jp, Runnable logic) {
    scopes.getScope(jp).enter(logic);
  }

  around() returns Object:
        executions(public void Foo.foo()) {
    Runnable r = new Runnable() {
        public Object retval;
        public void run() {
            retval = proceed();
        }
      };
    // The run() method of "r" executes in our thread,
    // not its own.
    enterScope(thisStaticJoinPoint, thr);
    // Scope no longer exists after call to enterScope()
    return thr.retval;
  }

  // etc.
}
```

Figure 6.2: A sample enforcement aspect.

6.3 Other aspect analysis techniques

We have herein demonstrated the use of aspects for collecting and analyzing dynamic information about a running program. There is a wide variety of data that could be collected with aspect-characterized dynamic analysis:

Instantiation and collection behavior: We operate in this realm of analysis, but other possibilities exist. A developer may be interested in knowing the maximum amount of live storage in the system at one time, or the maximum number of live objects. Thread-based analysis could be performed to determine the amount of work done by each thread in a thread pool.

Inter-class behavior: Inter-class call behavior could be detected in addition to inter-class references. Particularly high method-execution frequencies could be detected to indicate the most important places to optimize or inline code.

Dead code removal: Method and constructor implementations never used by a running system can be detected and replaced by a no-op in bytecode files to reduce the size of the software.

Class preloading: Classes used by an embedded system application could be determined and then either preloaded or placed in flash memory in the running system to buffer against unexpected pauses during runtime. In some circumstances it can be impossible to determine all loadable classes statically (*e.g.*, in a reflective language like Java).

Of course, these are just a few possibilities among a large number of potential applications for AOP, and AspectJ in particular, in implementing optimizations discovered through static or dynamic analyses.

6.4 Remarks

We have demonstrated the suitability of aspects for dynamic analysis and for the application of insights gleaned from analysis. While our examples have been limited chiefly to the collection of information processed in offline analysis, aspects are sufficiently expressive to allow for program analysis and optimization at runtime. As such, more powerful, adaptive dynamic analysis [60] could be performed during runtime.

Chapter 7

Experimentation

In this section we present results that quantify the scopes formed by our analysis and compare the resulting scoped object lifetimes to observed object lifetimes. As stated earlier, we assume the cost of scope creation and deletion is negligible; thus, we are interested in creating the most refined scopes possible—as near the top of stack as possible—to minimize the time an object spends from its logical death until the point of collection (by having its associated scope deleted). This method is designed to decrease the memory footprint of the resulting system by collecting objects as quickly as possible; however, because of potential accounting overhead of scoped memory regions, this is not guaranteed.

7.1 Experimental method

We instrumented Sun's JVM version 1.1.8 to collect the information required by our algorithm: birth and death frames of objects and object inter-reference information.[1] We also tagged each instantiated object with its instantiation site—that is, the `new` instruction that created that object. This information is required to relate referencing behavior to the vertices of the *doesReference* graph.

[1] Our aspects have been tested and perform this purpose, but we wished to use a specialized reference-counting garbage collector and had a logging-instrumented JVM to use.

7.2 Benchmarks

We then built the *doesReference* graph and computed a mapping from vertices to object liveness for the objects using the approach presented in Chapter 5. We report results using the following benchmarks (size 10) from the Java SPEC suite [68]: `raytrace` renders an image; `javac` is the Java compiler from Sun's JDK 1.0.2; `mpegaudio` is a computational benchmark that performs compression on sound files; `jess` is an expert-system shell application that solves a puzzle in logic. The size of these benchmarks is shown in Table 7.1.

Table 7.1: Benchmark sizes: the number of objects and distinct allocation sites encountered during execution.

benchmark	# objects	# allocation sites
`jess`	106,727	1,098
`raytrace`	559,287	925
`javac`	211,080	1,239
`mpegaudio`	9,166	827

Object references were traced by instrumenting the JVM to emit messages in response to `putfield` and similar instructions that cause one object to reference another. Object lifetimes were collected by using a simple reference-counting garbage collector implementation that finds dead objects at each frame pop. This particular reference-counting implementation counts only inter-object references, and handles local references (from local variables and Java's operand stack) by associating the object with the lowest stack frame that exhibits such behavior. When a frame pops, its object list is traversed and any object with a reference count of zero is then collected. While the reference-counting collector is approximate, it is close enough for these benchmarks [15]. Use of an exact collector would cause our experiments to take too much time [66].

Some inexact methods of garbage collection are not suitable for this analysis as they closely couple inter-object references and object lifetime. For example, contaminated garbage collection [15, 14] groups objects that reference each other into *equilive sets*, expecting that they will become collectible at the same time.[2] If based on a contaminated collector, our scope analysis would not output a properly nested scoping structure; the scopes would be unnecessarily conservative due to the collector's bidirectional association between an object making a reference and the target of the reference. In effect, our two forms of dynamically-collected data—the object reference and object collection traces—would become utterly tangled and useless.

[2] In brief: these equilive sets of objects are each associated to a frame; when a reference between two objects x and y occurs, the sets to which x and y belong are merged (unioned) and re-associated to the lower frame. In this way, garbage marking is done incrementally with each reference. When a frame pops, objects in equilive sets associated to it are known to be garbage and are scheduled for collection or recycling.

We are working on the instrumentation of Java code as described in Chapter 5 to use the scopes harvested by our analysis. However, we cannot at this time offer measurements of improved real-time determinism due to the use of these scoped memory regions.

7.3 Results

The results presented here show some relevant information about the benchmarks and demonstrate scope formation using our technique. We compute an assignment of each object to a non-garbage collected scoped memory area corresponding to its instantiation site. The scope assignment satisfies both the observed object lifetimes and the reference constraints of the RTSJ introduced in Chapter 2.

Figure 7.1 shows the number of objects we assign into each scope for the `mpegaudio` benchmark. Column 0 accounts for objects that are collected in their allocating frame, using the scoped memory areas we've discovered. Column 1 accounts for objects that are collected when the stack frame of their allocating method's caller pops.

Figure 7.2 shows the corresponding comparison for `raytrace`; Figure 7.3 shows the `javac` benchmark, and Figure 7.4 shows the results for `jess`. Of the four benchmarks, three allocate a plurality of their objects such that they become dead when their allocating frame pops; `mpegaudio`, a primarily computational benchmark that allocates fewer objects than any of the other three (see Table 7.1), allocates more objects that ultimately live longer when placed in scoped memory.

In the `jess` benchmark, a substantial number of objects are paired off in circularly-referencing relationships; due to this referencing behavior, our reference-counting garbage collector does not collect such objects until program termination; since this information serves as input to our analysis, our analysis assumes that such objects are longer-lived than they actually would be under an exact garbage collector.

For practical reasons, inter-procedural escape analysis is typically performed with limits on the maximum call-stack depth;[3] thus escape analysis [78] could perform rather well for a benchmark like `raytrace`

[3] Personal communication, Martin Rinard.

in which most of the objects become collectible after their allocating stack frame is popped. Our results indicate that for programs that allocate longer-lived objects (like `mpegaudio` in Figure 7.1), the added expense of deeper static analysis is justified.

Of course, due to our RTSJ-imposed reference constraints and the fact that we aggregate object behavior by *allocation site*, we cannot always collect (by closing a scoped memory) all objects as soon as they become collectible. To measure this, we consider the lifetime (in frames) of each object according to the reference-counting collector and subtract from this the lifetime of the object when placed in its assigned scope according to its allocation site's assigned scope index. The result is its "extra longevity" under the scope assignment.

Figure 7.5 shows the "extra longevity" afforded the objects of the `raytrace` benchmark. The plot indicates the number of objects that ultimately live longer than necessary when placed in our scoped memory areas and the additional number of frames for which they live. For example, when placed in scoped memory, there are 27,588 objects in `raytrace` that live one frame longer than is necessary.

Figure 7.6 shows a similar plot for the `mpegaudio` benchmark. Only 700 objects of this benchmark live one frame longer than necessary. In fact, we can accurately scope approximately 2,300 of the 9,000 objects in `mpegaudio` such that they live no longer than they would under the reference-counting collector.

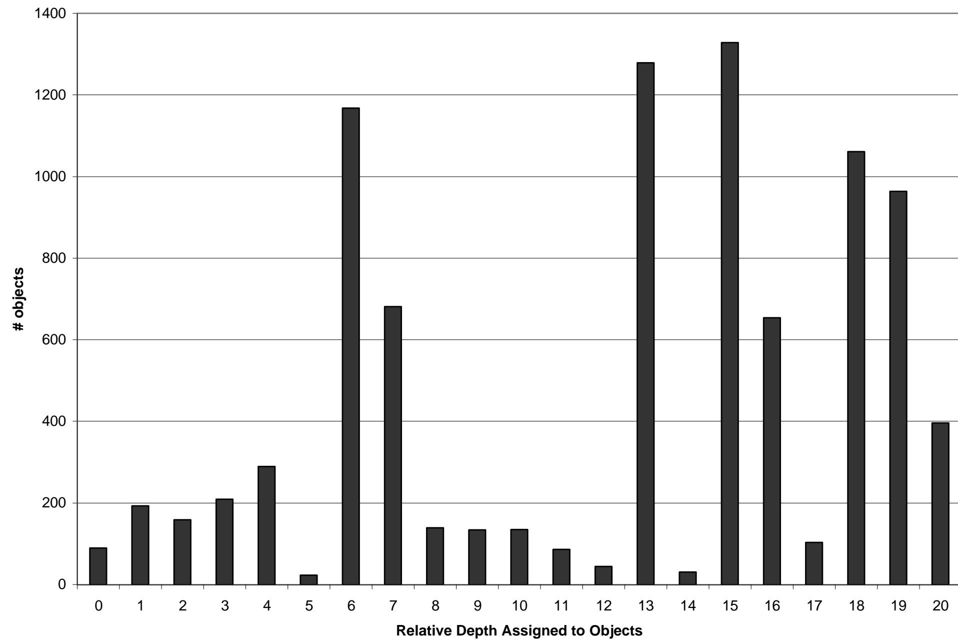

Figure 7.1: Scoping assignments for objects in `mpegaudio`.

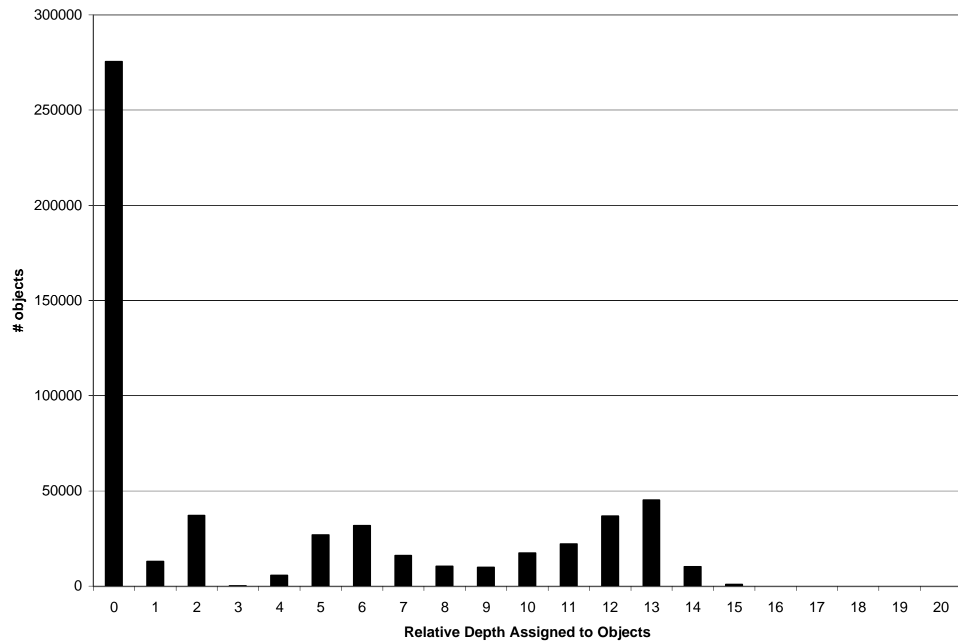

Figure 7.2: Scoping assignments for objects in `raytrace`.

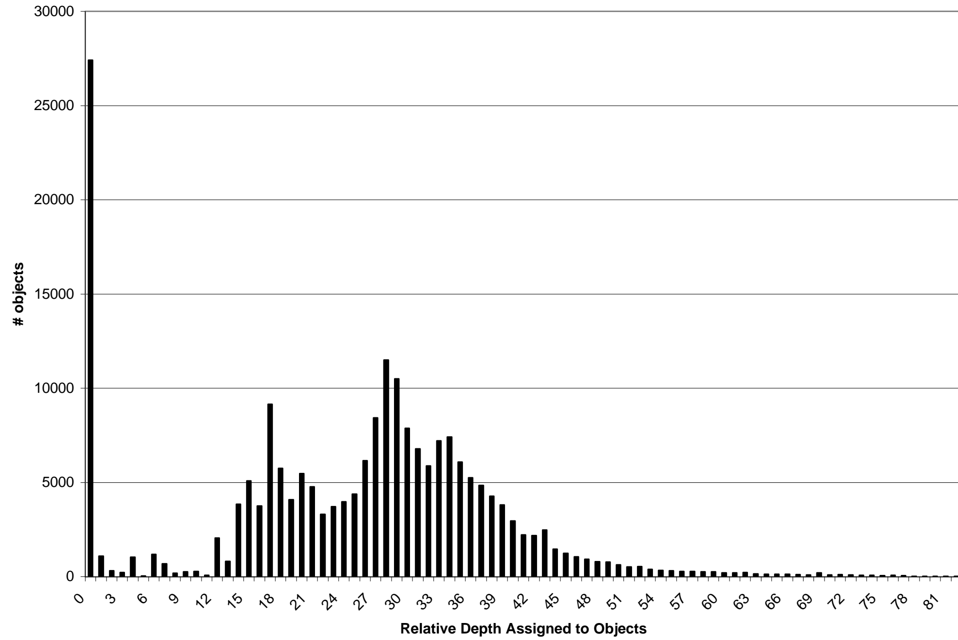

Figure 7.3: Scoping assignments for objects in `javac`.

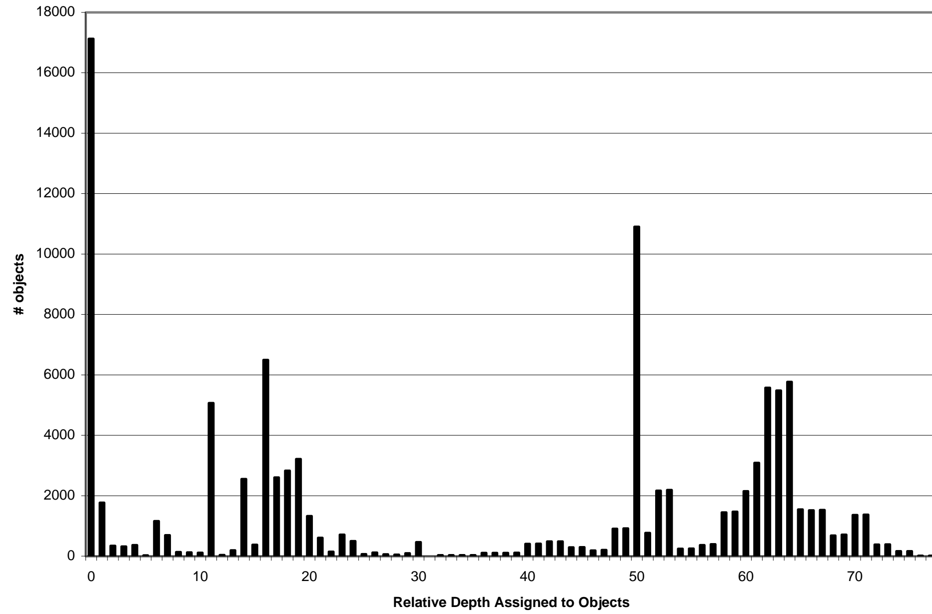

Figure 7.4: Scoping assignments for objects in `jess`.

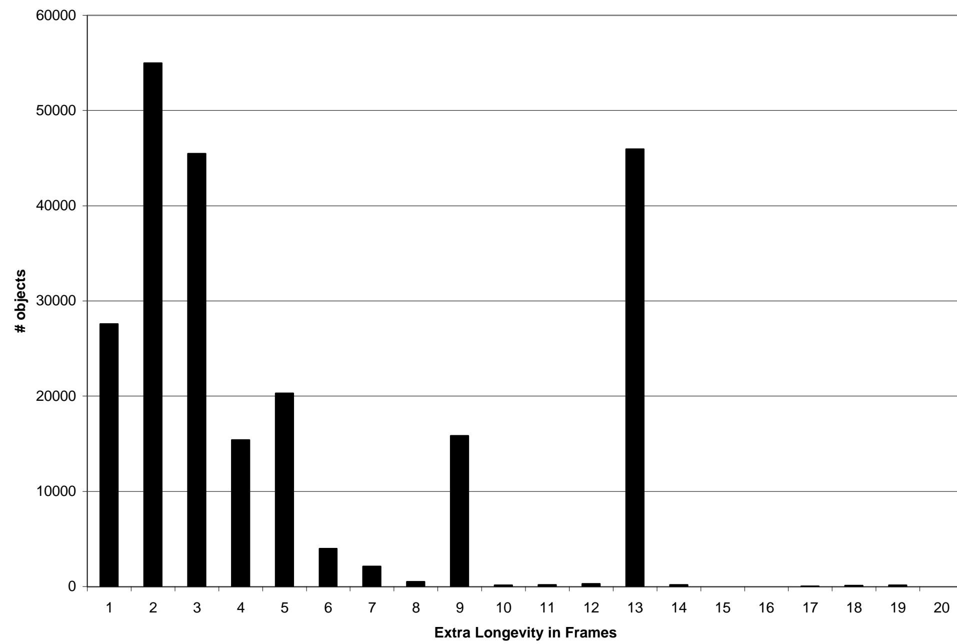

Figure 7.5: Extra longevity of objects in `raytrace` using scoped memory.

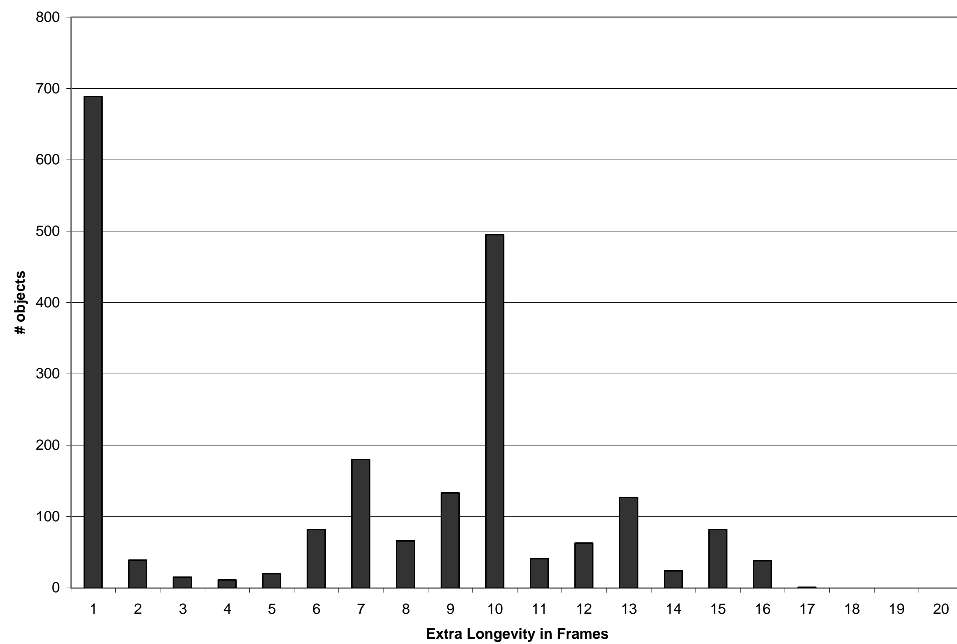

Figure 7.6: Extra longevity of objects in `mpegaudio` using scoped memory.

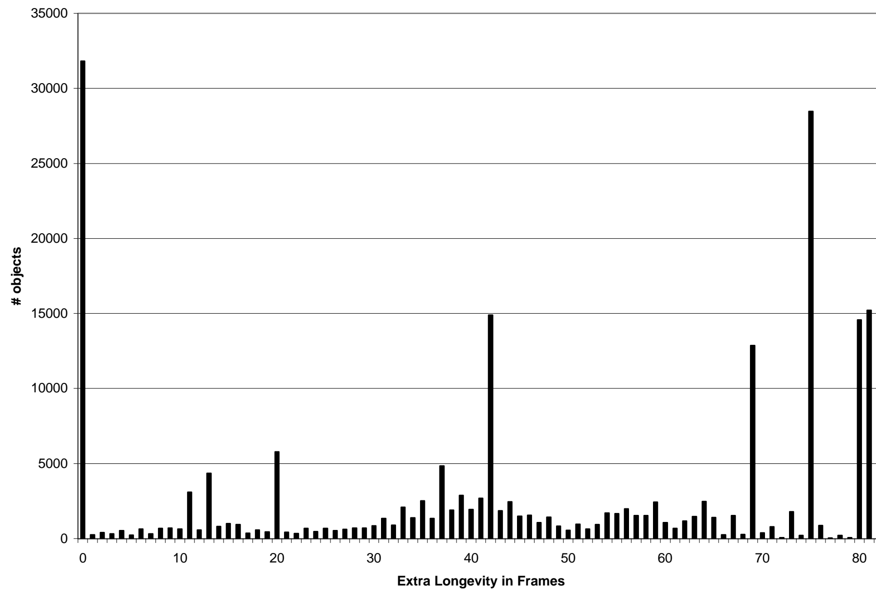

Figure 7.7: Extra longevity of objects in `javac` using scoped memory.

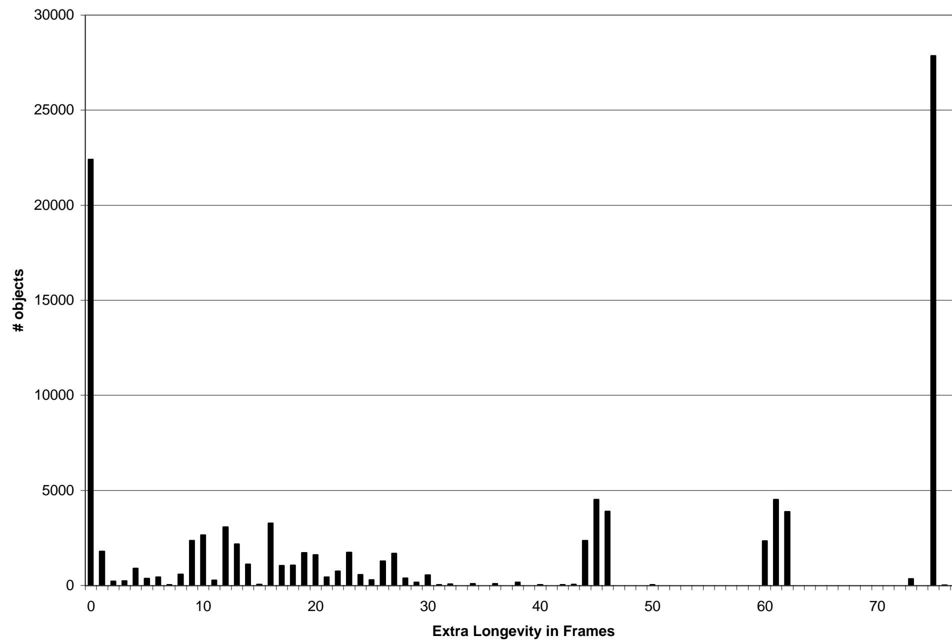

Figure 7.8: Extra longevity of objects in `jess`.

Chapter 8

Related Work

Advanced Separation of Concerns (ASoC) research is surveyed separately in Chapter 3. Other related work includes that of region-based memory systems and analysis techniques for object-oriented programs.

8.1 Region-based memory systems

Gay and Aiken [31] provide a regions library with such operations as `newregion` to create a new region, a corresponding `deleteregion`, and `ralloc` to allocate within a region; programmers target a region-annotated dialect of C, called RC, that permits them to indicate restricted pointers that may only point to objects in their own region or a limited (possibly empty) set of other regions. Their RC compiler performs static checks and injects code for runtime checks that ensure that the behavior of these restricted pointers fits their annotation. A reference count of external pointers into a region is kept, and a region may only be deleted when no outstanding external references exist.

A stack of regions is suggested by Tofte and Talpin [75, 74]. A programmer can bind the evaluation of an expression to a particular memory region, similar to the use of scoped memory regions in Real-Time Specification for Java [10] (RTSJ). This view of region-based memory management has been realized in the ML Kit [53], a Standard ML [52] compiler. Recently the ML Kit has been extended to allow for the garbage collection of these regions; an account of this extension is found in [34].

8.2 Analysis Techniques

Static pointer and escape analysis [77, 78] seeks to collect information about the referencing and lifetime behavior of objects; such analysis is often targeted to a variety of optimizations, including the allocation of objects on the stack and synchronization elimination. The analysis by Sălcianu and Rinard [63] introduces a *parallel interaction graph*, provides safety checking of supplied region gestures, and can be used to remove unnecessary runtime instructions for region verification, all for a multithreaded target program. These analyses are performed statically and may need to make overly conservative assumptions about the behavior of the program, as mentioned in Chapter 5. Manual modifications to the discovered regions may be required to achieve desired results [7].

Chapter 9

Conclusion

Automatic memory management is a useful technique that substantially releases the programmer from concerns about the lifetime of objects in a software system. There is a cost to using automatic memory management, however, and because the state of a system's memory and garbage collector are often highly unpredictable, a real-time system can miss important deadlines if it stops or slows program progress for an unbounded (or unreasonably bounded) period of time to manage memory internally.

The Real-Time Specification for Java addresses this problem by offering regions of memory that are not garbage collected but instead are reclaimed after a particular scope of execution exits. Chapter 5 presented an algorithm to discover how and where to instrument Java programs to take advantage of these non-garbage collected, scoped memory regions defined by the Real-Time Specification for Java [10] (RTSJ). We have presented experimental results to verify that much of the memory used by standard benchmarks is locally allocated (or is live for a short time on the stack) but that a significant amount can live longer; these long-lived allocations can be difficult for static analysis to handle without resorting to overly conservative assumptions.

9.1 Future work

We are presently investigating numerous improvements to our scope-detection tool:

Aspects: Aspect-Oriented Programming (AOP) [42] can be used to implement the reference and lifetime dynamic analyses and also to instrument the program to take advantage of scoped memory regions. In this approach, the analysis is designed, implemented, and then *weaved* into the program we wish to trace. Similarly, after the analysis is performed a behavioral description of the output of the analysis is weaved into the program. We have already implemented such profiling aspects [27, 25], but are unable to use these implementations to analyze Java library code or objects created internally by a Java Virtual Machine (JVM) due to limitations in the current version (1.0.6) of the AspectJ compiler. The primary advantages to using AOP for this purpose are (1) to avoid instrumenting and re-targeting Java interpreters individually to log appropriate data for our analysis, and (2) to separate the Java and RTSJ code cleanly in the target program.

Multiple threads: Further evaluation of our threading-aware analysis techniques as introduced in Section 5.4.4 is necessary. We have developed a prototype threading-aware AspectJ aspect to perform this multithreaded analysis and plan to continue its development as a part of this work.

Alternative liveness measures: Types of object death measurement other than the *relative death* metric described here are possible. One example is the time of death of an object relative to the entry of a method in a particular library or class. This type of measure could be quite useful in analyzing several modular components separately, then composing those analyses' data into a full, coherent runtime system that makes use of all of those components.

Code splitting: The granularity of a vertex in the *doesReference* graph can be made more precise by "code splitting" at an instantiation site, with the resulting `new` commands tailored by some calling context. This could help objects created by factory methods (and other methods that generate objects on behalf of the caller), as we could then consider not only the allocation site of the object but also the caller of the allocating method.

Combination of analyses: Our dynamic analysis could be combined with static escape analysis [78] to provide static analysis with areas where precision could be improved and to inform our analysis of areas where we should be more conservative.

Appendix A

Source for Analysis Tools

Probe.java

```
package autoscope.probe;

import java.io.*;
import java.util.*;

/**
 * Contains some important pointcuts for aspect probes in this
 * package.
 */

abstract aspect Probe {
  /** The ID string of the RCS file.<p>Currently:
   * $Id: Probe.java,v 1.1 2002/11/25 08:05:13 mdeters Exp $ */
  protected static final String rcsid =
    "$Id: Probe.java,v 1.1 2002/11/25 08:05:13 mdeters Exp $";

  /** A pointcut to exclude {@link 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) returns void :
    args(cmdline) && execution(public static void main(String[]));

  /** The PrintWriter to which to send output.
   * @see #outExtension()
   * @see #around(String[]) */
  protected PrintWriter out;

  /** The PrintWriter to send debugging information to -- by
   * default, this is a PrintWriter around System.err. */
  protected DebugPrintWriter debug = new DebugPrintWriter(System.err);

  /** Debugging on or off?  If <i>false</i>, all invocations of
   * methods (except <code>new</code>) on {@link
   * Probe.DebugPrintWriter} instances are ignored.  If <i>true</i>,
   * all such calls are permitted to proceed. */
  protected boolean debugOn;
```

```
/** Return the Object ID (class name + "@" + hex identity hash
 * code).  Meant for debugging purposes. */
protected String oid(Object o) {
  return o.getClass().getName() + "@" +
    Integer.toHexString(System.identityHashCode(o));
}

/** Advice to halt method invocations on {@link
 * Probe.DebugPrintWriter} when debugging is off (via {@link
 * #debugOn}). */
/* -- doesn't compile (compiler crash)
   Object around(DebugPrintWriter pw): call(* PrintWriter.*(..))
   && target(pw) && this(Probe) {
   System.out.println("debug " + thisJoinPoint);
   return null;
   }
*/

public void DebugPrintWriter.on() {}
public void DebugPrintWriter.off() {}

void around(Probe p): this(p) && call(public void DebugPrintWriter.on()) {
  p.debugOn = true;
}
void around(Probe p): this(p) && call(public void DebugPrintWriter.off()) {
  p.debugOn = false;
}

/** Must be overridden by subaspects to specify the output file
 * extension.
 * @return the filename extension to use for the {@link #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 {@link Probe} subaspect does.  Only thing is, the
 * subaspects of {@link Probe} have to have member aspects that
 * dominate {@link Probe.MainCutter} and force them (the
 * subaspects of {@link Probe}) to be loaded, adding to {@link
 * #theProbes} a reference to themself.  <i>Really ugly.</i> */
static aspect MainCutter dominates Probe+ {
  /** A {@link java.util.Vector Vector} of {@link Probe}
   * aspects.  This is added to with {@link #after(Probe)
   * <i>after</i>} advice on {@link Probe} initialization. */
  protected Vector theProbes = new Vector();
  /** Advice to grab onto <code>MakeMyPresenceKnown</code>
   * aspects in {@link Probe} subaspects.  It adds {@link Probe}
   * instances to the {@link java.util.Vector Vector} {@link
   * #theProbes} as they are initialized. */
  after(Probe p): this(p) && initialization(Probe.new(..)) {
    theProbes.add(p);
    p.debug.println("Probe.MainCutter adding " + p);
  }
  /** Advice to set up the log file output {@link
   * java.io.PrintWriter PrintWriter} ({@link Probe#out}).  The
   * filename extension is retrieved from {@link
   * Probe#outExtension()}.
   * @see 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();
    }
  }
}

/** Member class to serve as a debugging writer.  This class only
 * exists so that the <i>around</i> advice can distinguish
 * debugging calls from other {@link java.io.PrintWriter
 * PrintWriter} calls. */
public static class DebugPrintWriter extends PrintWriter {
  /** Create a new {@link DebugPrintWriter}, <i>with</i>
   * automatic line flushing, from an existing {@link
   * java.io.OutputStream OutputStream}.  Using this constructor
   * is identical to calling {@link
   * #DebugPrintWriter(java.io.OutputStream,boolean)} with the
   * second argument equal to <i>true</i>.
   * @param out An output stream */
  public DebugPrintWriter(OutputStream out) {
    this(out, true);
  }
  /** Create a new {@link DebugPrintWriter} from an existing
   * {@link java.io.OutputStream OutputStream}. This constructor
   * is identical to {@link java.io.PrintWriter's} {@link
   * java.io.PrintWriter#PrintWriter(java.io.OutputStream,boolean)
   * constructor} of the same signature.
   * @param out An output stream
   * @param autoFlush A boolean; if true, the println() methods
   * will flush the output buffer */
  public DebugPrintWriter(OutputStream out, boolean autoFlush) {
    super(out, autoFlush);
  }
  /** Create a new {@link DebugPrintWriter}, <i>with</i>
   * automatic line flushing. */
  public DebugPrintWriter(Writer out) {
    super(out, true);
  }
  /** */
```

```
    public DebugPrintWriter(Writer out, boolean autoFlush) {
      super(out, autoFlush);
    }
  }

  /** A member class to serve as an {@link Error} class for {@link
   * Probe}s */
  protected static class ProbeError extends Error {
    /** Construct a {@link ProbeError} with no detail message */
    public ProbeError() { }
    /** Construct a {@link ProbeError} with the specified detail
     * message.
     * @param s the detail message of the {@link ProbeError} */
    public ProbeError(String s) { super(s); }
  }
}
```

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 {
  /** The ID string of the RCS file.<p>Currently:
   * $Id: ReferenceProbe.java,v 1.1 2002/11/25 08:05:13 mdeters Exp $ */
  protected static final String rcsid =
    "$Id: ReferenceProbe.java,v 1.1 2002/11/25 08:05:13 mdeters Exp $";

  /** Whether or not to print stuff verbosely. */
  protected boolean verbose = false;

  /** A hash of encountered new sites. */
  protected Hashtable newLocHash = new Hashtable();

  /** A {@link java.util.Hashtable Hashtable} of {@link Thread}s to
   * the {@link org.aspectj.lang.reflect.SourceLocation
   * SourceLocation} of <code>new</code> sites */
  protected Hashtable newSiteThreadHash = new Hashtable();

  /** Returns our preferred filename extension "ref".
   * @return The {@link String} "ref", signalling that we want an
   * output file with a ".ref" extension.
   * @see Probe#outExtension() */
  protected String outExtension() { return "ref"; }

  /** Sets a "currently executing" <code>new</code> site {@link
   * org.aspectj.lang.reflect.SourceLocation SourceLocation} for the
   * current {@link Thread}. */
  before(): constructorCall() {
    newSiteThreadHash.put(Thread.currentThread(),
                          thisJoinPointStaticPart.getSourceLocation());
  }

  /** Links the {@link org.aspectj.lang.reflect.SourceLocation
   * SourceLocation} of a <code>new</code> site to the object
   * currently under construction.<p>
   * The {@link org.aspectj.lang.reflect.SourceLocation
   * SourceLocation} is retrieved from {@link #newSiteThreadHash},
   * which contains a mapping of {@link Thread}s to the
   * <code>new</code> 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)
      newLocHash.put(o, newSiteThreadHash.get(Thread.currentThread()));
  }

  /** Links the {@link org.aspectj.lang.reflect.SourceLocation
   * SourceLocation} of a <code>new</code> site to a constructed
   * object.<p>
   * The {@link org.aspectj.lang.reflect.SourceLocation
   * SourceLocation} is retrieved from this (constructor call) join
   * point.<p>
   * This advice is necessary for objects instantiated by our system
   * but whose constructor code is outside our system (for example,
   * instantiating a {@link 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 {@link
   * java.io.PrintWriter PrintWriter} specified in {@link #out})
   * that an object refers to another object.<p>
   * The {@link String} printed for each item is the {@link String}
   * returned by {@link #getSrcLoc(Object)}, unless parameter
   * <code>a</code> is an instance of {@link 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
   * @see ReferenceProbe.StaticClass
   * @see #getSrcLoc(Object) */
  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 {@link String} representing the new site for the given object.
   * @param o the object to get the new site for
   * @return a {@link String} of the form "Foo(Foo.java<1:3>)"
   * @exception a {@link 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 {
    SourceLocation loc = (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 {@link
   * ReferenceProbe} aspect to be class-loaded before {@link
   * Probe.MainCutter}'s advice runs. */
  static aspect MakeMyPresenceKnown dominates Probe.MainCutter {
    /** Simple advice to force the {@link ReferenceProbe} aspect
     * to be class-loaded before {@link Probe.MainCutter}'s advice
     * runs. */
    before(): mainCut(String[]) {
      ReferenceProbe.hasAspect();
    }
  }
}
```

LivenessProbe.java

```
package autoscope.probe;

// import java.util.*;

import org.aspectj.lang.*;

/**
 * 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.
 * @see autoscope.probe.DeathProbe
 * @see autoscope.probe.FinalizationProbe
 */

abstract aspect LivenessProbe extends Probe {
  /** The ID string of the RCS file.<p>Currently:
   * $Id: LivenessProbe.java,v 1.1 2002/11/25 08:05:13 mdeters Exp $ */
  protected static final String rcsid =
    "$Id: LivenessProbe.java,v 1.1 2002/11/25 08:05:13 mdeters Exp $";

  /** Returns our preferred filename extension "liv".
   * @return The {@link String} "liv", signalling that we want an
   * output file with a ".liv" extension.
   * @see Probe#outExtension() */
  protected String outExtension() { return "liv"; }

  // Ideally, we'd do this...
  //   private long Thread.frame = 0;
  // but for now, we have to do this...  :(

  /** The {@link DeathProbe.ThreadMap 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();

  Object around(): constructorCall() {
    Object o = proceed();
    Thread thr = Thread.currentThread();
    out.println("::new/" + (threads.getFrame(thr) - 1) + " " + oid(o));
    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);
  }
}
```

FinalizationProbe.java

```
package autoscope.probe;

import java.util.*;

/**
 * An aspect to detect when objects become collectible.  This aspect
 * uses advice around calls to {@link Object#finalize() finalize()} to
 * detect dead objects.
 * @see autoscope.probe.DeathProbe
 */

aspect FinalizationProbe extends LivenessProbe {
  /** The ID string of the RCS file.<p>Currently:
   * $Id: FinalizationProbe.java,v 1.1 2002/11/25 08:05:13 mdeters Exp $ */
  protected static final String rcsid =
    "$Id: FinalizationProbe.java,v 1.1 2002/11/25 08:05:13 mdeters Exp $";

  protected final Set fQ = new HashSet();

  private long Object+.birthFrame = 0;
  private Thread Object+.birthThread = null;

  // finalize() advice
  before(Object o): !target(autoscope.probe..*+) &&
    call(void *.finalize()) && target(o) {
    synchronized(fQ) {
      fQ.add(o.getClass().getName() +
             "@" + System.identityHashCode(o));
    }
  }

  /** 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 = proceed();

    // debug.println("::GC/"+frame+" after "+thisJoinPointStaticPart);

    System.gc();
    System.runFinalization();

    synchronized(fQ) {
      for(Iterator i = fQ.iterator(); i.hasNext();)
        debug.println("::DEATH_finalize "
                      + (String)i.next());
      fQ.clear();
    }

    return retval;
  }

  after() returning(Object o): constructorCall() {
    Thread thr = Thread.currentThread();
    o.birthThread = thr;
    o.birthFrame = threads.getFrame(thr) - 1;
  }

  /** Aspect containing simple advice that forces the {@link
   * FinalizationProbe} aspect to be class-loaded before {@link
   * Probe.MainCutter}'s advice runs. */
  static aspect MakeMyPresenceKnown dominates Probe.MainCutter {
    /** Simple advice to force the {@link FinalizationProbe}
     * aspect to be class-loaded before {@link Probe.MainCutter}'s
     * advice runs. */
    before(): mainCut(String[]) {
      FinalizationProbe.hasAspect();
    }
  }
}
```

DeathProbe.java

```
package autoscope.probe;

import java.lang.ref.*;
import java.util.*;

/**
 * An aspect to detect when objects become collectible.  This aspect
 * uses a {@link java.lang.ref.ReferenceQueue ReferenceQueue}
 * implementation.
 * @see autoscope.probe.FinalizationProbe
 */

aspect DeathProbe extends LivenessProbe {
  /** The ID string of the RCS file.<p>Currently:
   * $Id: DeathProbe.java,v 1.1 2002/11/25 08:05:13 mdeters Exp $ */
  protected static final String rcsid =
    "$Id: DeathProbe.java,v 1.1 2002/11/25 08:05:13 mdeters Exp $";

  /** A {@link java.lang.ref.ReferenceQueue ReferenceQueue} to use
   * for object {@link java.lang.ref.WeakReference WeakReference}s
   * becoming dead */
  protected final ReferenceQueue Q = new ReferenceQueue();

  /** A {@link java.util.Vector Vector} to keep track of currently
   * live object {@link java.lang.ref.WeakReference WeakReference}s
   * in the system.  If we didn't keep a strong reference to these
   * {@link java.lang.ref.WeakReference WeakReference}s, they would
   * be collected and not show up on our {@link #Q 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 = proceed();

    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());

    long 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();
      long birthFrame = r.getBirthFrame();

      if(birthFrame >= thisFrame) {
        out.println("::DEATH_queue " + 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 {@link
   * DeathProbe} aspect to be class-loaded before {@link
   * Probe.MainCutter}'s advice runs. */
  static aspect MakeMyPresenceKnown dominates Probe.MainCutter {
    /** Simple advice to force the {@link DeathProbe} aspect to be
     * class-loaded before {@link Probe.MainCutter}'s advice
     * runs. */
    before(): mainCut(String[]) {
      DeathProbe.hasAspect();
    }
  }
}
```

DeathProbeReference.java

```
package autoscope.probe;

import java.lang.ref.WeakReference;
import java.lang.ref.ReferenceQueue;

/**
 */

class DeathProbeReference extends WeakReference {
  /** The ID string of the RCS file.<p>Currently:
   * $Id: DeathProbeReference.java,v 1.1 2002/11/25 08:05:13 mdeters Exp $ */
  protected static final String rcsid =
    "$Id: DeathProbeReference.java,v 1.1 2002/11/25 08:05:13 mdeters Exp $";

  protected String str = null;
  protected long birthFrame;
  protected Thread birthThread;

  DeathProbeReference(Object referent, ReferenceQueue Q,
                      Thread thr, long frame) {
    super(referent, Q);

    birthFrame = frame;
    birthThread = thr;
    if(referent != null)
      str = referent.getClass().getName()
        + "@" + Integer.toHexString(System.identityHashCode(referent));
  }

  public long getBirthFrame() { return birthFrame; }
  public Thread getBirthThread() { return birthThread; }
  public String toString() { return str; }
}
```

Scopes.java

```
package autoscope.probe;

import java.util.*;
import java.io.*;

import org.aspectj.lang.reflect.*;
import util.*;

/**
 * A class to encapsulate the reading of raw data from a reference
 * trace file and develop a representation of legal scopes from it.
 * <p>For the reference trace file format, see:
 *   http://www.cs.wustl.edu/~mdeters/projects/middleware/trace.html
 */

public class Scopes {
  /** The ID string of the RCS file.<p>Currently:
   * $Id: Scopes.java,v 1.1 2002/11/25 08:05:13 mdeters Exp $ */
  protected static final String rcsid =
    "$Id: Scopes.java,v 1.1 2002/11/25 08:05:13 mdeters Exp $";

  /** The doesReference graph */
  protected DirectedGraph doesReference = new DirectedGraph();
  /** Whether or not we've coalesced strongly connected components */
  protected boolean collapsed = true;

  /** The input file buffer to read reference data from */
  protected BufferedReader infile;
  /** The line of the input file buffer currently being read */
  protected int refline = 0;

  /** Whether or not to instantiate {@link java.lang.Class} objects
   * (thus loading the classes, which takes extra time and requires
   * that they are compiled and on the <tt>CLASSPATH</tt>) */
  protected boolean instantiateClasses;

  /** Build a {@link Scopes} object with data read in from the
   * specified file with the specified {@link java.lang.Class}
   * instantiation behavior.
   * @param fname the file to open and read reference data from
   * @param instantiateClasses whether or not to instantiate {@link
   * java.lang.Class} objects for the data read in---requires that
   * the classes are compiled and on the <tt>CLASSPATH</tt>
   * @see Scopes#instantiateClasses */
  public Scopes(String fname, boolean instantiateClasses)
    throws FileNotFoundException {
    this(new FileReader(fname), instantiateClasses);
  }

  /** Build a {@link Scopes} object with data read in from the
   * specified file with the default {@link java.lang.Class}
   * instantiation behavior (which is to instantiate {@link
   * java.lang.Class} objects)
   * @param fname the file to open and read reference data from
   * @see Scopes#instantiateClasses */
  public Scopes(String fname)
    throws FileNotFoundException {
    this(fname, true);
  }

  /** Build a {@link Scopes} object with data read in from the
   * specified {@link java.io.Reader Reader} with the specified
   * {@link java.lang.Class} instantiation behavior.
   * @param in the Reader to read reference data from
   * @param instantiateClasses whether or not to instantiate {@link
   * java.lang.Class} objects for the data read in---requires that
   * the classes are compiled and on the <tt>CLASSPATH</tt>
   * @see Scopes#instantiateClasses */
  public Scopes(Reader in, boolean instantiateClasses) {
    infile = new BufferedReader(in);
    this.instantiateClasses = instantiateClasses;
  }

  /** Build a {@link Scopes} object with data read in from the
   * specified {@link java.io.Reader Reader} with the default {@link
   * java.lang.Class} instantiation behavior (which is to
   * instantiate {@link java.lang.Class} objects)
```

```
 * @param in the Reader to read reference data from
 * @see Scopes#instantiateClasses */
public Scopes(Reader in) {
  this(in, true);
}

/** Calculate the doesReference graph, if necessary, and return
 * it.
 * @return the internal {@link #doesReference} {@link
 * util.DirectedGraph DirectedGraph}, after strongly connected
 * components have been coalesced */
protected DirectedGraph calculateDoesReference() {
  if(!collapsed) {
    collapsed = true;
    doesReference =
      doesReference.coalesceStronglyConnectedComponents();
  }
  return doesReference;
}

/** Get the doesReference graph for this {@link Scopes} object.
 * @return a {@link util.DirectedGraph DirectedGraph} denoting
 * doesReference relationships; this is a clone of the internal
 * {@link #doesReference} {@link util.DirectedGraph DirectedGraph}
 * object, so you can use it without harming {@link Scopes}
 * internals */
public DirectedGraph getReferenceGraph() {
  return (DirectedGraph)calculateDoesReference().clone();
}

/** Get the legal scopes, as a directed acyclic graph, from this
 * {@link Scopes} object.
 * @return a DAG representing legal scopes */
public DirectedGraph getLegalScopes() {
  return calculateDoesReference().transpose();
}

/** Get one possible scope, as a tree, from this {@link Scopes}
 * object.
 * @return a topological ordering of the DAG nodes returned from
 * {@link #getLegalScopes()} */
public DirectedGraph getScopeTree() {
  DirectedGraph g = getLegalScopes();
  Vector verts = g.vertices(), topVerts = new Vector();

  while(verts.size() > 0) {
    topVerts.add(verts.firstElement());
    makeTreeVisitNode(g, verts);
  }

  g.newRoot(this);
  for(Enumeration e = topVerts.elements(); e.hasMoreElements();)
    g.addEdge(this, e.nextElement());

  return g;
}

/** Read all the data from the file associated with this {@link
 * Scopes} object.  The method simply calls the {@link
 * #nextLine()} method until it returns null.
 * @exception the same exceptions can be thrown as in the {@link
 * #nextLine()} method */
public void readAll()
  throws ClassNotFoundException, ParseError, IOException {
  while(nextLine() != null);
}

/** Read and process the next line of data from the file
 * associated with this {@link Scopes} object.
 * @exception a {@link ClassNotFoundException} is thrown if {@link
 * #instantiateClasses} behavior is on and the construction of a
 * {@link java.lang.Class} throws this exception
 * @exception a {@link Scopes.ParseError} is thrown with a
 * (hopefully) meaningful message if there is a problem parsing
 * the data file
 * @exception an {@link java.io.IOException IOException} can be
 * thrown by the internal {@link java.io.Reader Reader} object.
 * @return a {@link Scopes.Reference} object representing the
```

```
 * reference data from the line; <tt>null</tt> if there is no more
 * reference data to read */
public Reference nextLine()
  throws ClassNotFoundException, ParseError, IOException {
  // need to fix this to handle /* comments */ in the data file
  StringTokenizer tzr; Reference.Item left, right;

  try {
    String s = infile.readLine(), curtk = null;
    if(s == null)
      return null;
    tzr = new TrimmingStringTokenizer(s, ":(", true);
    ++refline;
    try {
      SourceLocation loc;
      String clsname;

      clsname = tzr.nextToken();
      if(tzr.nextToken().equals("("))
        loc = parseSourceLoc(tzr);
      else loc = null;
      left = instantiateClasses
        ? Reference.Item.getNew(Class.forName(clsname), loc)
        : Reference.Item.getNew(clsname, loc);

      while((curtk = tzr.nextToken(":")).equals(""));
      if(!curtk.equals(":"))
        throw new ParseError("colon separator expected", curtk);

      clsname = tzr.nextToken("(");
      if(!(curtk = tzr.nextToken()).equals("("))
        throw new ParseError("open parenthesis expected", curtk);
      loc = parseSourceLoc(tzr);
      right = instantiateClasses
        ? Reference.Item.getNew(Class.forName(clsname), loc)
        : Reference.Item.getNew(clsname, loc);
    } catch(NoSuchElementException e) {
      throw new ParseError("got end of line but expected more");
    }
    try {
      for(;;)
        if((curtk = tzr.nextToken()).length() > 0)
          throw new ParseError("junk at end of line", curtk);
    } catch(NoSuchElementException e) { }
  } catch(ClassNotFoundException e) {
    throw new ClassNotFoundException(e.getMessage()
                                     + " at line " + refline);
  } catch(IOException e) {
    throw new IOException(e.getMessage() + " at line " + refline);
  } catch(ParseError e) {
    throw new ParseError(e.getMessage() + " at line " + refline);
  }

  doesReference.addEdge(left, right);
  collapsed = false;

  return new Reference(left, right);
}

/** Parse a source location string of the form
 * "Foo.java&lt;10:20&gt;)".  The closing parenthesis at the end
 * is gobbled.
 * @param tzr a {@link java.util.StringTokenizer StringTokenizer}
 * whose current token pointer should be on the file name, as the
 * format above implies.
 * @return an object implementing the {@link
 * org.aspectj.lang.reflect.SourceLocation} interface
 * corresponding to the source location of the string tokenizer
 * passed in */
protected static SourceLocation parseSourceLoc(StringTokenizer tzr) {
  String filename, curtk = null;
  int line, column;

  try {
    filename = tzr.nextToken("<");
    tzr.nextToken();
    line = Integer.parseInt(curtk = tzr.nextToken(":"));
    tzr.nextToken();
```

```
    column = Integer.parseInt(curtk = tzr.nextToken(">"));
    tzr.nextToken();
    if(!(curtk = tzr.nextToken(")")).equals(")"))
      throw new ParseError("expected close parenthesis", curtk);
  } catch(NoSuchElementException e) {
    throw new ParseError("got end of line but expected more of new site");
  } catch(NumberFormatException e) {
    throw new ParseError("error reading numeric data in new site", curtk);
  }

  return new SourceLocationImpl(filename, line, column);
}

/** A utility routine to visit a node of a {@link
 * util.DirectedGraph DirectedGraph} during a topological
 * traversal.
 * @param g the {@link util.DirectedGraph DirectedGraph} we're
 * traversing
 * @param vxv a {@link java.util.Vector Vector} of <i>g</i>'s
 * nodes that are visitable
 * @return <i>true</i> if the {@link util.DirectedGraph
 * DirectedGraph} contains at least one node, and therefore the
 * ordering succeeded; <i>false</i> otherwise */
private static boolean makeTreeVisitNode(DirectedGraph g, Vector vxv) {
  return makeTreeVisitNode(g, vxv, null);
}

/** A utility routine to visit a node of a {@link
 * util.DirectedGraph DirectedGraph} when building a topological
 * ordering of a legal-scopes DAG.
 * @param g the {@link util.DirectedGraph DirectedGraph} we're
 * traversing
 * @param vxv a {@link java.util.Vector Vector} of nodes from the
 * {@link util.DirectedGraph DirectedGraph}---each node visited is
 * removed from this {@link java.util.Vector Vector}; an empty
 * {@link java.util.Vector Vector} indicates that we've visited
 * every node
 * @param vx the {@link Object} to visit; if <tt>null</tt>, visit
 * the first object in the <i>vxv</i> {@link java.util.Vector
 * Vector}.
 * @return <i>false</i> if object <i>vx</i> is not in the
 * <i>vxv</i> {@link java.util.Vector Vector}; <i>true</i> if the
 * object existed in the {@link java.util.Vector Vector} (and was
 * therefore removed) */
private static boolean makeTreeVisitNode(DirectedGraph g, Vector vxv,
                                         Object vx) {
  if(vx == null)
    vx = vxv.remove(0);
  else if(!vxv.remove(vx))
    return false;
  for(Iterator i = g.neighbors(vx); i.hasNext();)
    if(!makeTreeVisitNode(g, vxv, i.next()))
      i.remove();
  return true;
}

/** This {@link #main(String[]) main()} routine is a simple
 * command-line interface to the {@link Scopes} object.  Valid
 * usage is: <code>java {@link autoscope.probe.Scopes} [-n]
 * ref_file</code><p>
 * The <code>-n</code> option indicates that {@link
 * java.lang.Class} objects should not be instantiated (non-{@link
 * #instantiateClasses} behavior).  The <code>ref_file</code>
 * argument specifies the name of the file containing reference
 * trace data.
 * @param args the command line arguments */
public static void main(String[] args) {
  if(!((args.length == 2 && args[0].equals("-n")) || args.length == 1)) {
    System.err.println("Usage: java probe.Scopes [-n] ref_file");
    System.err.println("   -n  do not create java.lang.Class objects"
                       + " (no ClassNotFoundExceptions)");
    System.exit(1);
  }

  Scopes sc = null;

  try {
    sc = new Scopes(args[args.length - 1], args.length == 1);
```

```
      sc.readAll();
    } catch(Exception e) {
      System.err.println(e);
      System.exit(1);
    }

    DirectedGraph g = sc.getLegalScopes();
    System.out.println(g.toString());
    g = sc.getScopeTree();
    System.out.println(g.toString());
  }

  /** An member class used to keep track of a reference
   * (object-new-location-1 -> object-new-location-2) */
  public static class Reference {
    /** The ID string of the RCS file.<p>Currently:
     * $Id: Scopes.java,v 1.1 2002/11/25 08:05:13 mdeters Exp $ */
    protected static final String rcsid =
      "$Id: Scopes.java,v 1.1 2002/11/25 08:05:13 mdeters Exp $";

    /** The referring {@link Scopes.Reference.Item Item} */
    public Item from;
    /** The referred-to {@link Scopes.Reference.Item Item} */
    public Item to;

    /** Constructs a {@link Scopes.Reference Reference} object.
     * @param from the referring {@link Scopes.Reference.Item Item}
     * @param to the referred-to {@link Scopes.Reference.Item Item} */
    public Reference(Item from, Item to)
    { this.from = from; this.to = to; }

    /** A member class to track a reference item with new site
     * ({@link Class}, {@link
     * org.aspectj.lang.reflect.SourceLocation SourceLocation}) */
    public static class Item {
      /** The ID string of the RCS file.<p>Currently:
       * $Id: Scopes.java,v 1.1 2002/11/25 08:05:13 mdeters Exp $ */
      protected static final String rcsid =
        "$Id: Scopes.java,v 1.1 2002/11/25 08:05:13 mdeters Exp $";

      /** An {@link Scopes.Reference.Item} can be constructed
       * with a {@link Class} object or the name of a class.
       * The <i>is_cls</i> property is <i>true</i> if this
       * {@link Scopes.Reference.Item Item} contains a valid
       * <i>cls</i> property; <i>false</i> otherwise.  The
       * <i>clsname</i> is valid in either case. */
      public final boolean is_cls;
      /** The {@link Class} of the {@link Scopes.Reference.Item
       * Item}.  This is only valid if <i>is_cls</i> is
       * <i>true</i>. */
      public Class cls;
      /** The class name of the item; always valid */
      public String clsname;
      /** The {@link org.aspectj.lang.reflect.SourceLocation
       * SourceLocation} of the new site for this {@link
       * Scopes.Reference.Item Item} */
      public SourceLocation loc;

      /** Construct a new {@link Scopes.Reference.Item Item}
       * from a {@link Class} and a {@link
       * org.aspectj.lang.reflect.SourceLocation
       * SourceLocation}.<p>
       * The constructors of the {@link Scopes.Reference.Item
       * Item} class have protected access because {@link
       * Scopes.Reference.Item Item} objects should be
       * constructed through the {@link
       * Scopes.Reference.Item#getNew(Class,SourceLocation)
       * getNew()} method, described below.
       * @param cls the {@link Class} of the {@link
       * Scopes.Reference.Item Item}
       * @param loc the new site of this {@link
       * Scopes.Reference.Item Item} */
      protected Item(Class cls, SourceLocation loc) {
        this.cls = cls; this.clsname = cls.getName();
        this.loc = loc; is_cls = true;
      }
      /** Construct a new {@link Scopes.Reference.Item Item}
       * from a class name and a {@link
```

```
     * org.aspectj.lang.reflect.SourceLocation
     * SourceLocation}.<p>
     * The constructors of the {@link Scopes.Reference.Item
     * Item} class have protected access because {@link
     * Scopes.Reference.Item Item} objects should be
     * constructed through the {@link
     * Scopes.Reference.Item#getNew(Class,SourceLocation)
     * getNew()} method, described below.
     * @param clsname the name of the class of the {@link
     * Scopes.Reference.Item Item}
     * @param loc the new site of this {@link
     * Scopes.Reference.Item Item} */
    protected Item(String clsname, SourceLocation loc) {
      this.clsname = clsname;
      this.loc = loc; is_cls = false;
    }

    /** A {@link java.util.Hashtable Hashtable} of
     * already-known {@link Scopes.Reference.Item Item}s.
     * This keeps the {@link
     * Scopes.Reference.Item#getNew(Class,SourceLocation)
     * getNew()} method from instantiating two different Item
     * objects corresponding to the same class and new
     * site. */
    protected static Hashtable itemHash = new Hashtable();

    /** The public factory for getting a new {@link
     * Scopes.Reference.Item Item} object.  Constructs a new
     * {@link Scopes.Reference.Item Item} (or re-uses an old
     * {@link Scopes.Reference.Item Item}) with the specified
     * {@link Class} and {@link
     * org.aspectj.lang.reflect.SourceLocation
     * SourceLocation}.
     * @param cls the {@link Class} of the {@link
     * Scopes.Reference.Item Item}
     * @param loc the new site of the {@link
     * Scopes.Reference.Item Item}
     * @return an {@link Scopes.Reference.Item Item} object
     * corresponding to the given class and new site */
    public static Item getNew(Class cls, SourceLocation loc) {
      String key = cls.getName() + "@" + loc;
      Item it = (Item)itemHash.get(key);
      if(it == null)
        itemHash.put(key, it = new Item(cls, loc));
      return it;
    }

    /** The public factory for getting a new {@link
     * Scopes.Reference.Item Item} object.  Constructs a new
     * {@link Scopes.Reference.Item Item} (or re-uses an old
     * {@link Scopes.Reference.Item Item}) with the specified
     * class name and {@link
     * org.aspectj.lang.reflect.SourceLocation
     * SourceLocation}.
     * @param cls the name of the class of the {@link
     * Scopes.Reference.Item Item}
     * @param loc the new site of the {@link
     * Scopes.Reference.Item Item}
     * @return an {@link Scopes.Reference.Item Item} object
     * corresponding to the given class name and new site */
    public static Item getNew(String clsname, SourceLocation loc) {
      String key = clsname + "@" + loc;
      Item it = (Item)itemHash.get(key);
      if(it == null)
        itemHash.put(key, it = new Item(clsname, loc));
      return it;
    }
  }
}

/** An member class to encapsulate reference file parsing
 * problems. */
protected static class ParseError extends RuntimeException {
  /** The ID string of the RCS file.<p>Currently:
   * $Id: Scopes.java,v 1.1 2002/11/25 08:05:13 mdeters Exp $ */
  protected static final String rcsid =
    "$Id: Scopes.java,v 1.1 2002/11/25 08:05:13 mdeters Exp $";
```

```
    /** Construct a {@link Scopes.ParseError ParseError} with the
     * given message.
     * @param s the error message */
    public ParseError(String s) { super(s); }

    /** Construct a {@link Scopes.ParseError ParseError} with the
     * given message corresponding to the given token.<p>The
     * message is of the form: <code>&lt;base parse error
     * message&gt; (current token is "&lt;token&gt;")</code>
     * @param s the base error message
     * @param tok the token at which the error occurred */
    public ParseError(String s, String tok) {
      super(s + " (current token is \"" + tok + "\")");
    }

    /** Construct a {@link Scopes.ParseError ParseError} without a
     * message. */
    public ParseError() { super(); }

    /** Convert a {@link Scopes.ParseError ParseError} to a
     * string.<p>The string is of the form <code>"Parse Error:
     * &lt;message&gt;"</code> */
    public String toString() { return "Parse Error: " + getMessage(); }
  }
}
```

SourceLocationImpl.java

```
package autoscope.probe;

import org.aspectj.lang.reflect.SourceLocation;

/**
 * A class to serve as a simple implementation of AspectJ's {@link
 * org.aspectj.lang.reflect.SourceLocation SourceLocation} interface.
 * @see org.aspectj.lang.reflect.SourceLocation
 */

class SourceLocationImpl implements SourceLocation {
  /** The ID string of the RCS file.<p>Currently:
   * $Id: SourceLocationImpl.java,v 1.1 2002/11/25 08:05:13 mdeters Exp $ */
  protected static final String rcsid =
    "$Id: SourceLocationImpl.java,v 1.1 2002/11/25 08:05:13 mdeters Exp $";

  /** The filename of this source code location */
  protected String filename = null;
  /** The line number of this source code location */
  protected int line = -1;
  /** The column number of this source code location */
  protected int column = -1;
  /** The "within" (declaring) type of this source code location */
  protected Class withinType = null;

  /** Default constructor; leaves location values unspecified. */
  public SourceLocationImpl() { }

  /** Construct a {@link SourceLocationImpl} from the given file,
   * line, column, and "within" (declaring) type.
   * @param f the file name
   * @param l the line number
   * @param c the column number
   * @param w the "within" (declaring) type */
  public SourceLocationImpl(String f, int l, int c, Class w) {
    filename = f;
    line = l;
    column = c;
    withinType = w;
  }

  /** Construct a {@link SourceLocationImpl} from the given file,
   * line, and column.  The "within" (declaring) type is set to
   * <tt>null</tt>.
   * @param f the file name
   * @param l the line number
   * @param c the column number */
  public SourceLocationImpl(String f, int l, int c) {
    this(f, l, c, null);
  }

  /** Construct a {@link SourceLocationImpl} from the given {@link
   * org.aspectj.lang.reflect.SourceLocation SourceLocation}.  The
   * file name, line, column, and "within" (declaring) type are
   * taken from the parameter.
   * @param loc the {@link org.aspectj.lang.reflect.SourceLocation
   * SourceLocation} from which to construct this {@link
   * SourceLocationImpl} */
  public SourceLocationImpl(SourceLocation loc) {
    filename = loc.getFileName();
    line = loc.getLine();
    column = loc.getColumn();
    withinType = loc.getWithinType();
  }

  /** Get the file name of the {@link SourceLocationImpl}.
   * @return the file name of the {@link SourceLocationImpl} */
  public String getFileName() {
    return filename;
  }

  /** Get the line number of this {@link SourceLocationImpl}.
   * @return the line number of this {@link SourceLocationImpl}. */
  public int getLine() {
    return line;
  }
```

```
  /** Get the column number of this {@link SourceLocationImpl}.
   * @return the column number of this {@link SourceLocationImpl} */
  public int getColumn() {
    return column;
  }

  /** Get the "within" (declaring) type of this {@link
   * SourceLocationImpl}.
   * @return the "within" (declaring) type of this {@link
   * SourceLocationImpl} */
  public Class getWithinType() {
    return withinType;
  }

  public boolean isResolved() {
    return filename != null;
  }

  /** Copy values from a {@link
   * org.aspectj.lang.reflect.SourceLocation SourceLocation}.  Meant
   * to resolve instances constructed with the {@link
   * #SourceLocationImpl() default constructor}. */
  public void resolve(SourceLocation srcloc) {
    filename = srcloc.getFileName();
    line = srcloc.getLine();
    column = srcloc.getColumn();
  }

  /** Convert the {@link SourceLocationImpl} to a {@link String}.
   * The {@link String} is of the form:
   * <code>filename&lt;line:column&gt;</code>
   * */
  public String toString() {
    return filename + "<" + line + ":" + column + ">";
  }
}
```

ThreadMap.java

```
package autoscope.probe;

import java.lang.ref.*;
import java.util.*;

import org.aspectj.lang.*;

/**
 * A class to maintain a map of {@link Thread}s to stack
 * frame information
 */

class ThreadMap {
  /** The ID string of the RCS file.<p>Currently:
   * $Id: ThreadMap.java,v 1.1 2002/11/25 08:05:13 mdeters Exp $ */
  protected static final String rcsid =
    "$Id: ThreadMap.java,v 1.1 2002/11/25 08:05:13 mdeters Exp $";

  /** The internal {@link java.util.Hashtable Hashtable} */
  protected Hashtable hash;

  /** Constructs a {@link ThreadMap} with a default initial capacity
   * and load factor. */
  public ThreadMap() {
    hash = new Hashtable();
  }

  /** Constructs a {@link ThreadMap} with the specified initial
   * capacity and default load factor.
   * @param initialCapacity the initial capacity for the map */
  public ThreadMap(int initialCapacity) {
    hash = new Hashtable(initialCapacity);
  }

  /** Constructs a {@link ThreadMap} with the specified initial
   * capacity and load factor.
   * @param initialCapacity the initial capacity for the map
   * @param loadFactor the load factor for the map */
  public ThreadMap(int initialCapacity, float loadFactor) {
    hash = new Hashtable(initialCapacity, loadFactor);
  }

  /** Returns the current stack frame number of the given {@link
   * Thread}.
   * @param thr the {@link Thread} of which to query the frame.
   * @return the current frame number of the {@link Thread}
   * <i>thr</i> */
  public synchronized long getFrame(Thread thr) {
    try {
      return ((Long)hash.get(thr)).longValue();
    } catch(Exception _) {
      return -1;
    }
  }

  /** Increase the current stack frame number of the given
   * {@link Thread}.
   * @param thr the {@link Thread} for which to increment the
   * frame.
   * @return the current frame number (after incrementing) of {@link
   * Thread} <i>thr</i> */
  public synchronized long incFrame(Thread thr) {
    long l;
    hash.put(thr, new Long(l = getFrame(thr) + 1));
    return l;
  }

  /** Decrease the current stack frame number of the given {@link
   * Thread}.
   * @param thr the {@link Thread} for which to decrement the
   * frame.
   * @return the current frame number (after decrementing) of
   * the {@link Thread} <i>thr</i> */
  public synchronized long decFrame(Thread thr) {
    long l;
    hash.put(thr, new Long(l = getFrame(thr) - 1));
    return l;
```

```
  }

  // I *hate* saying "dominates" here!
  static private aspect ThreadMapMaintainer dominates LivenessProbe+ {
    Vector tms = new Vector(); // WeakVector ?

    Object around(): (call(*.new(..)) || execution(* *.*(..))) &&
      !within(autoscope.probe..*+) && !within(autoscope.runtime..*+) {
      Thread thr = Thread.currentThread();

      synchronized(tms) {
        for(Enumeration e = tms.elements(); e.hasMoreElements();)
          ((ThreadMap)((Reference)e.nextElement()).get())
            .incFrame(thr);
      }

      Object retval = proceed();

      synchronized(tms) {
        for(Enumeration e = tms.elements(); e.hasMoreElements();)
          ((ThreadMap)((Reference)e.nextElement()).get())
            .decFrame(thr);
      }

      return retval;
    }

    after() returning(ThreadMap tm): call(ThreadMap.new(..)) {
      tms.add(new MyWeakReference(tm));
    }

    before(ThreadMap tm):
      this(tm) && execution(void ThreadMap.finalize()) {
      tms.remove(new MyWeakReference(tm));
    }

    private static class MyWeakReference extends WeakReference {
      public MyWeakReference(Object referent) {
        super(referent);
      }
      public MyWeakReference(Object referent, ReferenceQueue Q) {
        super(referent, Q);
      }
      public boolean equals(Object o) {
        return (o instanceof MyWeakReference) &&
          get() == ((MyWeakReference)o).get();
      }
    }
  }
}
```

PartiallyConstructedException.java

```
package autoscope.probe;

/**
 */

class PartiallyConstructedException extends RuntimeException {
  private Object o;
  private Exception t;

  public PartiallyConstructedException(Object o, Exception t) {
    this.o = o;
    this.t = t;
  }

  public Object getObject() { return o; }
  public Exception getThrowable() { return t; }
}
```

Appendix B

Source for Instrumentation

Scope.java

```
package autoscope.runtime;

import org.aspectj.lang.JoinPoint;

import java.util.Stack;

public class Scope {
  protected static Stack currentScopes = new Stack();
  protected static Stack currentExecutionScopes = new Stack();

  protected Scope parentScope;
  String debugName;

  int refcount = 0;

  public Scope() {
    this(null, null);
  }

  public Scope(Scope _parentScope) {
    this(null, _parentScope);
  }

  public Scope(String _debugName) {
    this(_debugName, null);
  }

  public Scope(String _debugName, Scope _parentScope) {
    debugName = _debugName;
    parentScope = _parentScope;
  }

  public static Scope inScope(JoinPoint jp) {
    int ix = currentExecutionScopes.lastIndexOf(jp);
    if(ix < 0) return null;
    return (Scope)currentScopes.get(ix);
  }

  public static JoinPoint inScope(Scope sc) {
    int ix = currentScopes.lastIndexOf(sc);
    if(ix < 0) return null;
    return (JoinPoint)currentExecutionScopes.get(ix);
  }

  public void enter(JoinPoint jp, Runnable logic) {
    ScopeRunnable sr = new ScopeRunnable(jp, logic);

    synchronized(this) { ++refcount; }

    if(parentScope == null || inScope(parentScope) != null)
```

```
      sr.run();
    else parentScope.enter(jp, sr);
  }

  protected class ScopeRunnable implements Runnable {
    private Runnable innerLogic;
    private JoinPoint joinPoint;

    public ScopeRunnable(JoinPoint jp, Runnable logic) {
      innerLogic = logic;
      joinPoint = jp;
    }

    public void run() {
      synchronized(Scope.this) {
        currentScopes.push(Scope.this);
        currentExecutionScopes.push(joinPoint);
      }

      System.out.println("[ entering " + debugName + " ]");
      innerLogic.run();
      System.out.println("[ leaving " + debugName + " ]");

      synchronized(Scope.this) {
        currentExecutionScopes.pop();
        currentScopes.pop();

        if(--refcount == 0) {
          System.out.println("[[ scope " + debugName
                             + " collectible ]]");
        }
      }
    }
  }
}
```

ScopeRule.java

```
package autoscope.runtime;

import org.aspectj.lang.reflect.SourceLocation;

import java.util.*;

public class ScopeRule {
  private Class c;
  private SourceLocation srcloc = null;

  protected static Map rules = new Hashtable();

  private ScopeRule(Class _c, SourceLocation _srcloc) {
    c = _c;
    srcloc = _srcloc;
  }

  static ScopeRule getRule(Class _c, SourceLocation _srcloc) {
    Map h = (Map)rules.get(_c);
    if(h == null)
      return null;
    return (ScopeRule)h.get(_srcloc);
  }

  static ScopeRule applicableRule(Class _c, SourceLocation _srcloc) {
    Map h = (Map)rules.get(_c);
    if(h == null)
      return null;
    ScopeRule sr = (ScopeRule)h.get(_srcloc);
    if(sr != null)
      return sr;
    return (ScopeRule)h.get(null);
  }

  static ScopeRule makeRule(Class _c) {
    return makeRule(_c, null);
  }

  static ScopeRule makeRule(Class _c, SourceLocation _srcloc) {
    ScopeRule rule = getRule(_c, _srcloc);
    if(rule == null) {
      rule = new ScopeRule(_c, _srcloc);
      Map h = (Map)rules.get(_c);
      if(h == null) {
        h = new HashMap();
        rules.put(_c, h);
      }
      h.put(_srcloc, rule);
    }
    return rule;
  }
}
```

ScopeMap.java

```
package autoscope.runtime;

import org.aspectj.lang.*;
import org.aspectj.lang.reflect.*;

import java.util.HashMap;

public class ScopeMap extends HashMap {
  private Object jp2reflect(JoinPoint jp) {
    Signature sig = jp.getSignature();
    Class cl = sig.getDeclaringType();

    try {
      MethodSignature ms = (MethodSignature)sig;
      return cl.getDeclaredMethod(ms.getName(), ms.getParameterTypes());
    } catch(ClassCastException e) {
      if(!(jp instanceof ConstructorCallJoinPoint))
        throw new ClassCastException();
      return ScopeRule
        .applicableRule(cl, jp.getCorrespondingSourceLocation());
    } catch(NoSuchMethodException e) {
      throw new RuntimeException("NoSuchMethodException in "
                                 + "ScopeMap.jp2reflect() while "
                                 + "working on class " + cl.getName());
    }
  }

  public Scope getScope(JoinPoint jp) {
    try {
      Scope s = (Scope)super.get(jp2reflect(jp));
      // System.out.println("ScopeMap.getScope(\"" + jp + "\") returning " + s);
      return s;
    } catch(ClassCastException e) {
      throw new RuntimeException("ScopeMap.get() requires a "
                                 + "MethodExecutionJoinPoint or "
                                 + "ConstructorCallJoinPoint argument");
    }
  }
}
```

Params.java

```
package autoscope.runtime;

import autoscope.demo.*;
import java.util.*;

class Params {
  ScopeMap scopeMap = new ScopeMap();

  public Params() {
    Scope str = new Scope("str");
    Scope AB = new Scope("AB", str);
    Scope fooHT = new Scope("fooHT", AB);
    Scope demo = new Scope("demo", fooHT);

    scopeMap.put(ScopeRule.makeRule(String.class), str);
    scopeMap.put(ScopeRule.makeRule(A.class), AB);
    scopeMap.put(ScopeRule.makeRule(B.class), AB);
    scopeMap.put(ScopeRule.makeRule(Foo.class), fooHT);
    scopeMap.put(ScopeRule.makeRule(Hashtable.class), fooHT);
    scopeMap.put(ScopeRule.makeRule(Demo.class), demo);

    try {
      scopeMap.put(Demo.class.getDeclaredMethod("main", new Class[] { String[].class }), str);
    } catch(NoSuchMethodException e) {
      System.err.println("Uh oh.. Params() throwing " + e);
      System.exit(1);
    }
  }
}
```

DemoAspect.java

```
package autoscope.runtime;

import org.aspectj.lang.JoinPoint;
import java.util.*;

import autoscope.demo.*;

public aspect DemoAspect {
  protected Object retval = null;
  protected Params params = new Params();

  void enterScope(JoinPoint jp, Runnable logic) {
    params.scopeMap.getScope(jp).enter(jp, logic);
  }

  pointcut us(): within(autoscope.runtime..*);

  pointcut methodExecutions(): !us() && executions(* *.*(..));
  pointcut constructorCalls(): !us() && calls(*.new(..));

  around() returns Object : methodExecutions() {
    // enter scope associated with this method execution and compute
    enterScope(thisStaticJoinPoint, new RealtimeThread() {
        public void run() {
          System.err.println("-- method returning "
                             + (retval = proceed()));
        }
      });
    return retval;
  }

  around() returns Object : constructorCalls() {
    // have we already entered this scope?
    if(Scope.inScope(thisStaticJoinPoint) != null) {
      System.err.println("Scope.inScope");
      return null;
    } else {
      enterScope(thisStaticJoinPoint, new RealtimeThread() {
          public void run() {
            System.err.println("-- constructor returning "
                               + System
                               .identityHashCode(retval =
                                                 proceed()));
          }
        });
      return retval;
    }
  }
}
```

DemoAspectDebug.java

```
package autoscope.runtime;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.reflect.MethodExecutionJoinPoint;

aspect DemoAspectDebug dominates DemoAspectAssertions {
  // DEBUG advice
  around(Scope sc, JoinPoint jp) returns void:
    calls(void sc.enter(jp, Runnable)) &&
    instanceof(Scope) {

    // Argh!  AspectJ bug.  If I try to bind a "JoinPoint jp" in the
    // withincode(enterScope) construct, ajc creates bad Java code.
    // In the mean time...
    /*
      JoinPoint jp = (JoinPoint)
      ( (MethodExecutionJoinPoint)
      thisJoinPoint.getEnclosingExecutioJoinPoint() )
      .getParameters()[0];
    */

    System.out.println("entering scope for " + jp);
    proceed(sc, jp);
    System.out.println("leaving scope for " + jp);
  }

  around(Scope sc, JoinPoint jp, DemoAspect da) returns void:
    calls(void sc.enter(jp, Runnable)) &&
    instanceof(da) {

    System.out.println("entering scope for " + jp);
    proceed(sc, jp, da);
    System.out.println("leaving scope for " + jp + ", returning "
                       + System.identityHashCode(da.retval));
  }
}
```

DemoAspectAssertions.java

```
package autoscope.runtime;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.reflect.MethodExecutionJoinPoint;

privileged aspect DemoAspectAssertions {
  static void assert(boolean cond, String str) {
    if(!cond) {
      System.err.println("assertion failed: " + str);
      System.exit(1);
    }
  }

  // assertion advice
  around(Scope sc, JoinPoint jp) returns void:
    calls(void sc.enter(jp, Runnable)) {

    // Argh!  AspectJ bug.  If I try to bind a "JoinPoint jp" in the
    // withincode(enterScope) construct, ajc creates bad Java code.
    // In the mean time...
    /*
      JoinPoint jp = (JoinPoint)
      ( (MethodExecutionJoinPoint)
      thisJoinPoint.getEnclosingExecutionJoinPoint() )
      .getParameters()[0];
    */

    int numScopes = sc.currentScopes.size();
    int numExScopes = sc.currentExecutionScopes.size();

    assert(numScopes == numExScopes,
           "currentScopes(" + numScopes + ") and currentExecutionScopes("
           + numExScopes + ") different sizes "
           + "before entering scope for " + jp);

    proceed(sc, jp);

    assert(numScopes == sc.currentScopes.size(),
           "currentScopes different size after leaving ("
           + sc.currentScopes.size() + ") than before "
           + "entering (" + numScopes + ") scope for " + jp);
    assert(numExScopes == sc.currentExecutionScopes.size(),
           "currentExecutionScopes different size after leaving ("
           + sc.currentExecutionScopes.size() + ") than before "
           + "entering (" + numExScopes + ") scope for " + jp);
    /*
      JoinPoint rjp = (JoinPoint)sc.currentExecutionScopes.lastElement();
      assert(rjp == jp, "found wrong execution scope on top of stack "
      + "when leaving scope for " + jp + "; expected " + jp
      + " but got " + rjp);

      Scope rs = (Scope)sc.currentScopes.lastElement();
      assert(rs == sc, "found wrong scope on top of stack when "
      + "leaving scope for " + jp + "; expected " + sc
      + " but got " + rs);
    */
  }
}
```

References

[1] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, Boston, Massachusetts, USA, 2001.

[2] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, Boston, Massachusetts, USA, 2000.

[3] The AspectJ Organization. *Aspect-Oriented Programming for Java*, 2003. `www.aspectj.org`.

[4] The AspectJ Organization. *The AspectJ Programming Guide*, 2003. `dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/doc/progguide/`.

[5] Matthew H. Austern. *Generic Programming and the STL*. Addison-Wesley, Reading, Massachusetts, USA, 1999.

[6] Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf. The Role Object pattern. In *Pattern Languages of Programming (PLoP) Writers' Workshop on Roles and Analysis*, Monticello, Illinois, USA, September 1997. Available at `jerry.cs.uiuc.edu/~plop/plop97/Workshops.html`.

[7] William S. Beebee, Jr. and Martin Rinard. An implementation of scoped memory for real-time Java. volume 2211 of *Lecture Notes in Computer Science*, pages 289–305, Tahoe City, California, USA, October 2001. Springer-Verlag.

[8] Lodewijk Bergmans and Mehmet Aksit. Composing multiple concerns using composition filters. *Communications of the ACM (CACM)*, 44(10):51–57, October 2001.

[9] B. Blanchet. Escape analysis for object-oriented languages. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, pages

20–34, Denver, Colorado, USA, November 1999. ACM Press.

[10] 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.

[11] Greg Bollella, Ben Brosgol, Peter Dibble, Steve Furr, James Gosling, David Hardin, Mark Turnbull, and Rudy Belliardi. *The Real-Time Specification for Java*, "final v1.0" edition, November 2001. `www.rtj.org/rtsj-V1.0.pdf`.

[12] 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 conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 183–200, Vancouver, British Columbia, Canada, 1998. ACM Press.

[13] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley & Sons, New York, New York, USA, 1996.

[14] Dante J. Cannarozzi. Contaminated garbage collection. Master's thesis, Washington University in St. Louis, Department of Computer Science and Engineering, May 2003.

[15] Dante J. Cannarozzi, Michael P. Plezbert, and Ron K. Cytron. Contaminated garbage collection. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation (PLDI)*, pages 264–273, Vancouver, British Columbia, Canada, June 2000. ACM Press.

[16] Center for Distributed Object Computing, University of California at Irvine. *The ZEN ORB*, 2003. `www.zen.uci.edu`.

[17] Center for Distributed Object Computing, Washington University in St. Louis. *The ADAPTIVE Communication Environment (ACE)*, 2003. `www.cs.wustl.edu/~schmidt/ACE.html`.

[18] Perry Cheng and Guy Belloch. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN'01 conference on Programming language design and implementation (PLDI)*, pages 125–136, Snowbird, Utah, USA, June 2001. ACM Press.

[19] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 1–19, Denver, Colorado, USA, November 1999. ACM Press.

[20] Angelo Corsaro. jRate home page. `tao.doc.wustl.edu/~corsaro/jRate/`, 2003.

[21] Krysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston, Massachusetts, USA, 2000.

[22] Linda G. DeMichiel and Richard P. Gabriel. The Common Lisp Object System: An overview. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 276 of *Lecture Notes in Computer Science*, pages 151–170, Paris, France, June 1987. Springer-Verlag.

[23] Morgan Deters. Aspect-Oriented Programming with AspectJ – Course Notes: 30 October 2002. `www.cs.wustl.edu/~mdeters/seminar/fall2002/notes/1030.html`, October 2002.

[24] Morgan Deters. A case for aspect-oriented programming: The web utilities example. `www.cs.wustl.edu/~mdeters/doc/slides/darpa-01nov00.pdf`, November 2002.

[25] 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. `www.cs.ubc.ca/~kdvolder/Workshops/OOPSLA2001/ASoC.html`.

[26] 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)*, Toronto, Canada, May 2003. To appear.

[27] 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*, pages 25–30, Lancaster, United Kingdom, August 2001. Proceedings published as Technical Report CSEG/03/01 by the Computing Department, Lancaster University, Lancaster, United Kingdom.

[28] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN '98 conference on Programming language design and implementation (PLDI)*, pages 106–117, Montreal, Canada, June 1998. ACM Press.

[29] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Proceedings of the ACM OOPSLA 2000 Workshop on Advanced Separation of Concerns*, Minneapolis, Minnesota, USA, October 2000.

[30] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, USA, 1995.

[31] David Gay and Alex Aiken. Language support for regions. In *Proceedings of the ACM SIGPLAN'01 conference on Programming language design and implementation (PLDI)*, pages 70–80, Snowbird, Utah, USA, May 2001. ACM Press.

[32] Michael Golm and Jurgen Kleinoder. metaXa and the future of reflection. In *Proceedings of the ACM OOPSLA 1998 Workshop on Reflective Programming in C++ and Java*, Vancouver, British Columbia, Canada, October 1998.

[33] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Massachusetts, USA, 2000.

[34] Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In *Proceeding of the ACM SIGPLAN 2002 Conference on Programming language design and implementation (PLDI)*, pages 141–152, Berlin, Germany, June 2002. ACM Press.

[35] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 161–173, Seattle, Washington, USA, November 2002. ACM Press.

[36] William Harrison and Harold Ossher. Subject-oriented programming: a critique of pure objects. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications (OOPSLA)*, pages 411–428, Washington, DC, USA, September 1993. ACM Press.

[37] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(4):848–894, 1999.

[38] Frank Hunleth. Building customizable middleware using aspect-oriented programming. Technical Report WUCS–02–07, May 2002. Master's thesis.

[39] IBM alphaWorks. *HyperJ*, 2003. `www.alphaworks.ibm.com/tech/hyperj`.

[40] Intentional Software. `www.intentsoft.com`, 2003.

[41] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Massachusetts, USA, 1991.

[42] 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.

[43] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *Proceedings of the 5th ACM SIGPLAN'00 conference on Programming language design and implementation (PLDI)*, pages 235–248, San Francisco, California, USA, July 1992. ACM Press.

[44] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, Boston, Massachusetts, USA, second edition, 2000.

[45] Doug Lea. *Overview of package util.concurrent Release 1.3.2*, 2003. `gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html`.

[46] Karl Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, Massachusetts, USA, 1996.

[47] Karl J. Lieberherr and Ian M. Holland. Assuring good style for object-oriented programs. *IEEE Software*, 6(5):38–48, 1989.

[48] Tom Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, USA, 1997.

[49] 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, December 2002. Master's thesis.

[50] 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.

[51] 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.

[52] Robin Milner, Mads Tofte, Robert Harper, and David McQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, USA, 1997.

[53] The ML Kit. `www.it-c.dk/research/mlkit/`, 2003.

[54] Scott Nettles and James O'Toole. Real-time replication garbage collection. In *Proceedings of the conference on Programming language design and implementation (PLDI)*, pages 217–226, Albuquerque, New Mexico, USA, June 1993. ACM Press.

[55] Kelvin Nilsen. Issues in the design and implementation of real-time Java. *Java Developer's Journal*, 1(1):44, 1996.

[56] Object Management Group. *The Common Object Request Broker: Architecture and Specification (version 2.4)*, October 2000.

[57] Object Management Group. *Unified Modeling Language (UML) v1.4*, OMG Document formal/2001-09-67 edition, September 2001.

[58] Doug Orleans and Karl Lieberherr. DJ: Dynamic adaptive programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, volume 2192 of *Lecture Notes in Computer Science*, pages 73–80, Kyoto, Japan, September 2001. Springer-Verlag.

[59] Carlos O'Ryan, Fred Kuhns, Douglas C. Schmidt, Ossama Othman, and Jeff Parsons. The design and performance of a pluggable protocols framework for real-time distributed object computing middleware. In *IFIP/ACM International Conference on Distributed systems platforms (Middleware '00)*, volume 1795 of *Lecture Notes in Computer Science*, pages 372–395, Pallisades, New York, USA, April 2000. Springer-Verlag.

[60] Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek. 'C and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages*

and Systems (TOPLAS), 21(2):324–369, March 1999.

[61] Real-Time Java Working Group, International J Consortium. *Real-Time Core Extensions*, September 2000. `www.j-consortium.org/rtjwg/rtce.1.0.14.pdf`.

[62] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley Object Technology Series. Addison-Wesley, Reading, Massachusetts, USA, 1998.

[63] Alexandru Sălcianu and Martin C. Rinard. Pointer and escape analysis for multithreaded programs. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming (PPoPP)*, pages 12–23, 2001.

[64] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture, volume 2: Patterns for Concurrent and Networked Objects*. Wiley & Sons, New York, New York, USA, 2000.

[65] Lui Sha and John B. Goodenough. Real-time scheduling theory and Ada. Technical Report CMU/SEI–89–TR–014, Carnegie Mellon University, Software Engineering Institute, April 1989. `www.sei.cmu.edu/publications/documents/89.reports/89.tr.014.html`.

[66] Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. Heap profiling for space-efficient Java. In *Proceedings of the ACM SIGPLAN'01 conference on Programming language design and implementation (PLDI)*, pages 104–113, Snowbird, Utah, USA, June 2001. ACM Press.

[67] Charles Simonyi. The death of computer languages, the birth of Intentional Programming. Technical Report MSR–TR–95–52, Microsoft Research, Redmond, Washington, USA, September 1995.

[68] SPEC Corporation. Java SPEC benchmarks. Technical report, SPEC, 1999. Available by purchase from SPEC.

[69] Sun Microsystems. *Java Remote Method Invocation (RMI) Specification*, 1998. `java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html`.

[70] Sun Microsystems. *Java 2 SDK: New I/O: Documentation*, 2003. `java.sun.com/j2se/1.4/nio/`.

[71] 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.

[72] M. Tatsubori. An extension mechanism for the Java language. Master's thesis, University of Tsukuba, Graduate School of Engineering, Ibaraki, Japan, February 1999.

[73] TimeSys Corporation. *Timesys Java - Reference Implementation*, 2001. `www.timesys.com/index.cfm?hdr=java_header.cfm&bdy=java_bdy_ri.cfm`.

[74] 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.

[75] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, February 1997.

[76] TRESE Group, University of Twente. ComposeJ. `trese.cs.utwente.nl/prototypes/composeJ/`, 2001.

[77] Frédéric Vivien and Martin Rinard. Incrementalized pointer and escape analysis. In *Proceedings of the ACM SIGPLAN'01 conference on Programming language design and implementation (PLDI)*, pages 35–46, Snowbird, Utah, USA, June 2001. ACM Press.

[78] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 187–206, Denver, Colorado, USA, November 1999. ACM Press.

[79] Paul R. Wilson. Uniprocessor garbage collection techniques (long version). Submitted to ACM Computing Surveys, 1994.

Revision History

Brief revision notes for this printing and previous ones are listed below. The most recent revision and a full list of thesis errata, corresponding to all versions ever in print, are available online:

`http://www.morgandeters.com/msthesis/`

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`

---

25 Apr 2003 The copy you are now reading was typeset (TEX job *main* at 4:29am).

25 Apr 2003 Committee suggestions incorporated. Submitted to the Sever Institute of Technology and published as technical report WUCSE–03–27.

10 Mar 2003 Public thesis defense.

Vita

Morgan G. Deters

Degrees Held

Master of Science, Computer Science and Engineering
Washington University in St. Louis, May 2003

Bachelor of Science, Cum Laude, Computer Science
Bowling Green State University, May 2000

Teaching

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, St. Louis, Fall 2002. Delivered 4 lectures with Dr. Ron K. 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*, CS 6782
Washington University in St. Louis, Spring 2002.

Publications

Morgan Deters, Christopher Gill, and Ron Cytron. Rate-monotonic analysis in the C++ typesystem. To appear in *Proceedings of the RTAS 2003 Workshop on Model-Driven Embedded Systems (MDES)*, Toronto, Canada, May 2003. To appear.

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*, pages 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*, pages 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*, pages 25–30, Lancaster, United Kingdom, August 2001. Proceedings published as technical report CSEG/03/01 by the Computing Department, Lancaster University, United Kingdom.

Professional Societies

Association for Computing Machinery

May 2003

Short Title: Dynamic Assignment of RTSJ Regions Deters, M.Sc. 2003