Sustainable System Infrastructure and Big Bang Evolution:

Can Aspects Keep Pace?

by

Celina Gibbs
B.Sc., University of Victoria, 2005

Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Sustainable System Infrastructure and Big Bang Evolution:
Can Aspects Keep Pace?

by

Celina Gibbs
B.Sc., University of Victoria, 2005

Supervisory Committee

Dr. Yvonne Coady (Department of Computer Science)
Supervisor

Dr. Margaret-Anne Storey (Department of Computer Science)
Departmental Member

Dr. Hausi A. Müller (Department of Computer Science)
Departmental Member

Dr. Aaron Gulliver (Department of Electrical and Computer Engineering)
External Examiner

Supervisory Committee

Dr. Yvonne Coady (Department of Computer Science)

Supervisor

Dr. Margaret-Anne Storey (Department of Computer Science)

Departmental Member

Dr. Hausi A. Müller (Department of Computer Science)

Departmental Member

Dr. Aaron Gulliver (Department of Electrical and Computer Engineering)

External Examiner

ABSTRACT

Many rapidly evolving systems eventually require extensive restructuring in order to effectively support further evolution. Not surprisingly, these overhauls can reverberate throughout the system, forcing changes to hundreds of files. Though several studies have shown the benefits of aspect-oriented software development from the point of view of the modularization and evolution of crosscutting concerns, the question remains as to how well *aspects* fare when the code that is crosscut undergoes rapid, extensive restructuring. That is, can *aspects* keep pace when faced with a *big bang* type of evolution?

This case study demonstrates the concrete ways in which aspects impact the rapid and extensive restructuring of a memory management subsystem of a Java virtual machine. Compared with best efforts in a hierarchical decomposition coupled with a preprocessor, results show an aspect-oriented implementation fared no worse than the original in two out of four aspects, and better in the remaining two.

Table of Contents

List of Tables

List of Figures

Acknowledgements

Thank you to Robin Liu for all that he has taught me and for his help in the validation and testing of this work. Thank you to Jennifer Baldwin and Chris Matthews for their discussions, insightful questions and suggestions.

To my supervisor, Yvonne Coady, words cannot express my gratitude for your infinite supply of support, encouragement and energy. Thanks Coach.

Dedication

To my parents, for their continuing love and support. To my sisters, who taught me to treat the unexpected events that life deals as merely detours on the ultimate path to achievement. To my children, who teach me something new everyday.

To Dean, my rock, you are the calm in my storm.

Chapter 1 - Introduction

Legacy systems often face upheavals in system structure. The dawn of new system structure, marked by improved separation of concerns, is often preceded by a darkness in which the old structure must be torn down. This explosive type of evolution forces simultaneous changes throughout the code, possibly restructuring the dominant decomposition of the system. As structural boundaries of concerns are redefined, the ways in which concerns interact can also change dramatically. We term this intensive type of change to a system a *big bang* evolution.

Though aspects have been shown to be effective as a locus of control for evolving crosscutting concerns, the fact that they rely on explicit external interaction implies that they could have negative impact under these extreme conditions–when the *code that is crosscut* is undergoing a big bang evolution. The thesis of this work is that the structural support provided by aspects is sustainable when system infrastructure undergoes major structural reorganization.

By way of an introduction, Section 1.1 starts by providing the context of the thesis of this work in terms of related work. Section 1.2 follows with an overview of the experiment performed to support this thesis.

1.1 Related Work

This case study addresses questions surrounding aspect-oriented techniques within the systems domain. Specifically, we aim to answer questions about the cost of change in the context of extreme evolution. To get a clear view of the problem domain in which this

work lies, we first consider previous work from several related perspectives. Section 1.1.1 considers a software engineering perspective on structure, followed by an aspect-oriented perspective in Section 1.1.2. Sections 1.1.3 and 1.1.4 then consider related case studies in the area of Aspect-Oriented Software Development (AOSD) from systems and evolutionary perspectives, respectively. Section 1.1.5 provides related work structuring the memory management subsystem in which the experiment for this thesis takes place.

1.1.1 Software Engineering Perspective

The software engineering community has repeatedly demonstrated that modularity plays a key role in determining the cost of change. Dijkstra suggested the common model for developing programs in the 1970s was fundamentally broken [Dijkstra 1976]. He argued that designers inadequately understand most large programs they write, and proposed a model for proving the correctness of a program. The model allowed designers to decompose the problems, and ultimately the programs, into separate parts to be proven correct in relative isolation.

Parnas also reasoned about the decomposition of programs into modules and the criterion for modularity [Parnas 1972]. His work recognized the benefits modularity brings to the overall understanding of, and reasoning about, system behavior. His work suggests that the criterion for decomposing a program into modules is more *concern* based than execution based, and required that design decisions and not execution paths determine decomposition. Additionally, he observed that system behaviour is not limited to a modular view, but must also include the ways in which modules interact.

Wirth introduced a systematic approach to programming development by stepwise refinement [Wirth 1971]. He proposed the successive decomposition of a program from a top-down approach with program and data refinement done in parallel. Wirth's approach of gradual development was introduced as a method to introduce programming to students.

Early work by Murphy developed techniques to support reasoning about the structural intent of a system [Murphy 1996]. In the development and analysis of these structural views, Murphy was able to draw conclusions about the relationship between the ability to provide a structural view and the cost of change. Her work showed that a clear structural view of the relevant parts of a large system allows developers to reason in isolation about individual concerns and the changes that pertain to them.

Though structural boundaries are critical elements of software quality, unfortunately they tend to decay over time due to increasing dependencies between modules [Lehman and Belady 1985, Stevens, et al. 1974]. Structural deficiency results in the need for non-local changes that require considerable effort associated with non-local reasoning [Parnas and Clements 1990, Wulf and Shaw 1973]. Thus, evolution inevitably becomes impaired over time, as modularity naturally becomes compromised.

In [Wong 1999], the term *Big Bang* is associated with reverse engineering for improved evolvability and design as a solution to the structural decay exhibited in software over time.

1.1.2 Aspect-Oriented Perspective

AOSD supports the modular implementation of *crosscutting* concerns [Kiczales, *et al.* 1997]. It does so by allowing fragments of code that would otherwise be spread across several modular units to be co-located and to share context within a module called an *aspect*. When concerns are better separated, they are easier to reason about and work with, both independently and compositionally. Where object-oriented programming provides linguistic mechanisms designed to support the structure and modularity of object hierarchies, aspect-oriented software development provides linguistic mechanisms to support the structure and modularity of concerns that naturally cut across primary modular units of a system. The improved structure and modularity provides a high-level separation of concerns, supporting modular reasoning.

Original case studies involving aspect-oriented implementations generally reveal qualities associated with improved separation of concerns. Kersten and Murphy applied AOSD in a web-based application environment and achieved improvements in maintainability and modifiability [Kersten and Murphy 1999]. In a later study, Murphy *et al.* illustrated that AOSD can support separation of concerns at a *feature* granularity, contributing both structural improvements and identifying tradeoffs associated with refactoring [Murphy, et al. 2001]. Hanneman's aspect-oriented implementation of the Gang of Four (GOF) design patterns [Gamma, et al. 1995] showed improvements associated with modularity, such as increased locality and plugability [Hannemann and Kiczales 2002]. These case studies have demonstrated that aspect-oriented support for separation of crosscutting concerns yields the inherent improvements associated with fundamental tenets of good modularity. These results echo what Parnas identified as one

of the key contributions of modular programming–the ability to allow modules to be internally reassembled and replaced without reassembly of the whole system [Parnas 1972].

The work in the area of Multi-Dimensional Separation of Concerns (MDSOC) [Ossher and Tarr 1999] provides developers with clean partitioning of concerns along multiple dimensions. This view is intended to support a developer's view of each concern in isolation. The Hyper/J tool [Tarr and Ossher 2000] was developed to support on-the-fly, system remodularization based on MDSOC, but is not fully automated requiring developer input for integration. MDSOC provides a central composition rule in terms of concern interaction whereas with AOSD each aspect specifies its own interaction with the system, adding the complexity of interaction between aspects.

1.1.3 Systems Perspective

Low-level system infrastructures need to be fast yet flexible traits that are commonly at odds with each other. To reconcile this tension, standard practices within this domain include preprocessor directives and system patch files. These mechanisms are *de facto* standard in part because they introduce no overhead, and in part because they provide at least rudimentary means to achieve better configurability than traditional language constructs in C and Java. Though distasteful to many developers, they are a reality in today's system infrastructure software.

More recently, aspects have been offered as a potentially viable alternative associated with higher quality code refactorings [Sabbah 2004], adaptability in middleware [Zhang, et al. 2005, Duzan, et al. 2004], configurability in real time systems, and autonomic

computing in OS kernels [Engel and Freisleben 2005]. These case studies have begun to show that providing structure in systems does not necessarily mean giving up efficiency. This cross-section of system infrastructure using AOSD techniques demonstrates its potential suitability to the systems domain. For example, Fiuczynski *et al.* described current methods for patch file application to be tedious and error prone, scattering changes across a system based on line numbers [Fiuczynski, et al. 2005]. They proposed leveraging AOSD to make explicit the currently implicit crosscutting structure of these patch files.

Middleware's typically feature-oriented structure also lends itself to the application of aspect-oriented technologies. Specifically, Colyer's work has illustrated the application of AOSD in middleware at both a coarse-granularity of feature integration and a finer granularity of product line policy application [Colyer and Clement 2004]. In Tesanovic's case study, AOSD is applied to a typically hard real-time environment with rigid deadlines to provide a configurable, feedback-based, soft real-time environment for increased throughput. In both Colyer and Tesanovic's work, tradeoffs between granularity of concerns and benefits associated with modular implementations are identified [Tesanovic, et al. 2005]. Tesanovic's system showed improvements in configurability and reusability of crosscutting concerns when the scope of the aspects was limited, but at the expense of maintainability. Colyer's work illustrated the tradeoffs associated with using aspects to localize heterogeneous concerns in which diverse behaviour is witnessed at each interaction point.

The common ground between these case studies is that they demonstrate the ways in which aspects can begin to improve structure and bring modularity to concerns that do

not fit cleanly into the primary modularity of a system. In doing so, these implementations reap the fundamental benefits of modularity first established by pioneers like Dijkstra and Parnas.

1.1.4 Evolutionary Perspective

In terms of evolution, Banniassad's results showed that the lack of adequate separation of concerns can stand as an obstacle to evolution [Baniassad, et al. 2002], while Walker's work demonstrated that explicit separation can have mixed results in terms of understanding and evolving systems [Walker, et al. 1999]. Rashid's study of evolution in database systems [Rashid and Leidenfrost 2004], and previous work in operating systems [Coady and Kiczales. 2003] further showed the benefits of aspects in terms of evolution. These studies reaffirm that separation of concerns and information hiding supports change in large complex systems.

More specifically, Baniassad and Murphy's results identified that scattered crosscutting structure can in fact hinder modular reasoning. Walker and Baniassad's AOSD case study contributed experimental results that provide insight into the granularity and the scope of a crosscutting concern that developers can effectively reason about within an aspect. Coady introduced aspects into core operating systems code written in C, providing an arguably more modular implementation of non-object-oriented code. This case study showed improvements in extensibility, locality of change, and code redundancy.

1.1.5 Memory Management Perspective

In October 2001 the Jalapeño project announced the Jikes Research Virtual Machine (RVM) [IBM 2004a] as an open source, Java implementation to provide researchers with a testbed for research in virtual machine technologies. One of the core system elements that received much attention due to increasing needs for the development of new garbage collection (GC) strategies, was memory management. As a result, the original monolithic version of the memory management subsystem, known as the Jikes Memory management Toolkit (JMTk), was restructured into a more manageable and more portable toolkit, the Memory Management Toolkit (MMTk) [Alpern, et al. 2005].

Blackburn *et al.* provided details of this restructuring of JMTk into MMTk, and showed how the new modular implementation still maintained performance [Blackburn, et al. 2004]. This restructuring is illustrated in Figure 1, where we depict this original version of MMTk as MMTk[1].

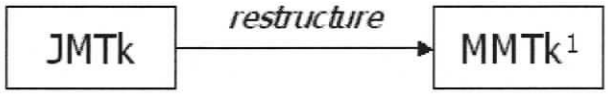


Figure 1: JMTk restructured to MMTk[1]

MMTk's modular implementation served to provide developers with improved structural clarity. This clarity is argued to be fundamental to the development of new GC strategies. The authors addressed the design's key goals by applying: (1) a clean interface between MMTk and the rest of the RVM, (2) a clean separation of policy from mechanism within MMTk, and (3) domain specific design patterns.

Blackburn's evaluation of MMTk was strictly quantitative. A combination of software engineering metrics (such as number of methods, lines of code, number of bytecodes, total cyclomatic complexity [McCabe 1976], and Lack of Cohesion of Methods (LCOM) [Henderson-Sellers 1996]) and performance benchmarks (SPEC JVM and SPEC JBB2000 [SPEC 2006]) were used to demonstrate superior code quality and performance relative to JMTk.

1.2 Evolution with Aspects: Experiment Overview

In order to investigate the impact rapid structural reorganization has on aspects, we performed an experiment to evaluate a possible evolution scenario of an aspect-oriented implementation of MMTk. Our experiment takes the work in [Blackburn, et al. 2004] a step further, introducing aspects to provide additional structure to key *crosscutting concerns* in MMTk – that is, core concerns that do not fall cleanly into the primary modularity, or the dominant decomposition, of the system.

This experiment forms the basis of this thesis and proceeds in four phases as follows: Phase I considers an alternative, aspect-oriented structural support within MMTk[1]. In this phase, MMTk[1] is manually mined for crosscutting concerns and those concerns are directly refactored as aspects to produce an aspect-oriented version, yielding $MMTk^{1}_{ao}$ illustrated in Figure 2 [Gibbs, *et al.* 2005b, Gibbs, *et al.* 2005a].

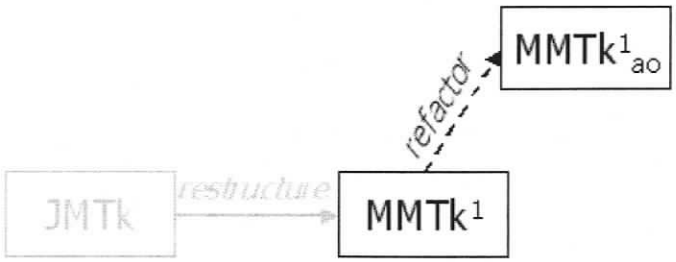


Figure 2: Phase I – Crosscutting concerns refactored in MMTk[1]

As researchers continue to develop the RVM, extensive evolution of MMTk[1] continues, rapidly disseminating further changes throughout the system. Thus, this platform provides an acid-test environment for our experiment assessing sustainability of infrastructural support mechanisms. Phase II, shown in Figure 3, examines the changes associated with this continued evolution of MMTk[1] over a ten-month period yielding MMTk[2]. These changes captured in this phase are denoted as Δ_{orig} in the figure. During this time the system underwent intense change. The boundaries of the primary modularity were redefined – modules were relocated and or eliminated, design patterns were fine-tuned to improve clarity, and new structural boundaries were established to improve portability.

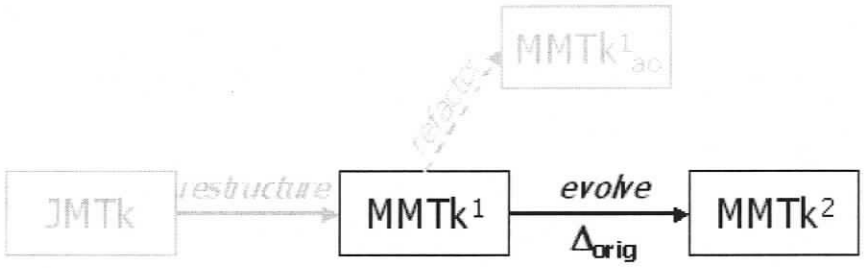


Figure 3: Phase II – Evolution continues in MMTk

In Phase III, the changes observed in the ten-month period of Phase II were directly applied to the original aspect-oriented version, $MMTk^{1}_{ao}$ yielding $MMTk^{2}_{ao}$ as is highlighted in Figure 4. The changes applied in this phase are denoted as Δ_{ao} in the figure.

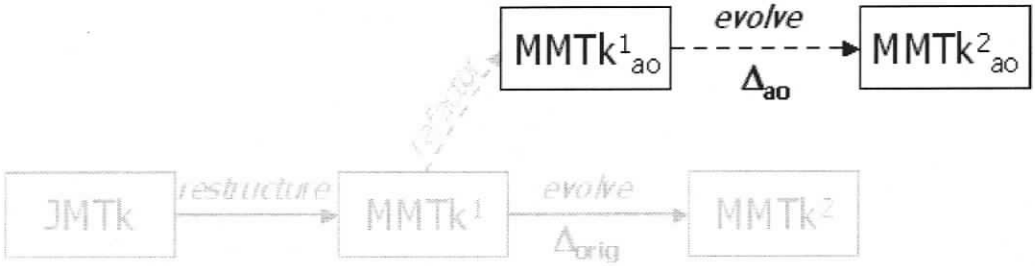


Figure 4: Phase III – Refactoring of the aspect-oriented version of MMTk

The core contribution of this work is the evaluation of the impact of this intense evolutionary change on the aspects within the restructured system. Phase IV, illustrated in Figure 5, is a detailed comparison of the evolutionary change overviewed in Phases II and III. That is, Phase IV compares and contrasts the evolution of $MMTk^{1}$ to that of $MMTk^{1}_{ao}$. Figure 5 additionally provides an overview and an approximate timeline of each phase of the experiment.

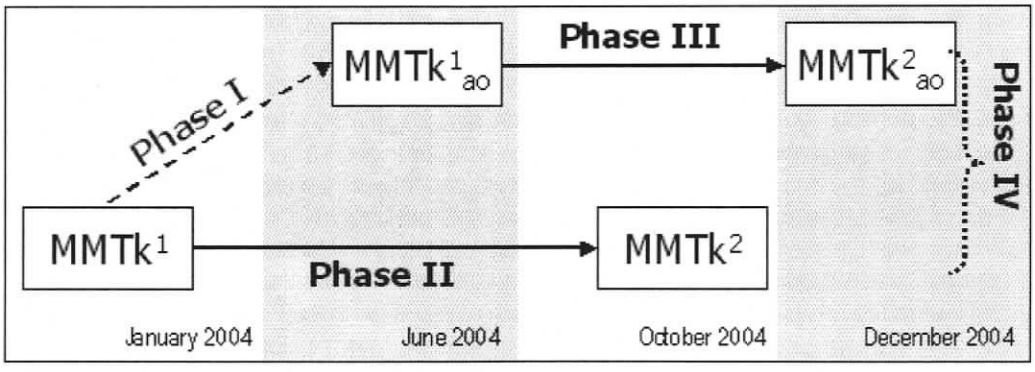


Figure 5: Phase IV – Evolution of MMTk verses evolution of $MMTk_{ao}$.

A key contribution of this experiment is that it is based on a real open source system undergoing real evolutionary change. The evolutionary changes to MMTk were performed as part of a collaborative memory management research project [Alpern, et al. 2005]. This evolution is motivated by core software engineering principles, and simultaneously bound by the inherent complexity of systems code. To support this reality, low-level mechanisms for efficiency were leveraged in the original evolution that would not typically be considered to support good software engineering practices.

This experiment provides a challenge for aspects due to the fact that crosscutting concerns and their interaction with other concerns were rapidly changing concurrently throughout this evolution. That is, we were able to better establish the value of both the internal structure and the external interaction explicitly defined by aspect-oriented mechanisms. Additionally, we were able to study the evolution of crosscutting and interacting concerns, the semantics of the woven system as a whole, and the evolution of the dependencies between the concerns.

1.3 Chapter Summary

This chapter provided a high-level overview of the experiment of this thesis, in which the isolation of a version of an evolving system and the application of AOSD mechanisms for separation of concerns was achieved. Changes to the original system by developers continued and these changes were then directly applied in this experiment to the aspect-oriented version of the system.

The subsequent chapters of this thesis now take on each phase of the experiment in turn Chapter 2 focuses on Phase I, the integration of aspect-oriented techniques and MMTk's

original structural support. Examples representative of three types of crosscutting concerns are established and are refactored into aspects. The structural benefits and tradeoffs between the aspect-oriented version and the original implementation are considered. Chapter 3 presents Phases II and III of the experiment. Phase II involves the extensive restructuring of the memory management subsystem over a ten-month period. This evolution involved a complete revamping of the current infrastructure. The consequential changes are identified and captured. Phase III involves the experimental evolution of $MMTk^{1}_{ao}$, applying the structural change tasks identified in Phase II to the aspect-oriented version of the system. Chapter 4 presents Phase IV, a comparison of the evolution of the original version of the system with the aspect-oriented version. Chapter 5 concludes with a summary, limitations of this experiment, and future work.

Chapter 2 - Memory Management, Crosscutting Concerns and Aspects

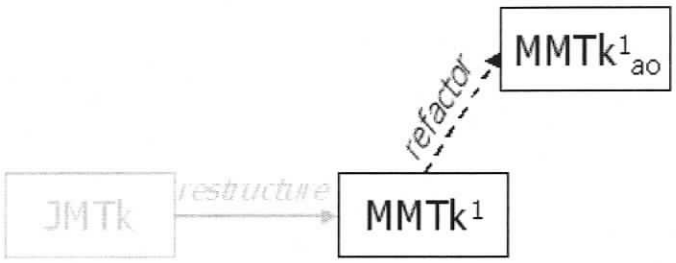


Figure 6: Phase I – Crosscutting concerns refactored as aspects

This chapter introduces key concepts, terminology and background on the code base that was used, followed by the details of Phase I of this experiment. This phase covers the mining and refactoring of crosscutting concerns in the memory management subsystem as aspects [Gibbs and Coady 2005]. The chapter proceeds as follows. Section 2.1 provides a brief introduction to aspect-oriented software development (AOSD) with specific details of AspectJ [Kiczales, et al. 2001]. Section 2.2 provides background on the Jikes Research Virtual Machine (RVM) [IBM 2004a], including its memory management subsystem MMTk [Alpern, et al. 2005], which is the core platform of this experiment. Section 2.3 provides the details of the refactoring in Phase I involving the application of aspect-oriented techniques to $MMTk^1$ highlighted in Figure 6.

2.1 Aspect-Oriented Software Development with AspectJ

The primary unit of modularity in object-oriented languages, such as Java, is the class. AspectJ [Kiczales, et al. 2001] is a simple extension that adds aspect-oriented software

development capabilities to Java. In AspectJ, the primary unit of modularity for crosscutting concerns is the aspect. The following is a brief introduction to AspectJ, a more complete treatment of this topic can be found in [IBM 2006].

AspectJ can be described in terms of five key mechanisms: *joinpoints*, *pointcuts*, *advice*, *intertype declarations* and *aspects*. Multiple types of **joinpoints** exist, for example `call` and `execution` joinpoints. A method `call` joinpoint corresponds to all the points of execution in a program when that method is explicitly called, whereas a method `execution` joinpoint corresponds to all the points when a method executes. Other mechanisms extract values from joinpoints, they are: `this`, which matches a calling object of a specified type; `target`, which matches and selects a receiving object of a specified type; and `args`, which matches parameter(s) of the specified type(s).

Pointcuts provide a mechanism to identify and select joinpoints either anonymously or as named, reusable units. They can be composed and precisely target a single method, or groups of methods, in terms of specific signatures. Leveraging lightweight pattern matching, inheritance and combinations using logical operators, a pointcut can attain a semantic representation of a group of related joinpoints. AspectJ provides the `within` mechanism to narrow the scope of a match to be contained by a specified package, class or method.

Advice defines functionality associated with an aspect, and precisely when it executes with respect to `before`, `after` or `around` a specified pointcut. The `before` and `after` advice execute before and after a pointcut respectively, while `around` advice has explicit control over the execution of the targeted joinpoint and the values passed into that joinpoint with the key word `proceed`.

An **intertype declaration** provides a mechanism to introduce new modular units such as classes, interfaces, methods and fields into another module. In particular the `declare parents` intertype declaration introduces hierarchal structure between modules, as shown in Figure 7.

An **aspect**, much like a class in Java is a modular unit representing a crosscutting concern composed of joinpoints, pointcuts, advice and Java constructs such as classes, interfaces, methods, and fields.

Reviewing these mechanisms together, joinpoints are principled points of execution in a program that can be picked out and represented in a meaningful way by pointcuts. Advice associated with a pointcut provides the mechanism to define the actions to take place at the joinpoints picked out by a pointcut. AspectJ encapsulates a crosscutting concern into a single module called an aspect. Intertype declarations allow an aspect to explicitly introduce functionality into, and establish relationships between, other modules.

For example, the piece of sample code shown in Figure 7 uses the `declare parents` intertype declaration to force a group of classes to implement a specified interface. An anonymous pointcut with a wildcard (`*`) matches all classes in `org.mmtk` or (`||`) the `com.ibm.JikesRVM.memoryMangers.mmInterface` packages. The number of matches is further limited, by listing all classes within the specified packages that will not (`!`) be included in the pointcut. Ultimately, all classes within `org.mmtk` and `com.ibm.JikesRVM.memoryMangers.mmInterface` with the exception of the ones listed in the negation are forced to implement `VM_Uninterruptible`, a low-level empty tag interface used for synchronization.

```
aspect Synchronization {

  declare parents :
    (org.mmtk.* || com.ibm.JikesRVM.memoryMangers.mmInterface.*)
     && !( *Header
              || org.mmtk.utility.AllocAdvice
              || com.ibm.JikesRVM.memoryManagers.mmInterface.MM_Constants
              ... )

   implements VM_Uninterruptible;
}
```

Figure 7: Sample aspect using intertype declaration

In contrast, the aspect in Figure 8 is composed of two pointcuts and one advice. The first named pointcut, `GCstrategy`, matches all classes within `org.mmtk`, and the next combines this first pointcut with a method call pointcut. The pointcut, `asserting`, matches all calls to `Assert._assert`, accessing the parameter of that method using `args`. The parameter type in the method signature is matched with the parameter type of the named pointcut. The advice in this aspect executes around the `asserting` pointcut, only proceeding with the execution of the joinpoint if the `Assert.VerifyAssertions` flag is set. The parameter remains unchanged and gets passed to `proceed` directly.

```
aspect VerifyingAssertions {

   pointcut GCstrategy():
       within(org.mmtk.*);

   pointcut asserting(boolean condition):
       call(void Assert._assert(boolean))
       && args(condition)
       && GCstrategy();

   void around(boolean b): asserting(b){
       if (Assert.VerifyAssertions)
            proceed(b);
   }

   ...
}
```

Figure 8: Sample aspect using named pointcuts and around advice

2.2 Jikes Research Virtual Machine (RVM)

The Jikes Research Virtual Machine (RVM) [IBM 2004a] is one of the first Java Virtual Machines with its core implementation written in Java. The Jikes RVM is a hotbed of research activity at more than sixty universities and affords researchers the opportunity to experiment with a variety of design alternatives in virtual machine infrastructure. Its core Java implementation allows researchers to investigate the impact of object-oriented structural support mechanisms in a systems environment. It is in this vein that the RVM's infrastructure is, at any given time, undergoing vast restructuring and evolutionary change in at least one area.

The subsystem this thesis focuses on is garbage collection (GC). One of the appealing features of development in a language like Java is its ability to shield developers from the inherent complexities associated with memory management. The introduction of automated GC into the development environment alleviates a programmer's burden of handling memory management by eliminating a large class of exceptions. The underlying mechanism that supports this automation continues to be a highly active area of research in the way of improved collection strategies, real-time memory management, concurrent garbage collectors, and benchmarking, among others.

As previously overviewed in Section 1.1.5, memory management within the Jikes RVM provides researchers interested in GC with a clean development environment in which to explore new GC strategies. However, the original memory management subsystem in the RVM, the Java Memory management Toolkit (JMTk), was a purely monolithic Java implementation. In order to provide a more modular environment conducive to experimentation, this monolithic implementation was significantly

restructured to leverage a more object-oriented design and implementation. This newly restructured implementation, the Memory Management Toolkit (MMTk), is the GC subsystem used in this thesis.

2.2.1 MMTk Background: Dominant Decomposition of GC

The benefits of GC are well known and have been appreciated for many years in many programming languages. GC separates memory management issues from program design – increasing reliability and eliminating memory management errors. Collection strategies have improved significantly over the last ten years, and ongoing work aims to further reduce costs and meet application-specific demands [Goetz 2003].

Costs not only involve performance impact, but also configuration complexity. In JDK 1.4.1, for example, there are six collection strategies and over a dozen command line options for tuning GC [Goetz 2003]. Basic strategies, such as reference counting, mark and sweep, and copying between semi-spaces, have been augmented with hybrid strategies, such as generational collectors, that treat different areas of the heap with different collection algorithms. Existing collectors not only differ in the way they identify and reclaim unreachable objects, but also in the ways they interact with user applications and the scheduler.

Plans and *policies* are two dominant structural elements within the RVM's memory management system. Plans are crosscut by all three of the concerns in this study, whereas the relationship between plans and policies is the focus of the *Prepare/Release* design pattern.

The plan and policy classes span three packages and separate hierarchies, as overviewed in Figure 9: *Plan* and *policy* class hierarchies, highlighting the `copyMS` plan. Each plan details a particular overall configuration for GC, where the specific policies for heap space management are one element of this configuration. The policies that make up each plan each have their own strategies for allocation and collection of memory space, which are drawn from basic allocation and collection mechanisms.

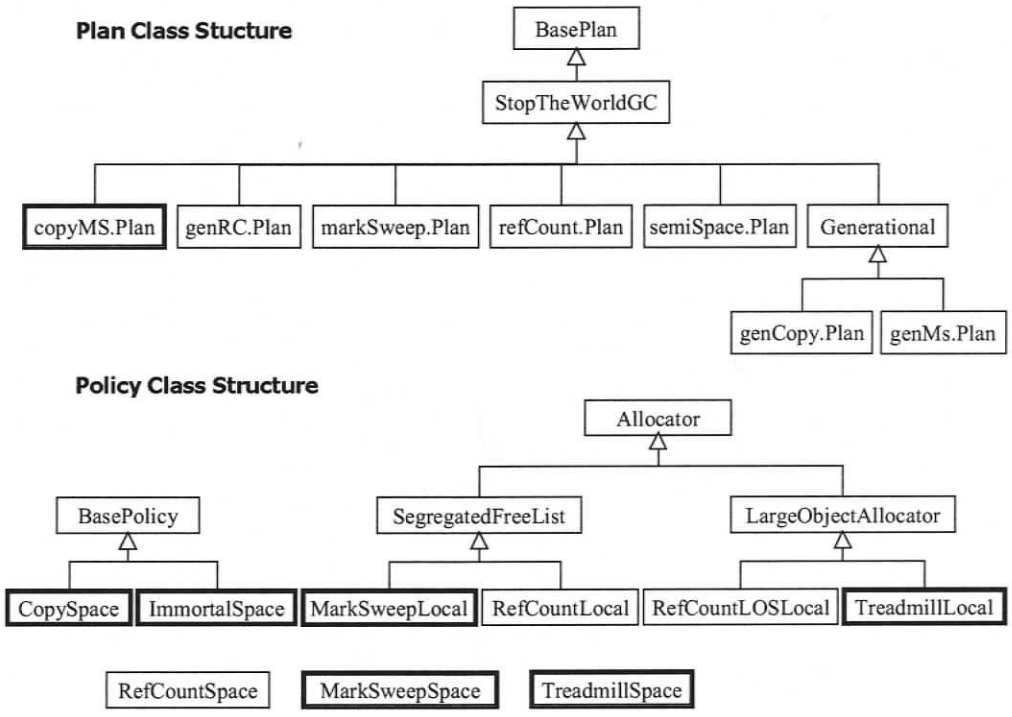


Figure 9: *Plan* and *policy* class hierarchies, highlighting the `copyMS` plan

The combinations of the mechanisms provided by the policies form the GC plans available in the Jikes RVM. For example, consider the case of the *copying mark and sweep* GC plan, highlighted in Figure 9: *Plan* and *policy* class hierarchies, highlighting the `copyMS` plan and residing in the `copyMS` package. This is a non-generational,

copying/mark-sweep hybrid collector. The policies employed by this plan are `CopySpace`, `ImmortalSpace`, `MarkSweep` and `Treadmill` also highlighted in Figure 9: *Plan* and *policy* class hierarchies, highlighting the `copyMS` plan (for a more detailed treatment of policy specifics see [Blackburn, et al. 2004]).

In more detail, the top half of Figure 9: *Plan* and *policy* class hierarchies, highlighting the `copyMS` plan shows that all current plans extend the `BasePlan` class. In the plan package, `BasePlan` holds the framework for all memory management schemes. `StopTheWorldGC` extends `BasePlan` and implements core functionality for the stop-the-world collector plans. Stop-the-world collectors require that all other threads be suspended while collection takes place.

The bottom half of Figure 9: *Plan* and *policy* class hierarchies, highlighting the `copyMS` plan highlights the hierarchical structure of policy. These classes are split over two packages. Some *space* policies inherit from the `BasePolicy` class in the `policy` package, *local* policies inherit from `Allocator` classes in the `utility` package, while other space policies do not extend any class.

2.3 Phase I: Refactoring of MMTk[1] to MMTk[1]$_{ao}$

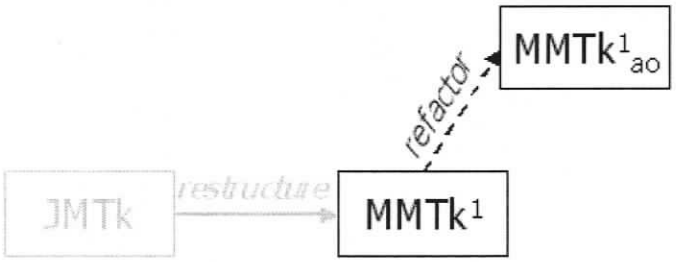


Figure 10: Phase I – Refactoring crosscutting concerns in MMTk

This section presents the details of Phase I of this experiment, applying aspects to a select group of crosscutting concerns within MMTk. The crosscutting concerns identified and refactored in this phase were selected from the design descriptions of concerns that were difficult to modularize as described in [Blackburn, et al. 2004] and [Sun 2004]. This subset resulted in representative concerns from three categories: (1) design invariants, (2) analysis tools, and (3) domain specific design patterns. Specifically, two design invariants are identified, the first being a low-level flag mechanism for assertion verification and the other handling coarse-grained synchronization. The analysis tool, *GCSpy* was chosen for two reasons with the first being its objective to provide minimally invasive implementation of monitoring a system and the second being its inherently wide scope in terms of the number of modules and packages it is dependant on. The *Prepare/Release* design patterns was chosen from a group of domain specific design pattern identified by Blackburn as a key structural component of MMTk.

The following subsections provide details of these concerns and their refactoring as aspects, along with an initial analysis of the structural benefits and tradeoffs of each. They proceed as follows. Section 2.3.1 provides an overview of the originally scattered implementation of the crosscutting concerns involved in the study. Sections 2.3.2 through 2.3.5 provide a high-level overview of the structure of these crosscutting concerns as aspects, along with the structural benefit and tradeoffs of each.

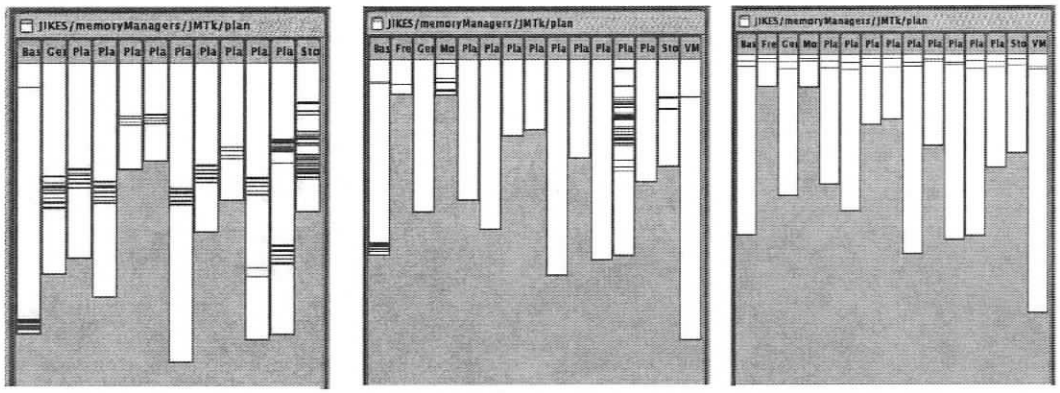
2.3.1 Identification of Crosscutting Concerns

The memory management subsystem consists of 134 classes, and supports nine different garbage collection (GC) strategies. But even within this well modularized implementation, some concerns naturally defy traditional structural boundaries. They are scattered the throughout the GC infrastructure, and tangled with the implementation of other concerns in an unclear way.

This section describes four crosscutting concerns in their scattered form within MMTk[1] [Gibbs and Coady 2005, Gibbs and Coady 2004a, Gibbs and Coady 2004b]: two design invariants, a heap visualization tool [Sun 2004], and a domain specific design pattern within GC plans [Blackburn, et al. 2004]. A high-level representation of the original scattered implementation of these concerns is illustrated in Figure 11. This figure is made up of three images highlighting three concerns where the white boxes represent classes in the system and the horizontal lines represent the instrumentation of the highlighted concern. These images in Figure 11 are generated with an aspect-mining tool [Hannemann 2003].

Taking the synchronization concern as representative of a design invariant, this concern provides low-level support for concurrency control by signifying that all methods of a class implementing the `VM_Uninterruptible` interface are uninterruptible. By far the majority of the classes in this subsystem implement this interface. This concern crosscuts 108 of the 134 classes, and a small portion of it is represented in Figure 11(a). The assertion verification invariant provides the mechanism to check if assertions are on, and uses a low-level global flag that is checked before every assertion. The fine-grained scattering of the implementation of this concern is similar to that of synchronization.

The heap visualization tool, *GCSpy*, allows developers to perform dynamic analysis of memory consumption. Its implementation is currently in place for one of nine GC plans, and crosscuts six classes. Figure 11(b) shows a high-level view of each of the key classes involved (including the eight plans that remain to be instrumented with this tool). Each of the vertical rectangles in the figure represents a source file for a class, and the horizontal lines mark the places in the code that involve this concern.



(a) **Design Invariant:** shows a small subset of the 108 classes implementing VM_Uninterruptible

(b) **Analysis Tool:** shows the implementation of 1 of the 9 GC plans, additional functionality required in utility classes

(c) **Design Pattern:** shows the protocol across all plans and the associated infrastructure

Figure 11: High-level view of scattered implementations of crosscutting concerns

The domain specific design pattern, *Prepare/Release* is a staged protocol that must proceed through a sequence of *global* and *local* activities for *prepare* and *release* phases of multithreaded GC. Every plan must execute these activities symmetrically (matching local/global and prepare/release). Each GC plan implements a different algorithm or strategy for heap management, which is defined in the *prepare* and *release* phases. The prepare phase acquires memory for collection which the release phase then recovers. Specific strategies are composed of combinations of the heap management policies

available in the Jikes RVM. The specific policies involved in these phases differ on a per-plan basis. Figure 11(c) illustrates the implementation of this pattern in the twelve classes it crosscuts.

Each of these concerns provides a fundamentally different kind of system element. The interface for synchronization and the flag for assertion verification design invariants we would like to configure holistically; *GCSpy* is a tool we would like to plug/unplug as necessary; and *Prepare/Release* design pattern is a protocol we would like to consistently enforce across plans. But, the same deep structural flaw impairs their evolution and adaptation: a lack of modularity.

The next four sections summarize the key features of the aspect-oriented implementation of these four scattered crosscutting concerns: the synchronization mechanism as the `Synchronization` aspect, the assertion verification as the `VerifyingAssertion` aspect, the *GCSpy* analysis tool as the `GCSpy` aspect, the *Prepare/Release* design pattern as the `PR_Protocol` aspect. In each case, we manually refactored the original system to exclude the concern involved, and reintroduced the concern with the aspect using AspectJ [Kiczales, et al. 2001]. Sections 2.3.2 and 2.3.3 begin with the structural details of the design invariant aspects [Gibbs and Coady 2004c], followed by `GCSpy` aspect in Section 2.3.4, and concluding with the `PR_Protocol` aspect in Section 2.3.5.

2.3.2 Design Invariant: Synchronization

The `VM_Uninterruptible` interface is necessary for correct synchronization within the RVM. When a class implements this interface, it signifies that its methods are not

interruptible (unless a finer-granularity of method-level synchronization is used). No explicit functionality is required for classes implementing this interface. Since 108 out of 134 classes within the memory management subsystem implement this interface, it is more succinct to specify which classes do *not* implement it. Our aspect for this concern is shown in Figure 12.

```
privileged aspect Synchronization {

  declare parents :
    (org.mmtk.* || com.ibm.JikesRVM.memoryMangers.mmInterface.*)
     && !( *Header
       || org.mmtk.utility.AllocAdvice
       || org.mmtk.utility.TracingConstants
       || org.mmtk.utility.CallSite
       || org.mmtk.policy.BasePolicy
       || org.mmtk.vm.ScanStatics
       || org.mmtk.vm.Constants
       || com.ibm.JikesRVM.memoryManagers.mmInterface.MM_Constants
       || com.ibm.JikesRVM.memoryManagers.mmInterface.SynchronizationBarrier
       || com.ibm.JikesRVM.memoryManagers.mmInterface.VM_CollectorThread
       || com.ibm.JikesRVM.memoryManagers.mmInterface.VM_GCMapIteratorGroup
       || com.ibm.JikesRVM.memoryManagers.mmInterface.VM_Handshake )

      implements VM_Uninterruptible;
}
```

Figure 12: `Synchronization` aspect

This aspect uses `declare parents`, forcing classes to implement the specified interface. This allows us to centrally identify all classes that do not (!) implement the interface. Wildcards (*) are used in this expression to capture all packages in `org.mmtk` or `com.ibm.JikesRVM.memoryManagers.mmInterface` and all classes that match `*Header`. As a result of this aspect, classes not captured in this list (including, by default, new classes added to the subsystem that do not match `*Header`) implement the synchronization mechanism. Though finer-grained synchronization mechanisms, such as `throws VM_PragmaUninterruptible` exist elsewhere in the code, this aspect centralizes configurability of this particular low-level mechanism.

2.3.3 Design Invariant: Assertion Verification

Jikes RVM uses a `boolean` field called `VerifyAssertions` as a flag to enable certain activities related to assertion handling. In Version 2.3.3 of the Jikes RVM, the `VM` class has a comment that alludes to the structure of this design:

```
/*
 * Note: code your assertion checks as
 * "if (VM.VerifyAssertions) VM._assert(xxx);"
 */
```

and an error is logged if this check is not done:

```
if (!VM.VerifyAssertions) {
      sysWriteln("vm: somebody forgot to
            conditionalize their call to assert ");
      ...
}
```

In this version all calls to `VM_Interface._assert`, `VM_Interface.sysFail` and `VM_Interface.spaceFailure` are preceded by a check of this flag that resides in that same class.

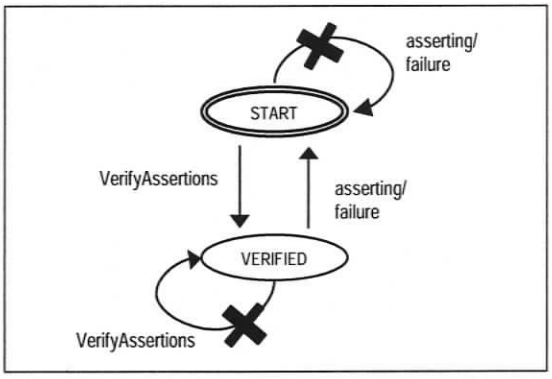


Figure 13: Assertion verification design invariant

The structure the `VerifyingAssertions` aspect is based on the two-stage design invariant illustrated in Figure 13. This simple invariant has a START state and a VERIFIED state, and the transition from the START state to the VERIFIED state is

triggered by a check of the `VerifyAssertions` flag. Once in the VERIFIED state, the assertion methods, `asserting` and `failing`, can execute and force transition back to the START state until the precondition is met again. As illustrated, consecutive transitions to the same state are violations of the design invariant, disallowing multiple verifications for each assertion or multiple assertions on one verification. This condition enforces the intent of the invariant.

```
privileged aspect VerifyingAssertions {

    pointcut GCstrategy():
        within(org.mmtk.*);

    pointcut asserting(boolean condition):
        call(void VM_Interface._assert(boolean))
        && args(condition) && GCstrategy();

    pointcut failing(String msg):
        call(void VM_Interface.sysFail(String))
        && args(msg) && GCstrategy();

    pointcut space_failure(VM_Address obj, byte space, String source):
        call(void Plan.spaceFailure(VM_Address, byte, String))
        && args(obj, space, source) && GCstrategy();

    void around(boolean b): asserting(b) {
        if (VM_Interface.VerifyAssertions)
            proceed(b);
    }

    void around(String str): failing(str) {
        if (VM_Interface.VerifyAssertions)
            proceed(str);
    }

    void around(VM_Address obj, byte space, String source):
          space_failure(obj, space, source) {
         if (VM_Interface.VerifyAssertions)
            proceed(obj, space, source);
    }
}
```

Figure 14: `VerifyingAssertions` aspect

Figure 14 shows the implementation of this invariant, defining pointcuts for the two failure method calls and one asserting call within `org.mmtk` package. Three around

advice attach to these pointcuts and only proceed with execution of the target methods if the `VerifyAssertions` flag is set.

2.3.4 Analysis Tool: *GCSpy*

The implementation of *GCSpy* involves instrumentation within the RVM in order to establish two things: (1) to gather data before and after garbage collection, and (2) to connect a *GCSpy* server and client-GUI for heap visualization. These changes require that existing methods be augmented, and new methods be added.

Table 1: Classes requiring instrumentation of `GCSpy` flag

CLASS	OCCURRENCES
Plan (SemiSpace)	4
MMInterface	2
StopTheWorldGC	4
FreeListResource	1
MonoToneVMResource	2

Part of the configuration strategy in the Jikes RVM involves checking a global flag, `if (VM_Interface.GCSPY)`, before invoking *GCSpy* functionality. This flag is set when the system is built using the *GCSpy* option. In total, instrumentation appears in thirteen places over five classes as shown in Table 1.

In the aspect-oriented implementation, all of this instrumentation has been removed from the existing methods in the system, and reintroduced by code that resides in one module – the `GCSpy` aspect. A portion of this code is shown in Figure 15. In general, this figure shows that the *GCSpy* related activity is performed before methods `boot`, `postCopy`, and `allocCopy`, and potentially instead of (`around`) the `postAlloc` method in

the `semiSpace` collector's `Plan` class. This collector is the only plan shown in Figure 11(a) that is instrumented with *GCSpy*, and is used as a starting point for developers to understand how to extend this implementation to other plans.

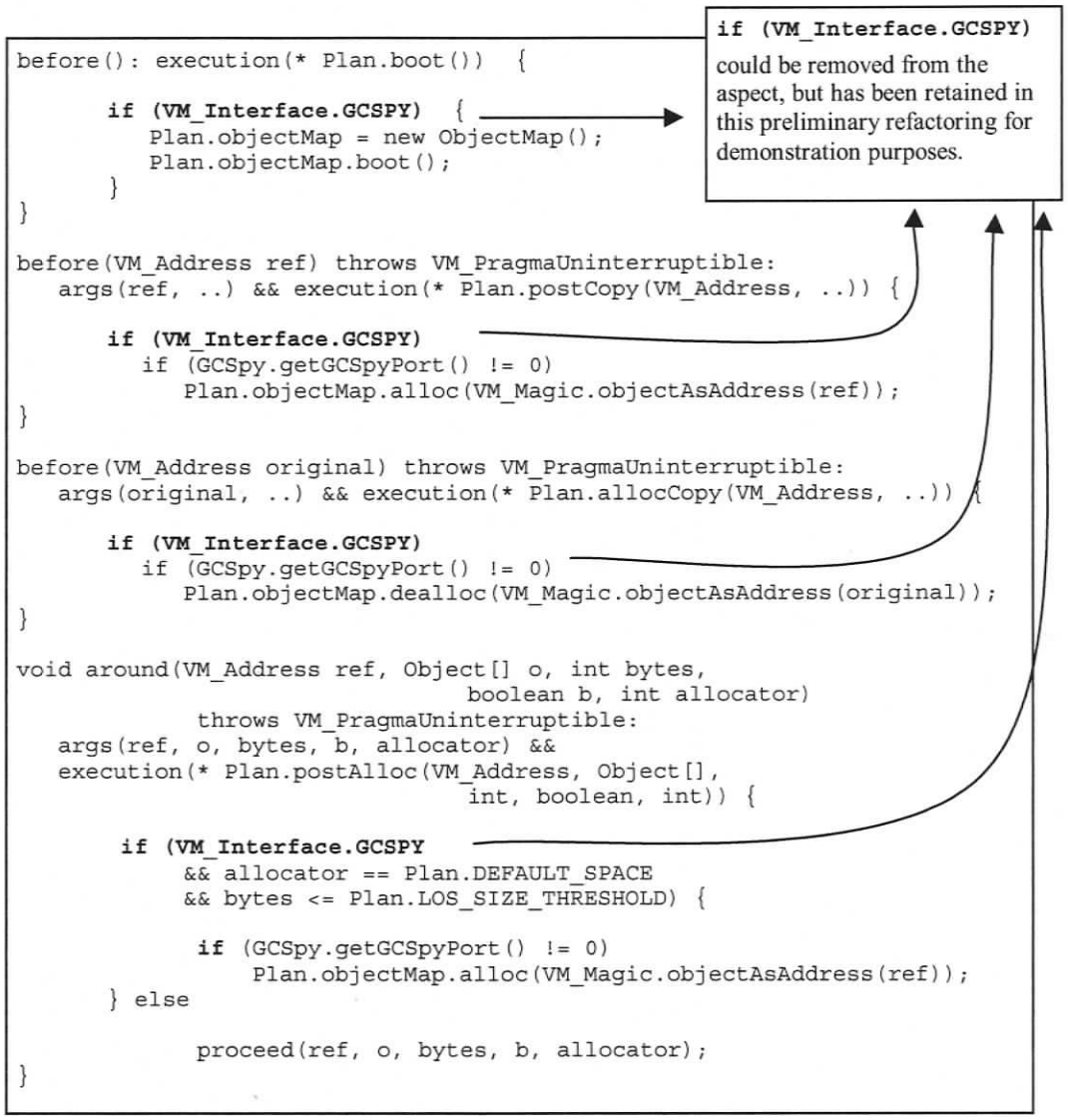


Figure 15: `GCSpy` aspect, a portion associated with flags in the `SemiSpace` plan class

In addition to this instrumentation, new methods must be added to existing classes from within the aspect. In total, thirteen new *GCSpy* methods appear across three classes as reported in Table 2.

Table 2: Classes requiring new *GCSpy* methods

CLASS	OCCURRENCES
Plan (SemiSpace)	6
BasePlan	5
MonoToneVMResource	2

Using AspectJ, *GCSpy* methods are introduced to these classes from within the `GCSpy` aspect by simply identifying the intended *<class>.<method>*. For example:

```
public void BasePlan.gcspyPrepare() {}
```

introduces the (empty) `gcspyPrepare()` method to the `BasePlan` class. All plans thus inherit this method, and may override it appropriately.

This aspect can be (un)plugged at compile time to any other GC plan that adheres to this interface. Given that all collectors have a similar structure, the `GCSpy` aspect is thus more easily extensible between plans than manual instrumentation. Additionally, as indicated in Figure 15, the `GCSpy` flag is not necessary, as compiling with an aspect yields an all-or-nothing result.

2.3.5 Design Pattern: *Prepare/Release*

Currently, MMTk comes with nine different GC strategies. Advice for developers who want to add a new plan can be found in the Jikes User Guide [IBM 2004b]:

> *"A good way to start is to compare some of the different plans and understand the significance of the differences."*

A comparison of plan classes shows a clear division of *global* and *thread-local* activities. Global activities require synchronization between collector threads on different processors in a multiprocessor environment, while thread-local activities do not.

Figure 16 highlights four of the key methods within the control-flow of `StopTheWorldGC.collect()`, showing where plans interact with policies. In this figure the control flow of the main thread, the collector thread, is highlighted by the large grey arrow. The collector thread flows through a four staged protocol, resulting in sequential calls to the four methods highlighted in the grey ovals in Figure 16: `globalPrepare()`, `threadLocalPrepare()`, `threadLocalRelease()` and `globalRelease()`.

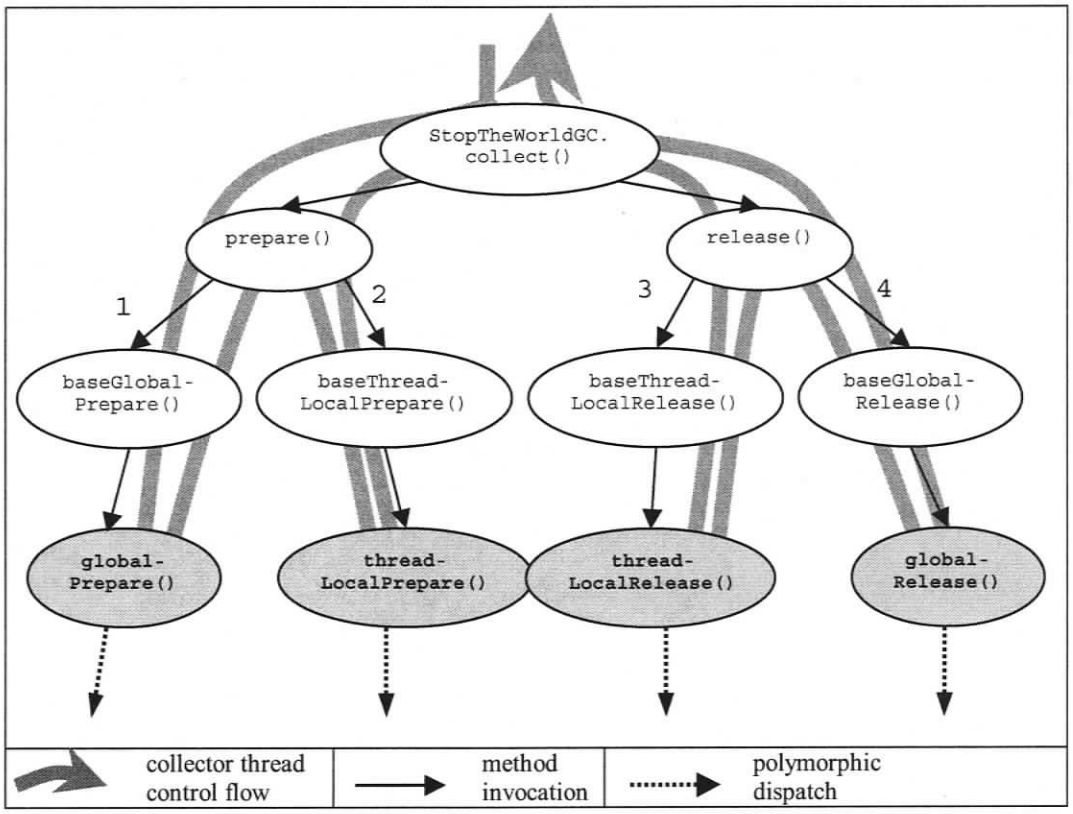


Figure 16: Control flow through *Prepare/Release* paths of garbage collection

The large grey arrow in Figure 16 indicates the control flow through each of the four paths: (1) `globalPrepare`, (2) `localPrepare`, (3) `localRelease`, and (4) `globalRelease`, where each path results in the dispatch of the overriding `Plan` methods. The system is built with a single `Plan` class, which is selected at the time of

configuration. This ordering is a critical part of GC protocol correctness. Each collector's `Plan` class that extends `StopTheWorldGC` has its own unique version of the four synchronized methods, and each invokes key methods within policy classes. Policy invocation is thus a staged, symmetric protocol dictated by the control flow through a plan. To understand the significance of the differences between how each plan is implemented, a developer must understand when policy objects are used along this path, and which policy objects are used.

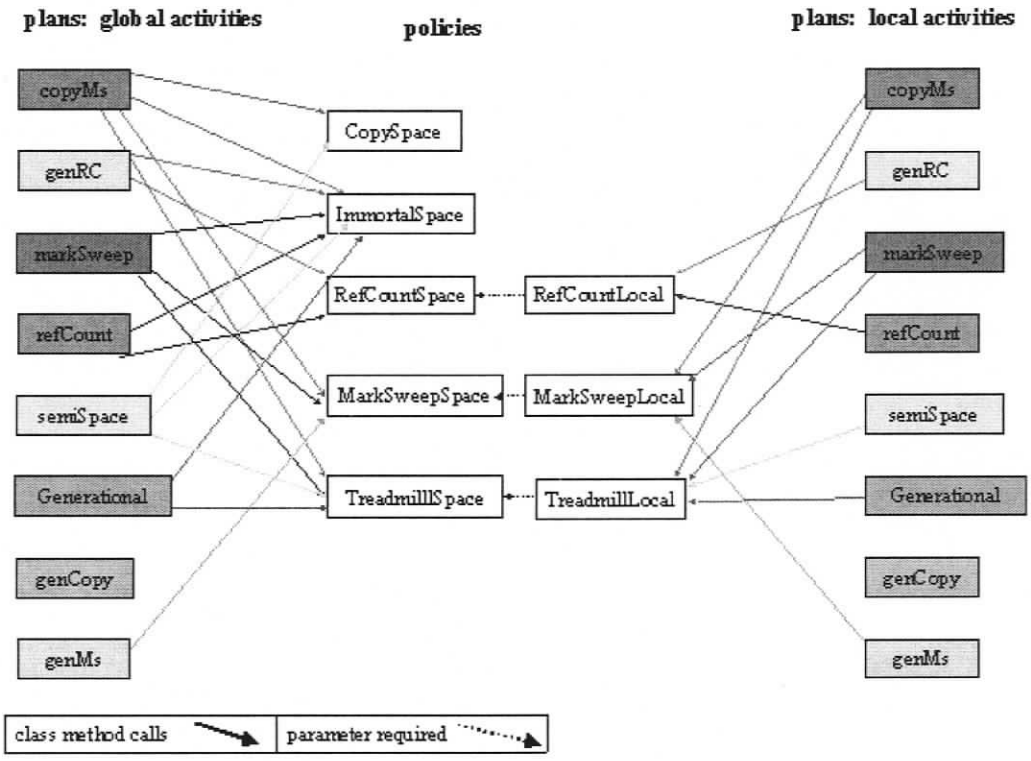


Figure 17: Global and local plan activities and the internal structure of policy

Figure 17 overviews high-level structural relationships between each *plan* and the *policies* it uses. The various plans are shown along the left and right edges of the figure respectively, their associated policies are shown in the center of the figure and the

relationship between the two is indicated with solid arrows. Additionally, the internal structure of a *policy* is represented by the separation of the policy classes into *`Space` and *`Local` groups.

The figure shows the clear division between the *global* paths (aligned along the left) and those used by the *local* paths (aligned along the right). The global paths employ only *`Space` classes, whereas the local paths employ only *`Local` classes. `Space` classes have a relationship with corresponding `Local` classes of similar names. This relationship is part of the internal structure of policy, where each of the `Local` classes take an instance of the associated `Space` class as a parameter, as indicated by the dashed arrows in Figure 17.

		Global					Local		
Plan \ Policy		Copy Space	Immortal Space	RefCount Space	MarkSweep Space	Treadmill Space	RefCount Local	MarkSweep Local	Treadmill Local
copy MS	p								
	r								
gen RC	p								
	r								
mark Sweep	p								
	r								
ref Count	p								
	r								
semi Space	p								
	r								
Gener-ational	p								
	r								
gen Copy	p								
	r								
gen MS	p								
	r								
p – prepare method call			r – release method call		Note: genCopy and genMS inherit policy from Generational				

Figure 18: `PR_Protocol` crosscuts all plans in a symmetrical manner

Given the structural properties outlined in Figure 16 and Figure 17, Figure 18 finally illustrates the symmetry of the crosscutting structure between the `prepare` and `release`

methods in the global as well as local activity. That is, each policy that is involved in `prepare` is also involved in `release`. The only exception to this rule is in the *Copy Mark and Sweep* (CopyMS) plan's use of `copySpace` policy. The `prepare` method of `copySpace` is called in the global activity yet, the `release` of `copySpace` is not. After filtering this symmetry out of the original implementation, we contacted the developers and confirmed that this exception was an oversight. It is important to note that in the current framework, this oversight has no impact on the functionality of the collector, but will be fixed in future versions of the RVM.

The motivation behind structuring the implementation of `PR_Protocol` hinges on consistent enforcement and manual verification – the ability to inspect the code and explicitly see the pattern. This domain-specific design pattern is considered fundamental in all plans, and its significance within Jikes GC plans was recently identified by Blackburn *et al.* [Blackburn, et al. 2004]. The aspects previously introduced were primarily motivated by centralized configurability, extensibility and (un)plugability whereas, the impact of this aspect is more subtle in nature.

In the original implementation, comments in the code remind the developers of the symmetry and the pattern, but there is no way to explicitly enforce it in the implementation itself – hence the original oversight with the `copyMS` plan. In contrast, as an aspect, crosscutting relationships can be explicitly structured and easier to see. Looking at an aspect-oriented implementation, developers can more easily understand the differences between plans with respect to this pattern and their use of policy. For example, Figure 19 shows the similarities and differences between three plans, `copyMS`, `genRC` and `refCount`, from this perspective.

When plan is *copyMS*	When plan is *genRC* or *refCount*
and on **globalPrepare** path *copySpace* prepare *immortalSpace* prepare *markSweepSpace* prepare *treadmillSpace* prepare and on **threadLocalPrepare** path markSweepLocalprepare(*markSweepSpace)* treadmillLocal prepare(*treadmillSpace)* and on **threadLocalRelease** path markSweepLocal release(*markSweepSpace)* treadmillLocal release(*treadmillSpace*) and on **globalRelease** path *immortalSpace* release *markSweepSpace* release *treadmillSpace* release	and on **globalPrepare** path *immortalSpace* prepare *refCount* prepare and on **threadLocalPrepare** path refCountLocal prepare(*refCountSpace*) and on **threadLocalRelease** path refCountLocal release(*refCountSpace*) and on **globalRelease** path *immortalSpace* release *refCount* release

Figure 19: Structure of *Prepare/Release* in `copyMS`, `genRC` and `refCount` plans

The corresponding `PR_Protocol` aspect for `copyMS` is shown in Figure 20. The internal structure of the aspect is a finite state machine, moving through the stages of the protocol in order. We believe that, in this form, developers can more effectively reason about the symmetry of this pattern, as it is better separated and exposed in context. To be consistent with the original implementation, the aspect fails to release `CopySpace`. However, due to the proximity of `prepare` and `release` in the aspect, we believe an oversight such as this to be far less likely to escape visual verification.

```
privileged aspect Prepare_Release_for_CopyMS {

    private final int GLOBAL_PREPARE = 0;
    private final int LOCAL_PREPARE  = 1;
    private final int LOCAL_RELEASE  = 2;
    private final int GLOBAL_RELEASE = 3;
    private int state = GLOBAL_PREPARE;

    after(Plan p):target(p)
     && ( execution(* Plan.globalPrepare(..))
        || execution(* Plan.threadLocalPrepare(..))
        || execution(* Plan.threadLocalRelease(..))
        || execution(* Plan.globalRelease(..))) {

      switch(state){

          case(GLOBAL_PREPARE):
             CopySpace.prepare();
             Plan.msSpace.prepare();
             ImmortalSpace.prepare();
             Plan.losSpace.prepare();
             state++;
             break;

          case(LOCAL_PREPARE):
             p.ms.prepare();
             p.los.prepare();
             state++;
             break;

          case(LOCAL_RELEASE):
             p.ms.release();
             p.los.release();
             state++;
             break;

          case(GLOBAL_RELEASE):
             Plan.losSpace.release();
             Plan.msSpace.release();
             ImmortalSpace.release();
             state = GLOBAL_PREPARE;
      }
    }
}
```

Figure 20: `PR_Protocol` aspect for `copyMS`

2.4 Chapter Summary

This chapter provided an overview of Phase I of the experiment for this thesis – the aspect-oriented refactoring of MMTk. This phase of the experiment demonstrated how aspects can be structured as a natural locus of control and how this new modularity

provided enhanced extensibility, centralized configurability and increased verifiability for the representative crosscutting concerns. We argued that from these results we could infer similar benefits for design invariants, analysis tools, and design patterns in general. The question remains however, as to how well these aspects will fare when faced with a rapidly evolving development environment. Chapter 3 now provides the details of Phase II, the actual evolution of $MMTk^1$ and Phase III, the application of this same set of evolutionary changes to $MMTk^1_{ao}$.

Chapter 3 - Evolution of a Memory Management Subsystem

Since its original restructuring of JMTk, the memory management subsystem of the Jikes RVM continues to undergo substantial evolution; this experiment considers a snapshot of this evolution over a ten-month period from January 2004 to October 2004. This chapter covers the evolutionary phases of this experiment, Phases II and III, dictated by these real changes captured by monitoring the code over that ten-month period. In Phase II, the key evolutionary steps made by the developers of MMTk are identified, and the change tasks they impose are established in terms of the number of modules affected. In this phase, the specific change tasks are further classified into one of three categories: (1) separation of MMTk from the Jikes RVM, (2) separation of concerns within MMTk or (3) general evolution and maintenance. Further, this phase provides a high-level overview of the relationship between the change tasks and the ultimate goal of improved portability and manageability.

The chapter then overviews these change tasks with respect to the evolution of $MMTk^{1}_{ao}$ in Phase III. This evolution is modeled after the extensive evolutionary change tasks identified in Phase II. That is, Phase III considers how the change tasks identified in Phase II would manifest themselves in the aspect-oriented version of the system, developed in Phase I. This chapter proceeds as follows. Section 3.1 provides the details of Phase II – the changes to $MMTk^{1}$ over the ten-month evolutionary period, and a high-level overview of how these change tasks play out within the RVM. Section 3.2 provides the details of Phase III – how these same changes play out in $MMTk^{1}_{ao}$.

3.1 Evolution of MMTk[1] to MMTk[2]

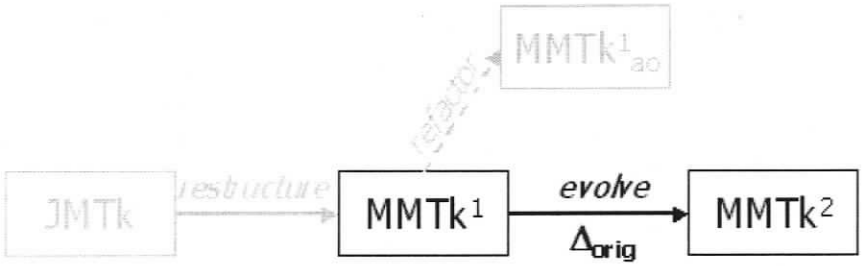


Figure 21: Phase II – Evolution of MMTk[1]

The evolution of MMTk[1] to yield MMTk[2] reveals a move towards an even more portable and manageable subsystem. Portability is necessary as MMTk can be used in other systems requiring garbage collection, such as the Glasgow Haskell Compiler[1] and Open Virtual Machine (OVM)[2]. Manageability is necessary as MMTk serves as a platform for developers to experiment with new GC strategies. As a result of these two main motivating factors, key evolutionary steps of MMTk include: (1) new adherence to a strict interface for portability, and (2) new hierarchical decomposition and migration of code within (sub)packages to better separate concerns. A high level overview of the extensive nature of this evolution over a ten-month period is provided by Table 3, with more detailed accounting of twelve major restructurings in Table 4.

[1] The Glasgow Haskell Compiler, http://www.haskell.org/ghc/, 2006.

[2] OVM, http://www.ovmj.org/, 2006.

Table 3: Evolution in MMTk: old and new class structure overview

Top Level Package	# of OLD CLASSES	Evolved Classes
GCspy	14	moved out of MMTk
Plan	22	15 (new hierarchy)
Policy	10	15 (new hierarchy)
Utility	63	14 (+5 new subpackages)
VMInterface	17	redistributed in MMTk

The twelve restructurings in Table 4 are significant to this thesis, as they form the basis of the experiment for the evolution in this study. The columns represent different categories of restructuring. On the left, changes 1.1-1.4 were necessary to make MMTk more portable, and to make its separation from the Jikes specific code cleaner. In the middle, changes 2.1-2.4 support better separation of concerns within MMTk, making it easier for developers to experiment with GC. In the final column, changes 3.1-3.4 represent more generic tasks associated with maintenance and evolution.

Table 4: Detailed evolution of twelve restructuring tasks of MMTk over ten months

BETTER SEPARATION OF MMTK FROM JIKES	BETTER SEPARATION OF CONCERNS WITHIN MMTK	GENERAL EVOLUTION AND MAINTENANCE
1.1) Eliminated MMTk and VM_Magic interaction	2.1) Restructured plan package	3.1) 1 class changed to implement synchronization interface
1.2) Eliminated utility classes	2.2) Restructured policy package	3.2) Eliminated assertion or failure methods
1.3) Relocated and renamed VM_Address class	2.3) Restructured utility package	3.3) Introduced new classes
1.4) Relocated and renamed synchronization interfaces	2.4) Redistristributed and eliminated VM_Interface	3.4) Introduced calls to assertion or failure methods

3.1.1 Inheritance and Preprocessing

The configuration of MMTk supports the selection of one of many different GC strategies or plan, where the system build is tailored to the GC plan selected. In the original implementation there is a package for each GC plan, which contained the implementation of core features in the `Plan` class. The specific `Plan` class to be included in a particular build of the system is determined by a command line option. In the evolved implementation, these subdirectories were eliminated and each GC strategy has its own distinctly named class (`CopyMS.java`, `SemiSpace.java`, etc.). The `Plan` class was migrated outside of MMTk, and made a subclass of each of these classes mutually exclusively, using preprocessor directives:

```
//-#if RVM_WITH_SEMI_SPACE
public class Plan extends SemiSpace implements Uninterruptible {
//-#elif RVM_WITH_SEMI_SPACE_GC_SPY
public class Plan extends SemiSpaceGCSpy implements Uninterruptible {
//-#elif RVM_WITH_COPY_MS
public class Plan extends CopyMS implements Uninterruptible {
…
(more plan classes)
//-#endif
```

The `Plan` class highlights another way in which plans in MMTk evolved, further leveraging hierarchical decomposition and this multiplexed approach. For example, `Semi_Space_GC_Spy` uses a different class than the regular `Semi_Space` plan. This better separates instrumentation for the dynamic analysis tool, *GCSpy*, from the regular plan. `Semi_Space_GC_Spy` and `Semi_Space` are siblings under a `Semi_Space_Base` superclass. Shared code is in the base class, functions instrumented for *GCSpy* are in the `GC_Spy` child, and uninstrumented versions of those functions are in the regular class. In the original implementation, this separation was not supported by a class hierarchy. Instead,

a combination of preprocessor directives and global flags were used in `Semi_Space`. We will return to this issue in Section 3.2.4.

3.1.2 Empty Interfaces

Another of the other interesting features of the Jikes system infrastructure is its lightweight leveraging of empty interfaces to flag concerns handled by lower level system code. For example, the majority of the classes in MMTk implement an interface called `VM_Uninterruptible` (*pre-evolution*) or `Uninterruptible` (*post-evolution*). Classes that implement this interface cannot be interrupted. The mechanism to supply the functionality for this concern, is actually provided by a lower level of the system, outside of the implementation of the Jikes core. We will revisit this issue in the conclusions in Section 5.2.

3.2 Evolution of $MMTk^1_{ao}$ to $MMTk^2_{ao}$

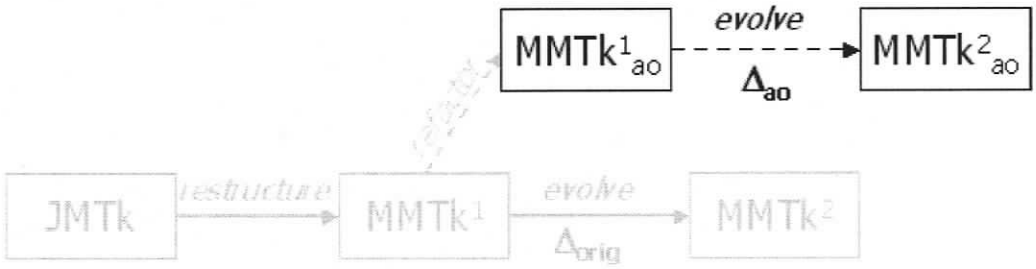


Figure 22: Phase III – Evolution of $MMTk_{ao}$

This section presents Phase III of this experiment, the evolution of the aspect-oriented version of MMTk ($MMTk_{ao}$) [Gibbs, *et al.* 2005a, Gibbs, *et al.* 2005b]. This phase of the experiment is modeled after the extensive evolutionary change tasks identified in

Phase II, where a single change can affect multiple concerns. Recall, these aspects are grouped into three representative categories: design invariants, analysis tools and design patterns. In addition to changes directly to the crosscutting concern localized by the aspect, we are interested in the changes to the other concerns that affect or interact with these representative aspects referred to as *interacting concerns*.

This section proceeds with an overview of how the evolution of the dominant decomposition of the system impacted each aspect. Section 3.2.1 provides background information on interactions between separate concerns. Sections 3.2.2 and 3.2.3 highlight the impact of the system restructuring on the two design invariant aspects. Sections 3.2.4 and 3.2.5 consider this impact on the analysis tool and the design pattern aspects respectively.

3.2.1 Concern Interaction

As Parnas observed, there is more to system behaviour than is shown by a modular decomposition. To clearly understand the behaviour of a system we must not only have a clear structural view of the core concerns, but also a view of the external interaction of those concerns. In AOSD, we can categorize concerns into two groups: *primary concerns* are the concerns that make up the primary modularity or the dominant decomposition of the system; *crosscutting concerns* (CCC) are those concerns that have a structure that cuts across the primary modularity of the system. A crosscutting concern interacts with the concerns that it crosscuts. That is, the concerns that an aspect crosscuts are *interacting concerns* (IC) with that aspect. The other concerns not touched by a

particular aspect are *non-interacting concerns* (NIC). Table 5 is a list of the crosscutting concerns in this experiment with their corresponding interacting concerns.

Table 5: Crosscutting concerns and their interacting concerns

CROSSCUTTING CONCERNS (CCC)	INTERACTING CONCERNS (IC) (in terms of packages)
DesignInvariants: VerifyAssertions Synchronization	MMTk.*
GCSpy	plan, policy, utility
PR_Protocol Protocol	plan

3.2.2 Design Invariant: Evolution of Synchronization

The aspect used for synchronization is very simple. It has a specific set of classes that do *not* need to implement a given interface that flags if the class is uninterruptible. As the majority of the classes in the subsystem cannot be interrupted, this list contains the exceptions to the rule. As previously mentioned, the interface itself is empty, and is used by a lower level concern.

During evolution, as new classes were added to the system, all of which implement the interface, the aspect deals with this correctly (Table 4, 3.3). This is due to the nature of the pointcut, which automatically applies to all classes not listed in Figure 23. Additionally, this aspect was affected by two more changes in the evolution from old to new. The interface was renamed (Table 4, 1.4), and the `BasePolicy` class changed status (Table 4, 3.1), such that it required the interface shown in a comment in Figure 23. This commented out line of code along with the renaming of the tag interface are the only

differences between the pre-evolution aspects shown in Figure 12 and the post-evolution aspect shown in Figure 23.

```
privileged aspect Synchronization {

  declare parents :
    (org.mmtk.* || com.ibm.JikesRVM.memoryMangers.mmInterface.*)
     && !( *Header
       || org.mmtk.utility.AllocAdvice
       || org.mmtk.utility.TracingConstants
       || org.mmtk.utility.CallSite
  //   || org.mmtk.policy.BasePolicy
       || org.mmtk.vm.ScanStatics
       || org.mmtk.vm.Constants
       || com.ibm.JikesRVM.memoryManagers.mmInterface.MM_Constants
       || com.ibm.JikesRVM.memoryManagers.mmInterface.SynchronizationBarrier
       || com.ibm.JikesRVM.memoryManagers.mmInterface.VM_CollectorThread
       || com.ibm.JikesRVM.memoryManagers.mmInterface.VM_GCMapIteratorGroup
       || com.ibm.JikesRVM.memoryManagers.mmInterface.VM_Handshake )

      implements Uninterruptible;
}
```

Figure 23: `Synchronization` aspect evolved

3.2.3 Design Invariant: Evolution of Assertion Verification

The Jikes RVM uses a boolean field, `VerifyAssertions`, as a global flag to enable assertion checking. The `VM` class, which originally was home for this flag, has a comment dictating the structure of this design invariant:

```
/* Note: code your assertion checks as
   "if (VM.VerifyAssertions) VM._assert(xxx);"  */
```

During evolution, the `VM_Interface` class was reorganized as several new classes, one of which was the `Assert` class, for better separation of this concern. Further refactoring eliminated a previously core method from this concern. This is shown in Figure 24, with the elimination of the `space_failure` pointcut from the pre-evolution to the post-evolution versions of the `VerifyingAssertions` aspect.

Pre-Evolution

```
privileged aspect VerifyingAssertions {

   pointcut GCstrategy(): within(org.mmtk. *);

   pointcut asserting(boolean condition):
       call(void VM_Interface._assert(boolean))
       && args(condition) && GCstrategy();

   pointcut failing(String msg):
       call(void VM_Interface.sysFail(String))
       && args(msg) && GCstrategy();

   pointcut space_failure(VM_Address obj, byte space, String source):
       call(void Plan.spaceFailure(VM_Address, byte, String))
       && args(obj, space, source) && GCstrategy();

   void around(boolean b): asserting(b){
       if (VM_Interface.VerifyAssertions)
            proceed(b);
   }
   void around(String str): failing(str){
       if (VM_Interface.VerifyAssertions)
           proceed(str);
   }
   void around(VM_Address obj, byte space, String source):
         space_failure(obj,space,source) {
       if (VM_Interface.VerifyAssertions)
           proceed(obj, space, source);
   }
}
```

Post-Evolution

```
privileged aspect VerifyingAssertions {

   pointcut GCstrategy(): within(org.mmtk.*);

   pointcut asserting(boolean condition):
       call(void Assert._assert(boolean))
       && args(condition) && GCstrategy();

   pointcut failing(String msg):
       call(void Assert.fail(String))
       && args(msg) && GCstrategy();

   void around(boolean b): asserting(b){
       if (Assert.VerifyAssertions)
            proceed(b);
   }
   void around(String str): failing(str){
       if (Assert.VerifyAssertions)
           proceed(str);
   }
}
```

Figure 24: `VerifyingAssertions` aspect: pre and post evolution

The general structure of the design invariant remains across the evolution of the system, as shown in Table 6 and Figure 24. Since the system is so performance critical, the presence/absence of this code is significant, and evidence that it has been removed for performance but restored for correctness appears in comments in the Concurrent Versions System (CVS)[3] logs.

Table 6: Assertion verification across two versions of the plan and policy packages

OCCURRENCE OF...	PRE-EVOLUTION	POST-EVOLUTION
if (verify_assertions)	98 instances, 19 classes	81 instances, 21 classes
call to _assert(..)	75 instances, 25 classes	68 instances, 23 classes
call to sysFail/fail(..)	29 instances, 9 classes	31 instances, 14 classes
call to spaceFailure(..)	14 instances, 8 classes	no longer exists

In terms of accounting for changes forced by evolution, this aspect deals with new/removed calls to `assert/failure` methods without requiring change (Table 4, 3.4), but still had a total of eight points of change in the evolution. During the separation of MMTk from Jikes, global fields used by MMTk were moved to be within its boundaries (Table 4, 1.2). Cleaning of the `Assert` class caused the removal of a method resulting in an obsolete pointcut/advice (Table 4, 2.4, and 3.2).

3.2.4 Analysis Tool: Evolution of *GCSpy*

The core implementation of *GCSpy*, a dynamic analysis tool for garbage collection, involves two parts: (1) gathering of data before and after garbage collection, and (2) connecting to a *GCSpy* server and client-GUI for heap visualization. In order to

[3] Concurrent Versions System (CVS), http://www.nongnu.org/cvs/, 2006.

instrument MMTk with this code, the configuration of the system with *GCSpy* requires that existing methods be instrumented, and new methods be added, as outlined in Table 7.

Table 7: Instrumentation of *GCSpy*

PACKAGE	CLASS	OCCURRENCES OF IF (GCSPY)	NEW METHODS
org.mmtk.plan	Plan (SemiSpace)	5	6
	BasePlan	0	5
	StopTheWorldGC	4	0
org.mmtk.utility	FreeListResource	1	0
	MonoToneVMResource	2	2
com.ibm.JikesRVM. memorymanagers mmInterface	MMInterface	2	0
	Total	14	13

In the original implementation, part of the configuration strategy for *GCSpy* involves checking a global flag, `if (VM_Interface.GCSPY)`, before invoking *GCSpy* functionality `()`. The flag is set using preprocessor directives, as follows:

```
public static final boolean GCSPY =
  //-#if RVM_WITH_GCSPY
  true;
  //-#else
  false;
  //-#endif
```

`MainThread.java`, part of the scheduler package, also uses the directive within its imports and to start the server:

```
//-#if RVM_WITH_GCSPY
import com.ibm.JikesRVM.memoryManagers.mmInterface.MM_Interface;
//-#endif
...
public void run () {
      //-#if RVM_WITH_GCSPY
      MM_Interface.startGCSpyServer();
      //-#endif
```

In `VM_BootRecord.java`, part of the system's runtime support package, 28 fields for *GCSpy* are declared when this directive is true and `VM_Syscall` contains the methods:

```
//-#if RVM_WITH_GCSPY
   // GCspy entry points
   public VM_Address gcspyDriverAddStreamIP;
   public VM_Address gcspyDriverEndOutputIP;
   …
   public VM_Address gcspySprintfIP;
//-#endif
```

`VM_Syscall.java`, has 28 syscall entry points:

```
//-#if RVM_WITH_GCSPY
public static VM_Address
gcspyDriverAddStream (VM_Address driver, int it) {
  return null; }

public static void
gcspyDriverEndOutput (VM_Address driver) {}

…
public static int
gcspySprintf (VM_Address str, VM_Address format,
              VM_Address value) { return 0; }
//-#endif
```

An assortment of several other classes, also not in MMTk, use this directive to selectively import and introduce *GCSpy* functionality. The typical format for these classes is of the form: `#if RVM_WITH_GCSPY, <include imports>` or `<define method bodies>, #else <provide empty bodies>.`

For example, in `ObjectMap.java`:

```
import com.ibm.JikesRVM.VM_SizeConstants;
import com.ibm.JikesRVM.VM_Uninterruptible;
import com.ibm.JikesRVM.VM_Address;

//-#if RVM_WITH_GCSPY
import org.mmtk.plan.Plan;
import org.mmtk.utility.Log;
…
//-#endif

/**
 * THIS CLASS IS NOT A GCSPY COMPONENT
 *
 */
```

Refactoring the portions of *GCSpy* handled by the preprocessor directives was straightforward. $MMTk_{ao}$ enjoys the added benefit of being able to plug/unplug at build time, instead of requiring the system to be first reconfigured and then rebuilt.

```
public class ObjectMap
  implements VM_SizeConstants, VM_Uninterruptible {

      //-#if RVM_WITH_GCSPY
      private static final int LOG_PAGE_SIZE = 12;
      static final int PAGE_SIZE = 1<<LOG_PAGE_SIZE;
      …
      public ObjectMap() { }

      public final void boot() {
            objectMap_ = Util.malloc(OBJECTMAP_SIZE);
            VM_Memory.zero(objectMap_, OBJECTMAP_SIZE);
            …
      }
      …
      //-#else
      public ObjectMap() {}
      public final void boot() {}
      …
      //-#endif
}
```

Within $MMTk_{ao}$, as opposed to subclassing and introducing redundant code to allow for the *GCSpy* functionality (as described in Section 3.1.1), the aspect can provide functionality to a standard version of the plan, for example, `SemiSpace`, as depicted in Figure 25. This figure reveals that the combination of global flags, preprocessor directives, and subclassing leveraged by the original implementation become unnecessary by moving the code into an aspect.

As *GCSpy* was the most thinly scattered concern considered in the study, involving many points in the execution of the system with almost a 1:1 ratio of pointcuts:advice, it was impacted by a majority of the restructuring in Table 4 (9/12 tasks, 1.1-1.3, 2.*, 3.1,2).

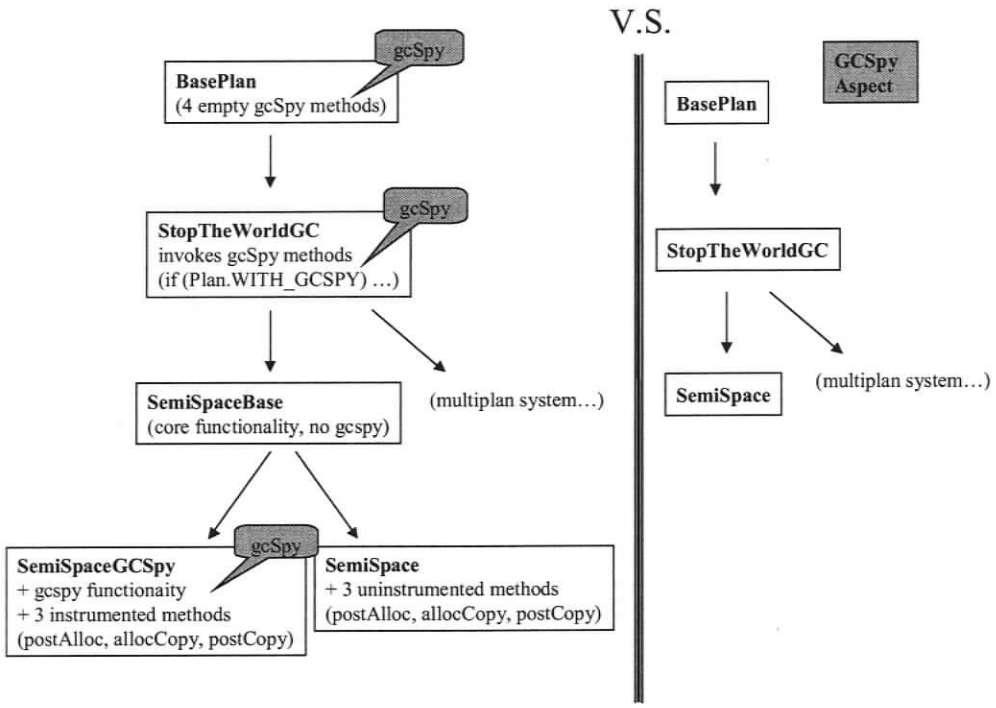


Figure 25: GCSpy in MMTk versus MMTk$_{ao}$

3.2.5 Design Pattern: Evolution of *Prepare/Release*

To understand how each plan is composed one must understand their relationships with the various policies supplied in the RVM. Plan and policy are thus two key features of the dominant decomposition of the RVM. Each policy has its own allocation and collection strategies, drawn from basic allocation and collection mechanisms. The memory management plans in the RVM are composed of different combinations of these policies. There are currently eight different memory management plans supported by the RVM.

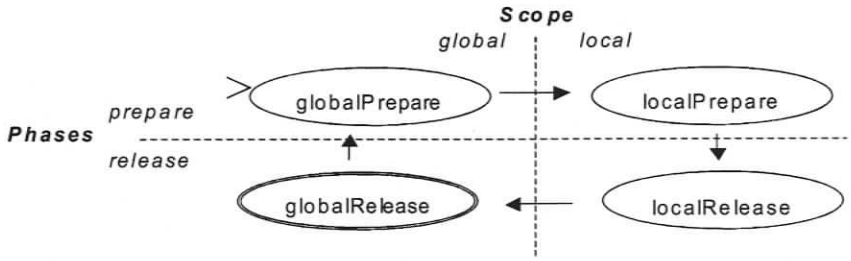


Figure 26: Finite state machine for `PR_Protocol`

GC strategies in the RVM each follow a simple algorithm of *prepare*, *process-all-work*, and *release* for collection. Since each of the GC strategies share the code base for *process-all-work*, the key differences between them are in the *prepare* and *release* phases. With a closer inspection of the original implementation a further breakdown of this design into *globalprepare*, *localprepare*, *localrelease* and *globalrelease* can be seen and represented as a simple finite state machine as illustrated in Figure 26. Each of these states is comprised of calls to the various policy mechanisms.

When the relationship between policy and plan is filtered out this way, a clear symmetry between the *prepare* and the *release* phases is uncovered. This symmetry is present in both the *local* and *global* scopes. Each of the policies involved in the *globalprepare* are in turn involved in the global release and the same is true in the case of the local scope.

Blackburn et. al. details the domain specific design patterns used in the implementation of MMTk, one being the *Prepare/Release* phases involved in garbage collection [Blackburn, et al. 2004].

```
//handles PR_Protocol of SemiSpace GC plan
privileged aspect PR_Protocol {

  private int state = 0;
  private final int GLOBAL_PREPARE     = 0;
  private final int LOCAL_PREPARE      = 1;
  private final int LOCAL_RELEASE      = 2;
  private final int GLOBAL_RELEASE     = 3;

  after(Plan p):target(p)
    && (execution(* Plan.globalPrepare(..))
    || execution(* Plan.threadLocalPrepare(..))
    || execution(* Plan.threadLocalRelease(..))
    || execution(* Plan.globalRelease(..))) {

      switch(state){

        case(GLOBAL_PREPARE):
            CopySpace.prepare();
            ImmortalSpace.prepare();
            Plan.losSpace.prepare();
            state++;
            break;

         case(LOCAL_PREPARE):
            p.los.prepare();
            state++;
            break;

         case(LOCAL_RELEASE):
            p.los.release();
            state++;
            break;

         case(GLOBAL_RELEASE):
            Plan.losSpace.release();
            CopySpace.release();
            ImmortalSpace.release();
            state = GLOBAL_PREPARE;
            break;
       }
   }
}
```

Figure 27: `PR_Protocol` aspect of `SemiSpace` plan

```
//handles PR_Protocol of MarkSweep plan
privileged aspect PR_Protocol {

  private int state = 0;
  private final int GLOBAL_PREPARE     = 0;
  private final int LOCAL_PREPARE      = 1;
  private final int LOCAL_RELEASE      = 2;
  private final int GLOBAL_RELEASE     = 3;

  after(Plan p):target(p)
    && (execution(* Plan.globalPrepare(..))
    || execution(* Plan.threadLocalPrepare(..))
    || execution(* Plan.threadLocalRelease(..))
    || execution(* Plan.globalRelease(..))) {


       switch(state){
         case(GLOBAL_PREPARE):
              Plan.msSpace.prepare();
              ImmortalSpace.prepare();
              Plan.losSpace.prepare();
              state++;
              break;

         case(LOCAL_PREPARE):
              p.ms.prepare();
              p.los.prepare();
              state++;
              break;

         case(LOCAL_RELEASE):
             p.ms.release();
             p.los.release();
             state++;
             break;

         case(GLOBAL_RELEASE):
             Plan.losSpace.release();
             Plan.msSpace.release();
             ImmortalSpace.release();
             state = GLOBAL_PREPARE;
             break;
       }
  }
}
```

Figure 28: `PR_Protocol` aspect of `MarkSweep` plan

The `PR_Protocol` aspects for `SemiSpace` and `MarkSweep` GC plans are shown in Figure 27 and Figure 28. By looking at the aspects in close proximity, the subtle differences in policy mechanisms employed by the two plans become evident. This same representation is scalable to all plans, providing developers of new plans a clear visual

representation of current implementations and an unambiguous, staged-protocol to follow in the development of new plans.

Pre and post-evolution versions of these plans differ in their pointcuts, as a result of the combination of hierarchical restructuring and preprocessor directives described with respect to plans (Table 4, 2.1). As a result, the pointcuts move from the generic use of `Plan.<method>`, to the specific use of `SemiSpace.<method>` and `MarkSweep.<method>`.

3.3 Chapter Summary

This chapter provided a summary of the evolution of the memory management subsystem in this study. The key evolutionary steps within the ten-month period spanning from January to October 2004 are identified and categorized. This extensive evolution forced changes to the majority of modules across every top-level package within this subsystem. The specific change tasks identified to support this restructuring were grouped into three categories of change. These categories vary in the granularity of an evolutionary objective. Two categories relate to improved separation of concerns – the first at the level of the RVM, and the second at the level of the memory management subsystem. The third category encompasses a finer granularity of changes related to general evolution and maintenance. Categorizing the changes in this way allowed us to reason about how varying granularities impact the different structural representations of crosscutting concerns.

The structural benefits of an aspect-oriented approach were shown Chapter 2, whereas this chapter also provided an overview of the impact of the change tasks on $MMTk_{ao}$.

Building on Chapter 2, this chapter looked at how the aspect-oriented version must be changed as a result of evolution. The evolution forced changes to all aspects in this study either directly or indirectly through interacting concerns. In this way, the RVM provides a good test environment for the sustainability of aspects over a period of extensive evolution. Chapter 4 now provides an analysis of these results, along with an overview of performance results for both the aspect and non-aspect versions.

Chapter 4 - Validation

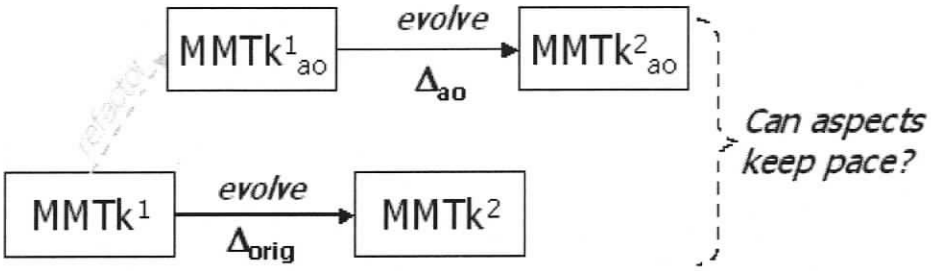


Figure 29: Phase IV – An Evolutionary Comparison, Δ_{ao} versus Δ_{orig}

This chapter provides an analysis of the results presented in the previous chapter. The focus is on Phase IV, shown in Figure 29, and the question of how well aspects hold up during extensive restructuring of the code they crosscut: That is, how does their evolution (Δ_{ao}) compare to their scattered and tangled counterparts in the non-aspect version (Δ_{orig})? This chapter provides a comparison by applying a combination of fine and coarse-grained evaluations of these implementations supplemented with a preliminary performance evaluation.

This chapter proceeds as follows. Section 4.1 begins with a fine-grained evaluation, considering the impact of the change tasks on the individual aspects in isolation followed by a generalization of these results. Section 4.2 provides a coarse-grained evaluation of the positive/negative/neutral impact each of the aspects had on the evolution of the system as a whole. Section 4.3 further considers changes that occur to more than one aspect. Section 4.4 provides an overview of performance benchmarks on MMTk and $MMTk_{ao}$ respectively.

4.1 Impact of Change

Table 8: Change Summary: required change(Δ), automatically captured change(AC)

	Aspect / Corresponding IC(s) / Evolutionary Change	Synchronization	IC	Verify Assertions	IC	GCSpy	IC	PR_Protocol	IC
Separation of MMTk	1.1) Eliminated MMTk and VM_Magic interaction					Δ	Δ		
	1.2) Eliminated utility classes					Δ	Δ		
	1.3) Relocated and renamed VM_Address class					Δ	Δ		
	1.4) Relocated and renamed synchronization interfaces	Δ	Δ			Δ	Δ		
Separation of concerns within MMTk	2.1) Restructured plan package					Δ	Δ	Δ	Δ
	2.2) Restructured policy package					AC	Δ	AC	Δ
	2.3) Restructured utility package					AC	Δ		
	2.4) Redistributed and eliminated VM_Interface class			Δ	Δ	Δ	Δ		
Evolution/ Maintenance	3.1) Class changed to implement synchronization interface	Δ	AC						
	3.2) Eliminated assertion or failure functions			Δ	Δ				
	3.3) Introduced new classes	AC	Δ			Δ	Δ	AC	Δ
	3.4) Introduced new assertion/failure calls			AC	Δ				

Table 8 summarizes the findings of this experiment, detailing precisely which change tasks impact each of the four aspects. The subsequent three sections consider these findings on a per-aspect basis. For each of the twelve restructurings identified in Chapter 3, each of the aspects was forced to change by virtue of the fact that a concern they interacted with (an interacting concern or IC) had changed. That is, while the dominant

decomposition of the system evolves, core aspect functionality stays the same. Thus, looking at the IC column associated with each aspect in Table 8, in all but one case, a change occurs. In this exceptional case however, no change occurs in the IC, but instead the aspect is changed. In this case, the status of `BasePolicy` flips from interruptible to uninterruptible (as indicated by the commented line in Figure 23). This change is already captured for the IC and is marked by the AC (*automatically captured*) in the IC column associated with the `Synchronization` aspect.

Looking at the rows associated with the restructurings, five change tasks apply to more than one aspect (1.4, 2.1, 2.2, 2.4, 3.3). In three of these, changes to ICs force changes in aspects (1.4, 2.1, 2.4), one of them is automatically absorbed by pointcuts (2.2), and the other is automatically absorbed by pointcuts in two out of three aspects involved (3.3).

4.1.1 Design Invariants: Assertion Verification and Synchronization

In both cases, the aspects that encapsulate design invariants have a positive impact in terms of providing a more precise and clear representation of the internal structure of the crosscutting concerns. Each appears to be in line with growth trends in the system. In the case of assertions, the (un)pluggable application of advice to all/no calls can be concisely and accurately represented (Figure 24). In the case of synchronization however, this requires a specific list, as the application of the advice must be selective in the classes that require synchronization instead of all-or-nothing.

In terms of negative impact, checking all `assert/failure` calls in $MMTk_{ao}$ means there is some (small) amount of redundancy relative to MMTk, where several assertions can be made consecutively in a compound statement. With synchronization however, the

negative impact stems from the current lack of structure in the code that is crosscut, resulting in an exhaustive list.

In terms of neutral impact, where MMTk and $MMTk_{ao}$ appear to tie in terms of evolvability, stems from the fact that changes to the location and existence of interacting concerns requires cosmetic updating of the aspect/objects in a similar fashion. Even with the inversion of synchronization status in `BasePolicy`, there would have to be one change made, either to the aspect or the class. We consider this a tie, with the aspect having the slight edge because the nature of the change arguably falls within the realm of the crosscutting concern and not the interacting concern. Similarly, with respect to the renaming of the `VM_Uninterruptible` interface, in $MMTk_{ao}$ this change was local to the aspect, whereas in MMTk it is assumed an IDE or *grep* could be used to search and replace uniformly throughout the code-base. The redistribution and elimination of the `VM_Interface` class during evolution also affected a design invariant, when the `Assert` class took over this functionality in the `utility` package, and one of the failure methods was eliminated. This change caused a renaming of all references to the method calls in the aspect in $MMTk_{ao}$, and throughout the code in MMTk.

Overall, we argue that, though the `Synchronization` aspect does no worse than its scattered counterpart in terms of evolution, the `VerifyingAssertions` aspect fares better due to its ability to grow/shrink correctly and precisely with the system, as new classes are added and old classes are removed.

4.1.2 Dynamic Analysis Tool: *GCSpy*

Even though the aspect sustained comparable change relative to changes made by MMTk developers in the original implementation, its internal structure and functionality held true across these changes. In terms of positive impact, it was able to eliminate some redundant code relative to the subclassed `SemiSpace_With_GCSpy` in MMTk (Section 2.1), and increase configurability by consolidating what was previously a collection of preprocessor directives coupled with global flags and subclassing.

In terms of negative impact, as a dynamic analysis tool, it is not surprising that `GCSpy` crosscuts multiple objects across multiple packages of the system. There is very little redundancy in the code captured by the `GCSpy` aspect, and thus there is a low ratio of pointcut:advice definitions. Because `GCSpy` crosses structural and hierarchical boundaries in its interaction, it is subject to change at those interaction points. Among other things, `GCSpy` interacts with policies, multiple allocators, heap management, and the main collector thread. The evolution of the system caused changes to all of these interacting concerns as well as changes to previously non-interacting concerns. The addition of new policies to the system and the addition and removal of classes dealing with memory management forced change in interaction.

The `GCSpy` aspect was also affected by the relocation/renaming/redistribution of the `VM_*` classes, but the impact of this was no worse for the aspect than for the original code. Specifically the redistribution and elimination of the `VM_Interface` class in the system evolution required changes to all references made to its fields and methods in both MMTk and $MMTk_{ao}$. $MMTk_{ao}$ had less change of this type due to the elimination of the `VerifyAssertion` and `GCSpy` field checks throughout the system. The case was the same

for the relocation and renaming of the synchronization classes. This change required seven changes to the aspect across five advice and two inter-type declarations. Those same changes would have also taken place in the corresponding classes in MMTk. In this evolution the `VM_Address` class was also refactored, renamed, and relocated forcing changes to parameters and return types of methods within `GCSpy`. These changes caused a refactoring of these functions and eliminated the use of the `VM_Magic` class. These changes again would be made in both the $MMTk_{ao}$ and MMTk.

Overall, we argue that the `GCSpy` aspect allows for improved evolution in $MMTk_{ao}$ due to the fact that it consolidates and unifies preprocessor directives/global flags/hierarchical decomposition as one configurable, locus of control for this dynamic analysis tool.

4.1.3 Design Pattern: *Prepare/Release*

The positive impact of the `PR_Protocol` aspect is the clarification of the design pattern. This clarification holds throughout the evolution. The negative impact however, involves changes that have to be made to the aspect as a result of changes to the interacting concerns. The restructuring of the plan package to facilitate the move to unique naming of classes from the developer's perspective had a negative impact on this aspect. Instead of being able to consolidate this combination of compiler directives and hierarchical decomposition, this aspect suffered from it. In MMTk these changes are limited to the package itself and require no other changes in the system. $MMTk_{ao}$ leveraged the generic `Plan.java` naming convention in its design. As a result of this change, the aspect for a given plan requires six occurrences of renaming change across four pointcuts and one advice. In the event that the `PR_Protocol` aspect is scaled across

all plans, each of the plans would require the corresponding renaming done to their pointcuts and advice.

4.2 Generalization

Table 9: Impact of aspects on system evolution

ASPECT	POSITIVE	NEGATIVE	NEUTRAL
Verify Assertions (BETTER)	internal structure is clear plugability is useful all or nothing – all method calls captured by invariant	some redundant field checking relative to MMTk	change of method used in a pointcut
Synchronization (no worse)	internal structure is clear and identifies a trend in design invariant, but only to a subset of classes (not all or nothing) localized change of and access to interface	must explicitly specify classes not to be captured in aspect (no dominant pattern can be leveraged)	change in class-concern interaction automatically captures new classes
GCSpy (BETTER)	eliminates redundancy increased configurability	diverse interaction with interacting concerns	evolution of some interacting concerns
PR_Protocol (no worse)	highlights domain specific design pattern clarifies relationship of CCCs and its ICs with finite state machine	one aspect per plan evolution of MMTk yielded a similar result	evolution of interacting concerns

Table 9 supplies a summary of the positive/negative/neutral effects of these aspects on evolution. The analysis presented here argues that the presence of aspects did not introduce a penalty in terms of the course-grained change tasks required, and that two out

of four aspects provide evidence of better sustainability in that their structure actually facilitated evolution, and further evolutionary trends.

Looking at the results from the view given in Table 9, we can begin to generalize some of our findings and shed some light on what underlying characteristics might predispose certain kinds of aspects to positive/negative/neutral impact. The positive impact of each aspect in this study results from the accepted benefits associated with the localization as applied to crosscutting concerns. The more interesting results are summarized in the final two columns detailing the negative/neutral effects.

In the design invariant aspects, the interaction points are numerous, but the behavior at those points is uniform and generalized. In this type of aspect, for example, `Synchronization`, the negative impact stems from the weak representation of the invariant, inviting scenarios that may unintentionally be encompassed by the aspect. Again, with a more aggressive refactoring this abstraction may be improved and the problem may be alleviated. It is not unreasonable to assume that with proper tool support, the visibility of aspect interaction can be easily traced. Much like current aspect tool support in other environments where aspect interaction is visible in the way of gutter annotations and quick navigation between interaction points is supported.

In the `GCSpy` and `PR_Protocol` aspects, the diversity of interaction with the interacting concerns fuels the negative impact in terms of changes that ripple from interacting concerns to aspects. The `GCSpy` aspect's diversity is the result of the varied responsibilities at each of its many interaction points forcing a 1:1 ratio of pointcut to advice. The `PR_Protocol` aspect's diversity stems from the leveraging of the generic naming convention, which encompasses all plan types, but introducing a 1:1 ratio of

aspect to GC plan. As a result of this diversity, the negative/neutral impact encountered is the number of changes that must be made to the aspect when these interacting concerns evolve.

Yet another way of considering the resulting changes to $MMTk_{ao}$ is presented in the following subsection. This perspective precisely reveals what we believe to be one of the most important challenges remaining in terms of wide-scale adoption of AOSD in system infrastructure software. That is, fear of unanticipated interactions.

4.3 Fear of the Unknown: New Interactions, Multiple Aspects

Table 10: Changes by categories per aspect

IC_{old} ->IC_{new} Change in interaction of CCC with IC_{old}		**IC -> NIC Elimination of interaction of CCC with IC**		**NIC -> IC_{new} Introduction of interaction of CCC with Previously NIC**	
CCC	Δ	CCC	Δ	CCC	Δ
GCSpy	1.1	GCSpy	1.2	PR_Protocol	2.1
	1.2		1.3	GCSpy	2.1
	1.4	VerifyAssertions	3.2		2.4
	2.1			VerifyAssertions	2.4
	3.3			Synchronization	1.4
					3.1

No one can predict how a system will ultimately evolve. To get a slightly different perspective on the data in Table 8, we further categorized the changes into three groups: modification of interaction, elimination of interaction (where an interacting concern becomes a non-interacting concern, or NIC), and new interaction, as shown in the three columns in Table 10. Restructurings not listed in the table do not require changes to aspects.

Table 10 demonstrates that over the evolution considered here, one aspect requires changes to existing interactions, two aspects eliminate interactions, and all four aspects deal with new interactions. Furthermore, of these new interactions, two out of four of them impact more than one aspect (2.1, 2.4).

Arguably, these new interactions that require corresponding changes to multiple aspects potentially pose the biggest threat to developers apprehensive of AOSD. The importance of tool support for visual representation of these interactions cannot be underestimated here, but is beyond the scope of this thesis.

4.4 Performance

The results of running tests from the DaCapo Benchmark Suite[4] version beta050224 on the Jikes RVM v2.3.3, Linux 2.6.8-1.521smp, gcc-3.3.3-7, AspectJ v1.2, using an AMP Dual Athlon MP 2400+ machine with 1024 MB memory are shown in Table 11.

Table 11: DaCapo Benchmark performance results for MMTk and $MMTk_{ao}$

BENCHMARK	MMTK	$MMTk_{ao}$
Antlr	40708 ms	40937 ms (+0.56%)
Bloat	39752 ms	39550 ms (-0.51%)
Fop	14720 ms	14744 ms (0.16 %)
Jython	94148 ms	92297 ms (-1.97 %)
Pmd	42516 ms	43087 ms (1.34%)
Ps	87286 ms	86806 ms (-0.55%)

In these tests, the aspects included in $MMTk_{ao}$ are those that constitute its core functionality – `PR_Protocol` and `Synchronization`. Though the results show some

[4] DaCapo Project, http://www-ali.cs.umass.edu/DaCapo/, 2006.

noise (the average of three runs are reported), there is no discernable performance penalty for these aspects.

In order to stress-test a large, fine-grained aspect, we did a separate analysis of the `VerifyingAssertions` aspect. In its current incarnation, these tests hit the code within the aspect with 10s-100s of million invocations of assertion code, as reported in Table 12.

Table 12: Invocation analysis of the `VerifyingAssertions` aspect

BENCHMARK	INVOCATIONS OF ASSERTION CODE
Antlr	74,781,288
Bloat	86,211,537
Fop	35,325,811
Jython	217,938,922
Pmd	99,554,310
Ps	100,534,321

Results from the fast path (assertions turned off) and the slow path (assertions turned on) introduces in the range of a roughly 10% penalty as shown in Table 13. In a future refactoring of the system we plan to move all assertion checking into a composition of aspects, so that when assertions are not required, the overhead would be removed.

Table 13: DaCapo Benchmark results with assertions off: MMTk vs $MMTk_{ao}$

	ASSERTIONS OFF		
	MMTk (ms)	$MMTk_{ao}$ (ms)	Increase (%)
Antlr	37114	39824	7.3
Bloat	34346	37779	10
Fop	12794	14232	11.24
Jython	81689	88987	8.93
Pmd	38419	41460	7.92
Ps	78867	81020	2.73

4.5 Chapter Summary

This chapter provided validation for the thesis of this work, identified as the final phase, Phase IV of this experiment. An aspect-oriented implementation of four crosscutting concerns within the base system was first achieved in Phase I. Identification and classification of the change tasks involved in the evolution of this memory management subsystem was achieved in Phase II. In Phase III, the change tasks identified in Phase II were applied to the aspect-oriented version of the system, setting the stage for the analysis in Phase IV. This chapter provided results from the comparison of the aspect-oriented version of MMTk to the strictly object-oriented version from both structural and evolutionary perspectives. Results showed that, relative to the original implementation, the aspect-oriented version on the whole fared better. The combination of fine and coarse-grained evaluation and the identification of criteria for assessment provided leverage to generalize the evaluation of these aspects.

The results showed that this system restructuring impacted all of the aspects in this experiment. This, coupled with the fact that these changes *simultaneously* impact multiple aspects, confirmed that this indeed is was intensive restructuring of the current infrastructure. Given that we believe this evolution scenario to be representative, if these kinds of changes cannot be effectively managed, the scalability of collections of aspects in this infrastructure is still an open question, and fertile ground for future work. The final chapter of this thesis considers this and other areas for future work, in addition a complete summary of this experiment and an analysis and a discussion of its limitations.

Chapter 5 - Conclusion

This final chapter summarizes the ways in which these results support the thesis of this work, that the structural support provided by aspects is sustainable when system infrastructure undergoes major evolutionary change. The chapter concludes with a discussion of limitations of this experiment and future work.

5.1 Experimental Results

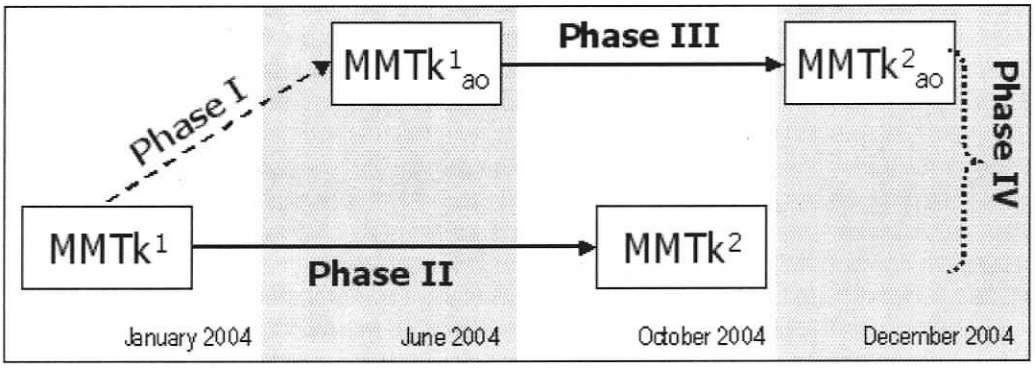


Figure 30: Experiment overview revisited

This thesis presents the results of a four phase experiment that compared the evolution of the original implementation of a memory management subsystem with an aspect-oriented version (Figure 30, revisits Figure 5 here for convenience).

The aspects include representatives from three different categories: design invariants, dynamic analysis tools [Sun 2004], and domain specific design patterns [Blackburn, et al. 2004]. The original RVM relies on preprocessor directives along with hierarchical decomposition to achieve a highly efficient, flexible system. In order to determine if

aspects could keep pace with evolution in a system based on this composition, we considered evolutionary restructuring tasks over an intensive change period of ten-months (version 2.3.3 from January – October 2004). In total, twelve significant restructurings across four aspects were considered in the study, where a single restructuring often had impact on multiple aspects. The aspects were refactored using AspectJ.

With respect to the impact of the twelve restructuring tasks, the results show that evolution of aspects fared no worse than the original implementation in two of four aspects, and better in two of four aspects. Large aspects did no worse than small aspects in terms of evolvability, and results from the DaCapo Benchmark show the system sustained a worst-case performance penalty in the range of 10%. This most extreme performance penalty was incurred by a fine-grained aspect that touches many points within the system with uniform interaction at each of those points. These results demonstrate that existing mechanisms for AOSD can support sustainable system infrastructure relative to their preprocessed/hierarchical counterparts in the somewhat inhospitable domain of system infrastructure software. Further, the resulting system structure is shown to be better suited for trends in evolution.

These results validate the thesis of this work, that structural support provided by aspects is sustainable when the system infrastructure and undergoes extensive evolution. Given that all aspects here improved structural properties, that they did no harm in the context of evolution, and half of them did better than the original implementation, we believe this provides evidence that aspects can indeed keep pace, and are a key ingredient to sustainability in system infrastructure.

5.2 Limitations

Constructing $MMTk_{ao}$ allowed us to ask *what-if* with respect to a diverse set of aspects faced with big bang evolution. This work did not include the cost of the initial refactoring to $MMTk_{ao}$, nor the impact of tool support during the process of evolution. The case study is also limited by the fact that a single developer performed all of the changes, and evolution of $MMTk_{ao}$ was dictated solely by the actual evolution of MMTk. It is reasonable to assume that given the original structure of $MMTk_{ao}$, evolution might have played out differently than it did from MMTk. The aspects themselves were identified by considering details of the design decisions made, and by manual inspection of the control flow and the dominant decomposition of the system.

5.3 Future Work

Future work from this thesis follows three possible paths. The first most natural path is a continuation of the experiment of aspects within MMTk, investigating further how aspects fare under continued evolution and new development. Further investigation into the relationship between the aspect characteristics and the cost of change is possible. As well, the redesign of the current infrastructure specifically with aspects could provide a parallel study for comparison. The second possibility branches away from MMTk, leveraging the portability of MMTk to consider the aspects of $MMTk_{ao}$ in other systems and environments. Further we can consider how different memory management systems could benefit from the structural support aspects provide. The third path branches away from memory management and considers the use of AOSD to provide structure in other

system domains. Each of these paths is considered in more detail in Sections 5.3.1 through 5.3.3.

5.3.1 Future Work within the Jikes RVM

Future work specifically within MMTk could consider: future evolutions of MMTk; further refactorings; enhanced characterizations of interactions; and the application of dynamic aspects.

The most directly related future work is the exposure of $MMTk_{ao}$ to further evolutionary changes. The structural changes by MMTk developers continued after this ten-month observation period and would provide more results further demonstrating the impact of change. This future work would include rolling this implementation forward as new versions of MMTk are released, and employing mining tools to identify other aspects. Given that the core purpose of MMTk is to provide a platform for the development of new GC strategies, a possible comparison between the development of a new strategy in $MMTk_{ao}$ verses MMTk would also be of interest.

In terms of future refactorings, this study provides only a coarse-grained, largely qualitative assessment of how these types of aspects can be expected to fare during large-scale change to the system they crosscut. This initial refactoring was intentionally done in such a way as to be least invasive to the original system as possible. Based on our experience, we believe that to truly leverage the power of AOSD in these examples, a more aggressive refactoring is required. That is, a modular design that would specifically consider crosscutting concerns as part of the criterion for decomposing a program into modules. For example, with respect to the semantics encompassed by the

`Synchronization` aspect, a stronger naming convention in the interacting concerns could influence the design of this aspect. Naming conventions are currently used within this system to impart design understanding to developers and could easily be used in this case to clarify which classes are in fact uninterruptible. This would facilitate the creation of more property based aspects that would provide a greater understanding of exactly what kind of classes fall into this synchronization family.

A further consideration for future work involves clear characterizations of interactions. The fact that new concerns simultaneously impact multiple aspects confirms that the management issue of compositions of aspects requires a solution before the question of scalability can be more completely resolved. Further investigation of how aspect compositions [Lopez-Herrejon, et al. 2006, Tanter 2006] could bring out an underlying structure in fine-grained crosscutting concerns is required to answer these questions. Additionally, although the four examples in this case study are limited, their diversity provides a basis for the categorization of aspect types and how each will hold up under system change. Specifically looking at the example of the dynamic analysis tool *GCSpy*, characterizations of the underlying nature of these types of tools, having a relatively large number of interaction points within a system that are necessarily scattered across many modules begin to surface. It provides a general view of how an aspect with many interaction points will react when any or all these points are changed. Future work includes a more detailed analysis of meaningful characterizations.

The Jikes RVM also provides a platform for the research in the area of *dynamic aspects* [Bockisch, et al. 2004]. Dynamic aspects allow the execution path, in terms of joinpoints, to be determined at runtime, allowing for runtime configuration of

crosscutting concerns. Dynamic aspects could allow developers to couple structural relationships with run-time configuration options. This combination would be particularly powerful in the context of garbage collection research. The reasons are twofold. First, additional support for crosscutting concerns makes the internal structure of policy more clear, and makes the interaction between plan and policy explicit. Second, the ability to dynamically configure garbage collection allows for more responsive experimentation, and enables more effective attempts at feedback-based, system-wide optimization [Stampflee, et al. 2004, Liu, et al. 2005, Liu, et al. 2004].

5.3.2 Future Work with Memory Management

MMTk has been reportedly ported to other virtual machine platforms including the OVM. Future work would include observing both the impact of the port on the aspects and the impact of aspects on the port. This would provide multiple platforms of integration for the same aspects, providing a venue for direct comparison between different environment and their impact on similar aspects. Specifically, the OVM provides support for real-time applications, therefore leading us to the integration of aspects into a real-time environment. Increasing demands for real-time systems are vastly outstripping the ability for developers to robustly design, implement, compose, integrate, validate, and enforce real-time constraints. It is essential that the production of real-time systems take advantage of approaches that enable higher software productivity.

The Real-Time Specification for Java (RTSJ) introduces abstractions through which developers must manage resources, such as non-garbage collected regions of memory [Bollella, et al. 2000]. The inability to manage the inherent complexity associated with

these concerns ultimately compromises the development, maintenance and evolution of safety critical code bases and increases the likelihood of fatal memory access errors at runtime. Though many accidental complexities have proven to be time consuming and problematic – type errors, memory management, and steep learning curves – a disciplined approach using a combination of tools, patterns, and aspects could help.

Scoped types are one of the latest developments in the general area of type systems for controlled sharing of references [Zhao, et al. 2004]. The key insight of the scoped type work was the necessity to make the scope structure of the program explicit in order to have a tractable verification procedure. The *JavaCop* "pluggable types" checker [Andreae, *et al.* 2006b] verifies scoped types as a pluggable type. The key idea is that pluggable types layer a new static system over an existing language, and *JavaCop* can then check syntax-directed rules at compile time. *JavaCop* makes it possible to leverage package structure in order to construct and check a relatively high-level description of the scoped type definitions. We believe coupling these with aspects to be a powerful idea.

The design of memory management in a real-time system may be clear; but its implementation typically is not because it is inherently tangled throughout the code. For this reason we propose an aspect-oriented approach for modularizing scope management within real-time systems using both scoped types and *JavaCop*. Collaborative work on this model has been introduced in the Scoped Types and Aspects for Real-Time System (STARS) project [Gibbs and Coady 2006, Andreae, *et al.* 2006a]. Future work in answering the question as to how to best strike a balance between tools, patterns and aspects remains.

5.3.3 Future Work with Structure in Other Systems Domains

This section considers future work in accessing the impact of AOSD on structure in other systems domains. Two possibilities presented here involve software transactional memory (STM) and configurable network protocol (CP) frameworks.

Much like the complexities associated with manual memory management, the implicit abstractions provided to support concurrent programming are intricate and fine-grained, resulting in a problematic, error prone programming environment. The use of STM [Herlihy and Moss 1993] for concurrency control demonstrates improved separation of high-level and low-level support for concurrent access, providing a programming environment that better shields developers from the error-prone intricacies of concurrent programming. STM implementations leverage high-level abstractions provided by software transactions, making concurrency control explicit [Harris and Fraser 2003]. This design alleviates pressure on developers, but ultimately the scattered nature of STM is still present. Additionally, developer's are unable to fine-tune concurrency control hidden in STM mechanisms. This approach ultimately sacrifices concurrency control in order to reduce complexity.

Concurrency control inherently crosscuts an application, and we believe it is this characteristic that contributes to the complexity associated with concurrent programming. Classic benefits of AOSD such as improved separation of concerns and a centralized locus of control, could provide a medium for developers to more effectively manage the underlying mechanisms and power of STM. Future work investigates the use of AOSD to put the control back into concurrency control with STM.

Another area in the systems domain that could benefit from AOSD is CP frameworks. These frameworks must provide software that is highly reusable, configurable, extensible and evolvable. But the extent to which non-functional requirements can be generically supported in such frameworks is a matter of some debate. Recently, the developers of the Cactus CP suggested that their micro-protocols could have benefited from AOSD mechanisms [Hiltunen, et al. 2006].

System infrastructure software in general is complex and intricate making evolvability, adaptability and configurability complicated. AOSD has been shown in multiple cases, discussed in related work, to support these key attributes but with the complexities associated with system infrastructure comes additional considerations [Siadat, et al. 2006]. These considerations include: (a) the issues associated with fine-grained semantic pointcuts, (b) lack of language integration with typical systems languages (i.e. C), and (c) performance overhead and increased memory consumption. Hence, an experiment that could be used to validate the efficacy of aspects in CP frameworks must explicitly establish the impact of AOSD on these three key issues in a domain specific context. This is an important experiment because by investigating related mechanisms for improved structure such as Cactus, the AOSD community may be able to help push the envelope on effective mechanisms to support truly configurable systems.

5.4 Conclusions

Separation of concerns is a core principle of software engineering, decomposing problems and programs into modules to provide structural support that can facilitate evolution. This thesis showed that AOSD can provide structural support for separation of

crosscutting concerns in systems. This support was further shown to persist under large-scale evolutionary change, demonstrating that existing mechanisms for AOSD can support sustainable system infrastructure relative to preprocessed/hierarchical alternatives.

Bibliography

[Alpern, *et al.* 2005] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar and M. Trapp, "The Jikes Research Virtual Machine project: Building an open-source research community", *IBM Systems Journal*, vol. 44, no. 2, pp. 399-417, 2005.

[Andreae, *et al.* 2006a] C. Andreae, Y. Coady, C. Gibbs, J. Noble, J. Vitek and T. Zhao, "STARS: Scoped Types and Aspects for Real-Time Systems", *European Conference on Object-Oriented Programming (ECOOP)*, Nantes, France, 2006a, to appear.

[Andreae, *et al.* 2006b] C. Andreae, J. Noble and T. Millstein, "A Framework for Implementing Pluggable Type Systems", *ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Portland, USA, 2006b, to appear.

[Baniassad, *et al.* 2002] E. Baniassad, G. Murphy, C. Schwanninger and M. Kircher, "Managing crosscutting concerns during software evolution tasks: an inquisitive study", *ACM International Conference on Aspect-Oriented Software Development (AOSD)*, Enschede, Netherlands, 2002, pp. 120-126.

[Blackburn, *et al.* 2004] S. Blackburn, P. Chung and K. McKinley, "Oil and Water? High Performance Garbage Collection in Java with MMTk", *International Conference on Software Engineering (ICSE)*, Edinburgh, UK, 2004, pp. 137-146.

[Bockisch, *et al.* 2004] C. Bockisch, M. Haupt, M. Mezini and K. Ostermann, "Virtual Machine Support for Dynamic Join Points", *ACM International Conference on Aspect-Oriented Software Development (AOSD)*, Lancaster, UK, 2004, pp. 83-92.

[Bollella, *et al.* 2000] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr and M. Turnbull, *The Real-Time Specification for Java*, Addison-Wesley, 2000.

[Coady and Kiczales. 2003] Y. Coady and G. Kiczales., "A retroactive study of aspect evolution in operating system code", *ACM International Conference on Aspect-Oriented Software Development (AOSD)*, Boston, USA, 2003, pp. 50-59.

[Colyer and Clement 2004] A. Colyer and A. Clement, "Large-scale AOSD for Middleware", *ACM International Conference on Aspect-Oriented Software Development (AOSD)*, Lancaster, UK, 2004, pp. 56-65.

[Dijkstra 1976] E. W. Dijkstra, *A Discipline of Programming, Englewood Cliffs*, United States: Prentice Hall, 1976.

[Duzan, *et al.* 2004] G. Duzan, J. Loyall and R. Schantz, "Building Adaptive Distributed Applications with Middleware and Aspects", *ACM International Conference on Aspect-Oriented Software Development (AOSD)*, Lancaster, UK, 2004, pp. 66-73.

[Engel and Freisleben 2005] M. Engel and B. Freisleben, "Supporting Autonomic Computing Functionality via Dynamic Operating System Kernel Aspects", *ACM International Conference on Aspect-Oriented Software Development (AOSD)*, Chicago, USA, 2005, pp. 51-62.

[Fiuczynski, *et al.* 2005] M. E. Fiuczynski, R. Grimm, Y. Coady and D. Walker, "patch(1) Considered Harmful", *USENIX Tenth Annual Workshop on Hot Topics on Operating Systems (HotOS)*, Santa Fe, USA, 2005, pp. 91-96.

[Gamma, *et al.* 1995] G. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Gibbs and Coady 2005] C. Gibbs and Y. Coady, "Aspects of Memory Management", *IEEE Hawaiian International Conference On System Sciences (HICSS)*, Hawaii, USA, 2005, pp. 275b.

[Gibbs and Coady 2004a] C. Gibbs and Y. Coady, "Garbage Collection in Jikes: Could Dynamic Aspects add Value?" *Dynamic Aspects Workshop held at AOSD '04*, Lancaster, UK, 2004a,

[Gibbs and Coady 2006] C. Gibbs and Y. Coady, "It is Time to Get Real with Real-Time: How Can Aspects, Patterns and Tools Help?" *Workshop on Aspects, Components and Patterns for Infrastructure Software held at AOSD '05*, Bonne, Germany, 2006,

[Gibbs and Coady 2004b] C. Gibbs and Y. Coady, "OASIS: Organic Aspects for System Infrastructure Software", *Workshop on Reflection AOP and Metadata for Software Evolution held at ECOOP '04*, Darmstadt, Germany, 2004b, pp. 42-52.

[Gibbs and Coady 2004c] C. Gibbs and Y. Coady, "SAUCI: System Aspects for Uniformity in Software Infrastructure", *International Conference on Software & Systems Engineering and their Applications (ICSSEA)*, Paris, France, 2004c,

[Gibbs, *et al.* 2005a] C. Gibbs, R. Liu and Y. Coady, "And the Band Played On: Are Aspects Adrift in a Sea of Sinking Code?" *Workshop on Linking Aspect Technology and Evolution held at AOSD '05*, Chicago, USA, 2005a,

[Gibbs, *et al.* 2005b] C. Gibbs, Y.Coady and R. Liu, "Sustainable System Infrastructure and Big Bang Evolution: Can Aspects Keep Pace?" *European Conference on Object-Oriented Programming (ECOOP)*, Glasgow, Scotland, 2005b, pp. 241-261.

[Goetz 2003] B. Goetz, "How does garbage collection work?" www-106.ibm.com/developerworks/java/library/j-jtp10283/, 2003.

[Hannemann 2003] J. Hannemann, "The Aspect Mining Tool (AMT)", http://www.cs.ubc.ca/~jan/amt/, 2003.

[Hannemann and Kiczales 2002] J. Hannemann and G. Kiczales, "Design pattern implementations in Java and AspectJ", *ACM International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, Seattle, USA, 2002, pp. 161-173.

[Harris and Fraser 2003] T. L. Harris and K. Fraser, "Language Support for Lightweight Transactions", *ACM International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, Anaheim, USA, 2003, pp. 388-402.

[Henderson-Sellers 1996] B. Henderson-Sellers, *Object-oriented metrics — measures of complexity*, Prentice-Hall, New Jersey, USA, 1996.

[Herlihy and Moss 1993] M. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural support for lock-free data structures", *International Symposium on Computer Architecture*, San Diego, USA, 1993, pp. 289-300.

[Hiltunen, *et al.* 2006] M. Hiltunen, F. Taiani and R. Schlichting, "Reflections on Aspects and Configurable Protocols", *ACM International Conference on Aspect-Oriented Software Development (AOSD)*, Bonne, Germany, 2006, pp. 87-98.

[IBM 2006] IBM, "The AspectJ Programming Guide", eclipse.org/aspectj/doc/released/progguide/index.html, 2006.

[IBM 2004a] IBM, "Jikes Research Virtual Machine", www-124.ibm.com/developerworks/oss/jikesrvm/, 2004a.

[IBM 2004b] IBM, "Jikes Research Virtual Machine User's Guide", www-124.ibm.com/developerworks/oss/jikesrvm/user-guide/HTML/userguide.html, 2004b.

[Kersten and Murphy 1999] M. Kersten and G. Murphy, "Atlas: A case study", *ACM International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, Denver, USA, 1999, pp. 340-352.

[Kiczales, *et al.* 2001] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. G. Griswold, "An overview of AspectJ", *European Conference on Object-Oriented Programming (ECOOP)*, Budapest, Hungary, 2001, pp. 303-326.

[Kiczales, *et al.* 1997] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier and J. Irwin, "Aspect-Oriented Programming", *European Conference on Object-Oriented Programming (ECOOP)*, Jyväskylä, Finland, 1997, pp. 220-242.

[Lehman and Belady 1985] L. L. Lehman and L. A. Belady, "Program Evolution", *APIC Studies in Data Processing*, vol. 3, 1985.

[Liu, *et al.* 2004] R. Liu, C. Gibbs and Y. Coady, "MADAPT: Managed Aspects for Dynamic Adaptation based on Profiling Techniques", *Third Workshop on Reflective and Adaptive Middleware held at ECOOP '05*, Glasgow, Scotland, 2004, pp. 83-94.

[Liu, *et al.* 2005] R. Liu, C. Gibbs and Y. Coady, "SONAR: System Optimization and Navigation with Aspects at Runtime", *Dynamic Aspects Workshop held at AOSD '05*, Chicago, USA, 2005,

[Lopez-Herrejon, *et al.* 2006] R. Lopez-Herrejon, D. Batory and C. Lengauer, "A disciplined approach to aspect composition", *Symposium/Workshop on Partial Evaluation and Semantic-Based Program Manipulation (PEPM)*, Charleston, USA, 2006, pp. 68-77.

[McCabe 1976] T. J. McCabe, "A complexity measure", *IEEE Transactions on Software Engineering (TSE)*, vol. 2, no. 4, pp., 1976.

[Murphy, *et al.* 2001] G. Murphy, A. Lai, R. Walker and M. Robillard., "Separating features in source code: An Exploratory Study", *IEEE International Conference on Software Engineering (ICSE)*, Toronto, Canada, 2001, pp. 275-284.

[Murphy 1996] G. C. Murphy, "Lightweight Structural Summarization as an Aid to Software Evolution", *Computer Science*, University of Washington, 1996,

[Ossher and Tarr 1999] H. Ossher and P. Tarr, "Multi-dimensional separation of concerns using hyperspaces", *IBM Research Report*, vol. 21452, 1999.

[Parnas 1972] D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules", *Communications of the ACM*, vol. 15, no. 12, pp., 1972.

[Parnas and Clements 1990] D. L. Parnas and P. C. Clements, *A rational design process: How and why to fake it, Software State-of-the-Art: Selected Papers, in T. DeMarco and T. Lister, eds.*, Dorset House Publishing, 1990.

[Rashid and Leidenfrost 2004] A. Rashid and N. A. Leidenfrost, "Supporting Flexible Object Database Evolution with Aspects", *Generative Programming and Component Engineering (GPCE)*, Vancouver, Canada, 2004, pp. 75-94.

[Sabbah 2004] D. Sabbah, "Aspects - from Promise to Reality, Keynote", *ACM International Conference on Aspect-Oriented Software Development (AOSD)*, Lancaster, UK, 2004, pp. 1-2.

[Siadat, *et al.* 2006] J. Siadat, R. Walker and C. Kiddle, "Optimization Aspects in Network Simulation", *ACM International Conference on Aspect-Oriented Software Development (AOSD)*, Bonne, Germany, 2006, pp. 122-133.

[SPEC 2006] SPEC, "Standard Performance Evaluation Corporation (SPEC)", *SPEC JVM and SPEC JBB*, http://www.spec.org, 2006.

[Stampflee, *et al.* 2004] O. Stampflee, C. Gibbs and Y. Coady, "RADAR: Really low-level Aspects for Dynamic Analysis and Reasoning", *Workshop on Programming Languages and Operating Systems held at ECOOP '04*, Oslo, Norway, 2004,

[Stevens, *et al.* 1974] W. P. Stevens, G. J. Meyers and L. L. Constantine, "Structured Design", *IBM Systems Journal*, vol. 13, 1974.

[Sun 2004] Sun, "GCspy: A Generic Heap Visualisation Framework", research.sun.com/projects/GCSpy, 2004.

[Tanter 2006] É. Tanter, "Aspects of Composition in the Reflex AOP Kernel", *International Symposium on Software Composition*, 2006, pp. 98-114.

[Tarr and Ossher 2000] P. Tarr and H. Ossher, "Hyper/J User and Installation Manual", www.research.ibm.com/hyperspace, 2000.

[Tesanovic, *et al.* 2005] A. Tesanovic, M. Amirijoo, M. Björk and J. Hansson, "Empowering Configurable QoS Management in Real-Time Systems", *ACM International Conference on Aspect-Oriented Software Development (AOSD)*, Chicago, 2005, 2005, pp. 39-50.

[Walker, *et al.* 1999] R. Walker, E. Baniassad and G. Murphy, "An Initial Assessment of Aspect-Oriented Programming", *International Conference on Software Engineering (ICSE)*, Los Angeles, USA, 1999, pp. 120-130.

[Wirth 1971] N. Wirth, "Program Develoment by Stepwise Refinement", *Communications of the ACM*, vol. 14, no. 4, pp. 221-227, 1971.

[Wong 1999] K. Wong, "The Reverse Engineering Notebook", *Computer Science*, University of Victoria, Victoria, Canada, 1999,

[Wulf and Shaw 1973] W. Wulf and M. Shaw, "Global variable considered harmful", *SIGPLAN Notices*, vol. 8, no. 2, pp. 28-34, 1973.

[Zhang, *et al.* 2005] C. Zhang, G. Gao and H. A. Jacobsen, "Towards Just-in-time Middleware Architectures", *ACM International Conference on Aspect-Oriented Software Development (AOSD)*, Chicago, USA, 2005, pp. 63-74.

[Zhao, *et al.* 2004] T. Zhao, J. Noble and J. Vitek, "Scoped Types for Realtime Java", *International Real-Time Systems Symposium*, Miami, USA, 2004, pp. 101-110.