Aspects of Memory Management

Celina Gibbs and Yvonne Coady
University of Victoria
celinag@uvic.ca, ycoady@cs.uvic.ca

Abstract

With the constant demand for system change and upgrades comes the need to simplify and ensure accuracy in this process. As structural boundaries decay, non-local modifications compound the costs of system evolution and adaptation. Aspect-Oriented Programming (AOP) [11, 12] aims to improve structural boundaries for concerns that are inherently crosscutting – no single hierarchical decomposition can localize both the crosscutting concern and the concerns it crosscuts.

This paper provides a case study of three crosscutting concerns within a rapidly evolving memory management subsystem of a JVM. The study shows how aspects can be structured as a natural locus of control, and how this new modularity provides leverage for system evolution and adaptation. Demonstrated benefits include enhanced extensibility for a dynamic analysis tool, centralized configurability for a subsystem-wide synchronization mechanism, and increased verifiability for a domain-specific design pattern.

1 Introduction

The Jikes Research Virtual Machine (RVM) [7] is a hotbed of research activity. It affords researchers the opportunity to experiment with a variety of design alternatives in virtual machine infrastructure. The project is open source and written in Java. One of the core system elements receiving much attention is garbage collection (GC). State of the art technologies for improving GC performance continue to rapidly evolve, in particular for multiprocessor systems.

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 10 years, and ongoing work aims to further reduce costs and meet application-specific demands [5].

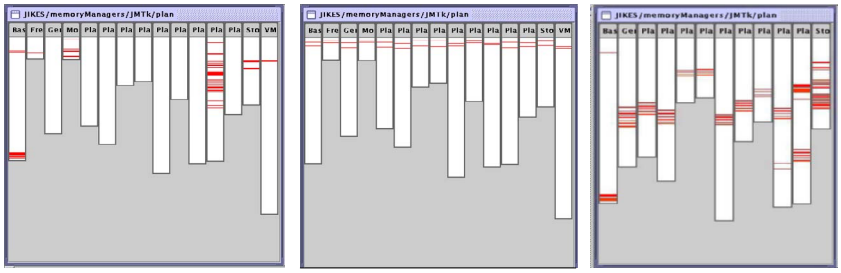
Costs not only involve performance impact, but also configuration complexity. In JDK 1.4.1, for example, there are six collection strategies and over a dozen command line options for tuning GC [5]. Basic strategies, such as reference counting, mark and sweep, and copying between semi-spaces, have been 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.

The memory management subsystem in the Jikes RVM is both modular and efficient [2]. It consists of 134 classes, and supports eight different GC strategies. But even within this well modularized implementation, some concerns naturally defy traditional structural boundaries. They are scattered throughout GC infrastructure, and tangled with the implementation of other concerns in an unclear way. Unstructured, their implementation is a liability to the otherwise natural ebb and flow of evolution within this rapidly changing system. This study considers three crosscutting concerns:

The GCSpy heap visualization tool [22]. This concern allows developers to perform dynamic analysis of memory consumption. Its implementation is currently in place for 1 of eight GC plans, and crosscuts six classes. Figure 1(a) shows a high-level view of each of the key classes involved (including the seven 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[1].

The VM_Uninterruptible interface. This concern provides low-level support for concurrency control by signifying that all methods of a class implementing this 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 1(b).

[1] These figures were generated using the *Aspect Browser*, www.cs.ucsd.edu/usrs/wgg/software/AB



(a) GCSpy **(b) VM_Uninterruptible** **(c) Prepare/Release**

Figure 1: high-level views of scattered implementation. Rectangles represent a collection of source files, and horizontal lines highlight code segments relating to each concern: (a) shows the implementation of GCSpy in 1 of the 8 GC plans, along with its additional functionality required within utility classes, (b) shows a small subset of the 108 classes implementing VM_Uninterruptible, and (c) shows the Prepare/Release protocol across all plans and associated infrastructure.

The Prepare/Release design pattern within GC plans. This concern is a staged protocol that must proceed through a sequence of global and local activities for prepare/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/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 1(c) overviews the implementation of this pattern in the 12 classes it crosscuts.

Each of these concerns provides a fundamentally different kind of system element. GCSpy is a tool we would like to plug/unplug as necessary; VM_Uninterruptible is a synchronization mechanism we would like to configure holistically; and Prepare/Release is a protocol we would like to consistently enforce across plans. But, their evolution and adaptation are impaired by the same deep structural flaw: lack of modularity.

This paper proceeds as follows. Section 2 provides the current hierarchical structure of the code involved in the study. Section 3 provides a high-level overview of these concerns in their AOP implementation. Section 4 contrasts the original and AOP implementations with respect to established trends in evolution and need for adaptation. Section 5 concludes with a discussion of future work.

2 Background: Dominant Decomposition of GC

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.

Plans and policies exist in separate packages and separate hierarchies, as overviewed in Figure 2. 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.

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 *CopyMS* plan, a non-generational, copying/mark-sweep hybrid collector. The policies employed by this plan are *copySpace*, *immortalSpace*, *markSweep* and *treadmill* (for a more detailed treatment of policy specifics see [2]). The hierarchical structure of these classes is illustrated in Figure 2.

In more detail, the top half of Figure 2 shows that all current plans inherit from 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 2 highlights the hierarchical structure of policy: 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.

3 Aspects of GC

This section summarizes the key features of the AOP implementation of GCSpy, VM_Interruptible, and Prepare/Release. In each case, we manually refactored the original system to exclude the concern involved, and reintroduced the concern as an aspect using AspectJ [6].

3.1 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.

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 13 places over 5 classes as shown in Table 1.

Table 1: *if (VM_Interface.GCSPY)*

Class	***Occurrences***
Plan (SemiSpace)	4
MMInterface	2
StopTheWorldGC	4
FreeListResource	1
MonoToneVMResource	2

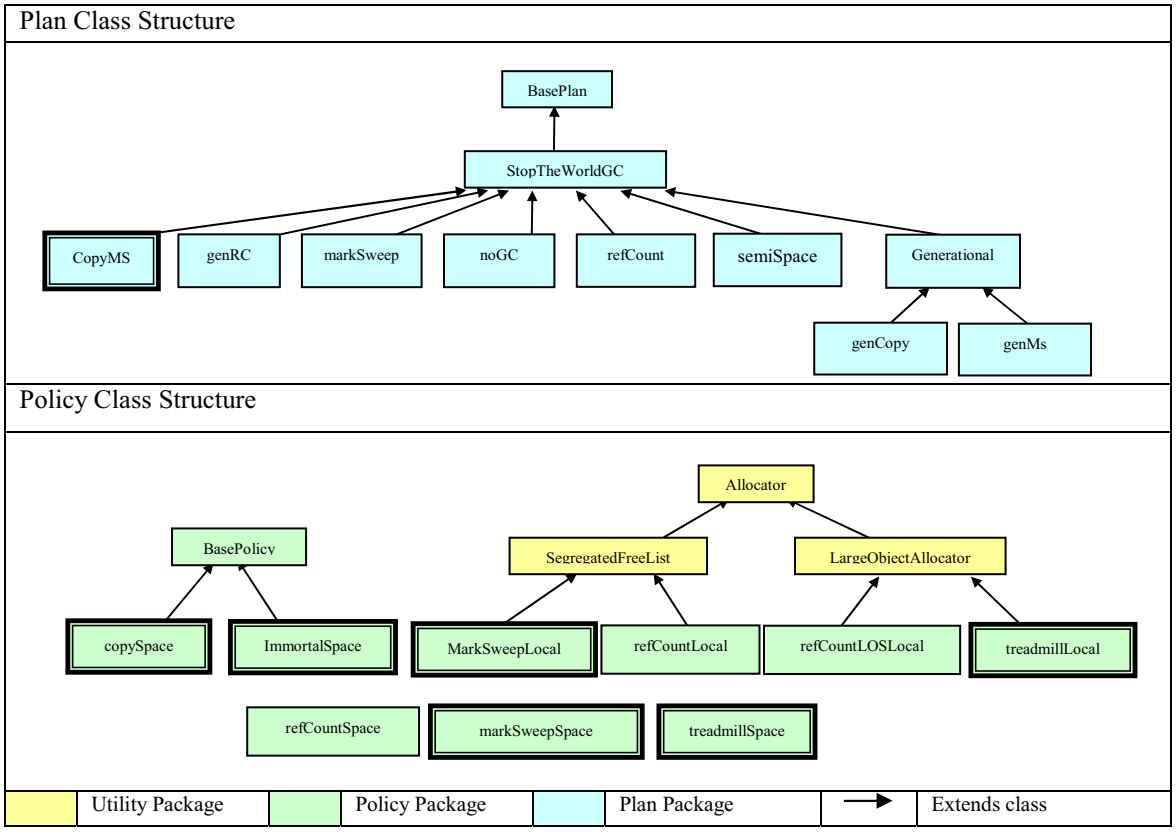


Figure 2: Class hierarchies for plans and policies, highlighting the *CopyMS* plan, and the policies it uses.

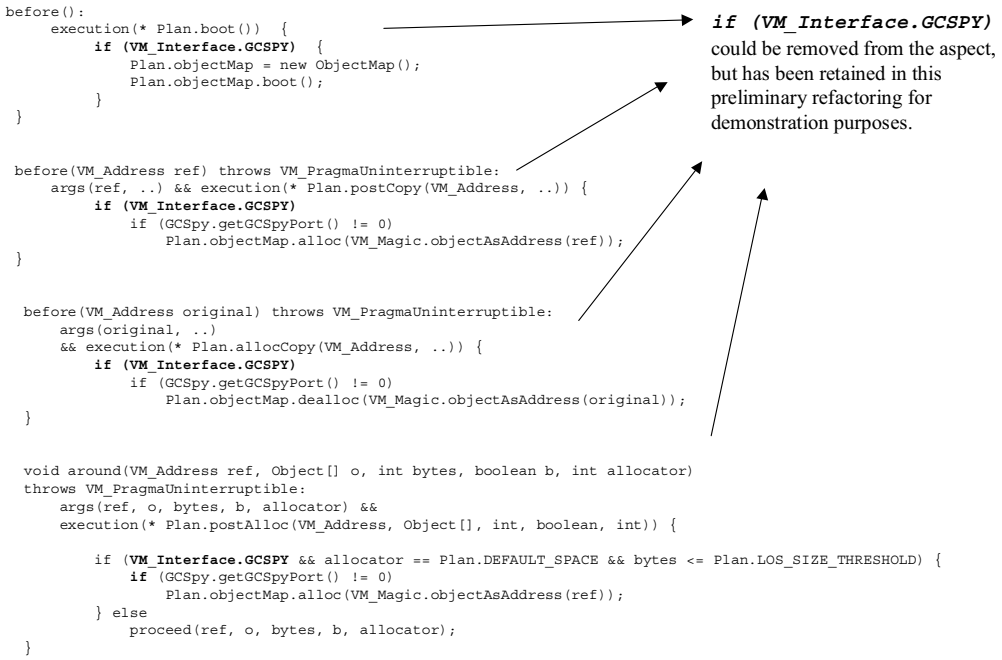


Figure 3: A portion of the CGSpy aspect providing functionality associated with *if(VM_Interface.GCSPY)* for four methods within the SemiSpace Plan class.

In the AOP 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 3. In general, this Figure shows that 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 in Figure 2(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.

In addition to this instrumentation, new methods must be added to existing classes from within the aspect. In total, 13 new GCSpy methods appear across 3 classes as reported in Table 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.

Table 2: *new gcspy methods required*

Class	*Occurrences*
Plan (SemiSpace)	6
BasePlan	5
MonoToneVMResource	2

This aspect can be (un)plugged at compile time to any other GC plan that adheres to this interface. Given that all stop-the-world collectors have a similar structure, the GCSpy aspect is thus more easily extensible between plans than manual instrumentation. Additionally, as indicated in Figure 3, the GCSpy flag is not necessary, as building with the aspect yields an all-or-nothing result.

3.2 VM_Uninterruptible

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 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 takes the following form:

```
aspect Interruptible_Classes_in_MMTk {

  declare parents :
      (org.mmtk.* || com.ibm.JikesRVM.memoryManagers.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 4: Centralized *VM_Uniterruptible* within MMTk.

This aspect shown in Figure 4 uses *declare parents*, which forces one or many classes to implement a specified interface. This construct 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 VM_Uninterruptible synchronization mechanism[2]. Finer-grained synchronization mechanisms, such as *throws VM_PragmaUninterruptible* shown in Figure 3, exist elsewhere in the code. This aspect centralizes configurability of this high-level mechanism.

3.3 Prepare/Release

Currently, the Jikes RVM comes with eight different plans. Advice for developers who want to add a new plan can be found in the Jikes User Guide [8]:

> *"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 4 highlights four of the key methods within the control-flow of *StopTheWorldGC,* showing where plans interact with policies: *globalPrepare()*, *threadLocalPrepare()*, *threadLocalRelease()* and *globalRelease()*.

The large arrow in Figure 5 indicates that the program executes each of the four paths exclusively and sequentially. This ordering is a critical part of GC protocol correctness. Each 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 6 overviews high-level structural relationships between plans and policies. The relationship between the various plans (along the left and right edges of the figure) and their associated policies (in the center of the figure) is indicated by the 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 6.

Given the structural properties outlined in Figures 5 and 6, Figure 7 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 *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

[2] We should note here that tools such as AspectJ Development Tools (http://www.eclipse.org/ajdt/) allow programmers to see this relationship and navigate between aspects and the code they crosscut.

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 Prepare/Release 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 – its significance within Jikes GC plans was recently identified by Blackburn et. al. [2]. Whereas the GCSpy aspect is primarily motivated by extensibility and (un)pluggability, and the VM_Uninterruptible aspect is primarily motivated by holistic and centralized configurability, 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 oversight with *copyMS*). 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 8 shows the similarities and differences between three plans, *copyMS*, *genRC* and *refCount*, from this perspective.

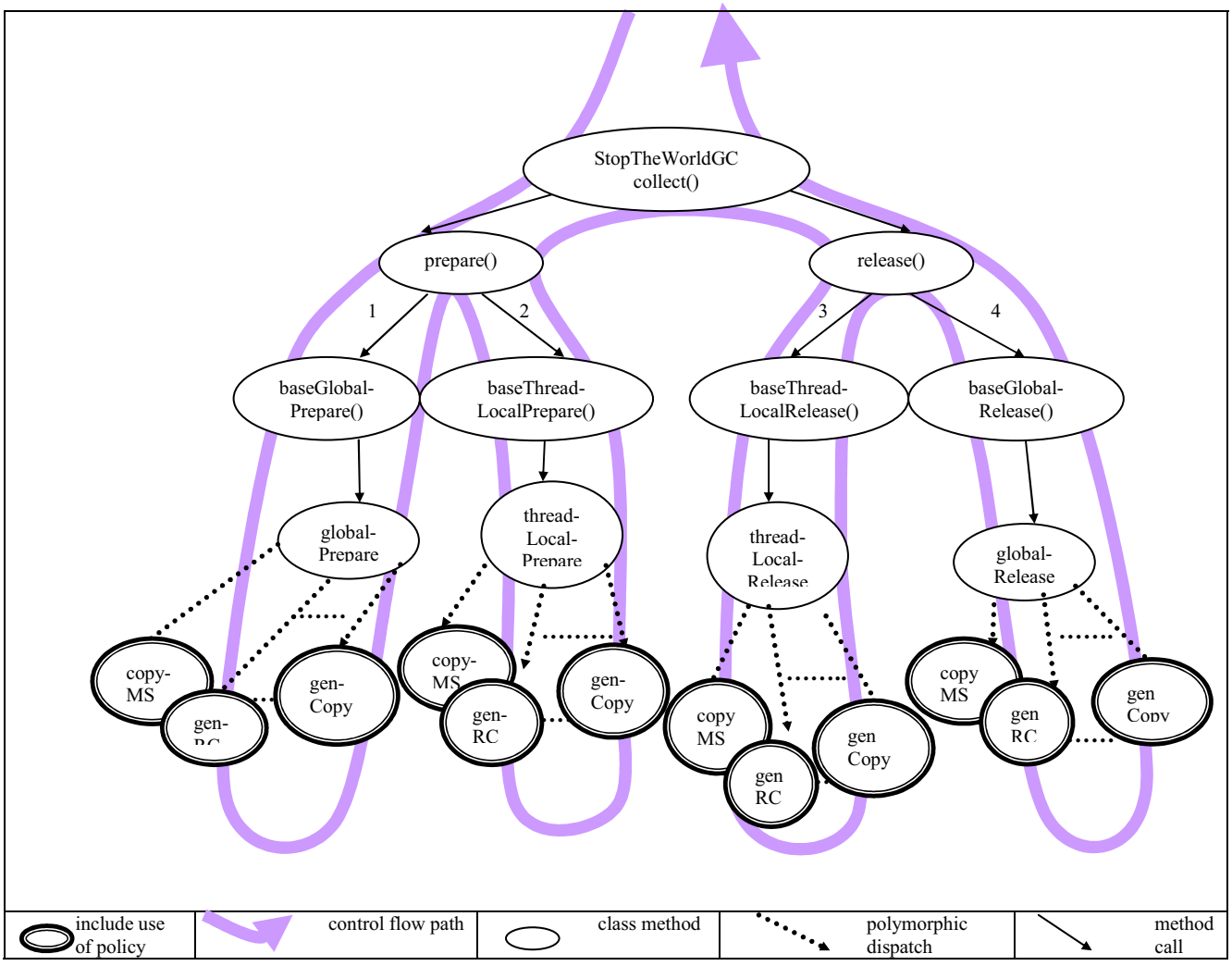


Figure 5: Control flow through prepare/release paths of garbage collection.

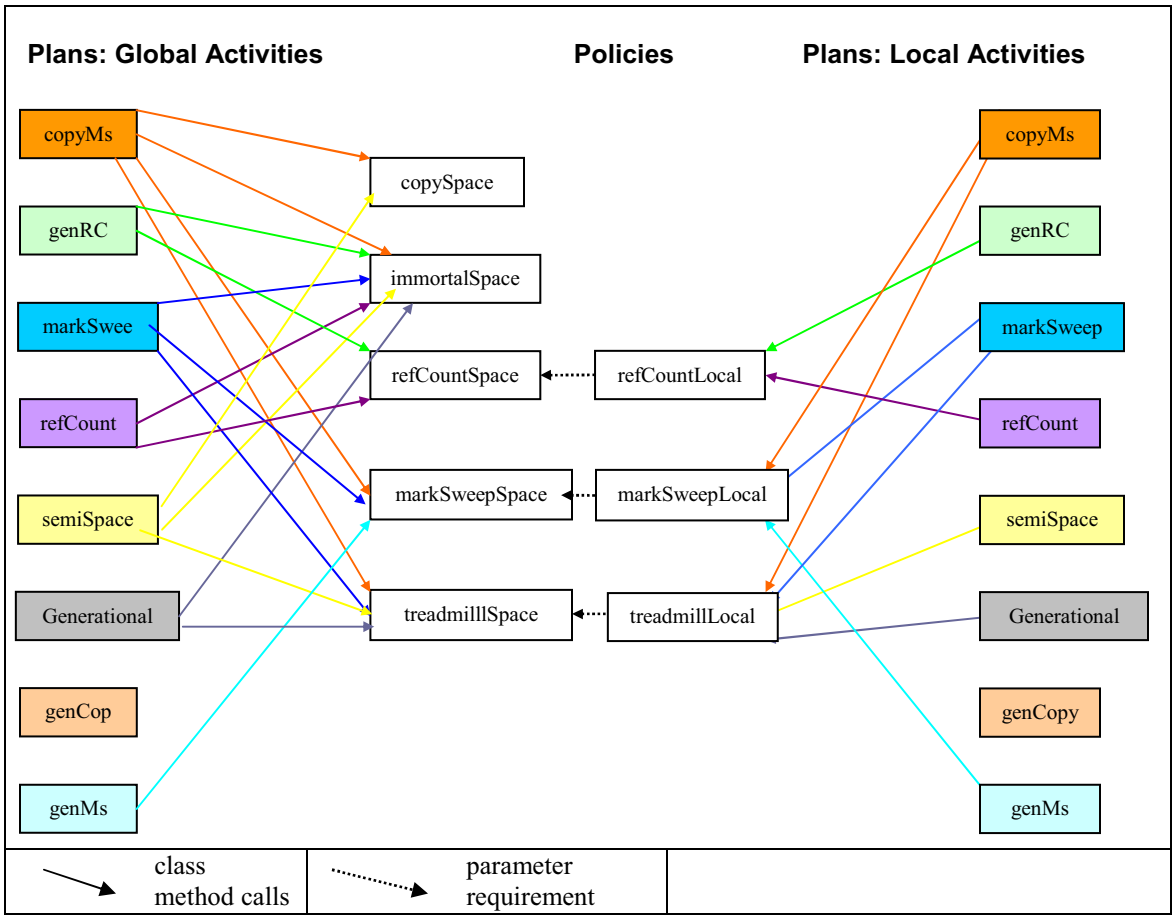


Figure 6: The relationship between global and local plan activities and the internal structure of policy.

		Global					Local		
Policy / Plan		copy Space	immortal Space	refCount Space	markSweep Space	treadmill Space	refCount Local	markSweep Local	treadmill Local
Copy MS	p								
	r								
Gen RC	p								
	r								
MS	p								
	r								
Ref Count	p								
	r								
Semi Space	p								
	r								
Generational	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 7: Prepare/Release crosscuts all plans in a symmetrical manner – the exception in *copyMS* where *copySpace* is not released is an oversight (with no functional consequence) and will be fixed in future versions of the RVM.

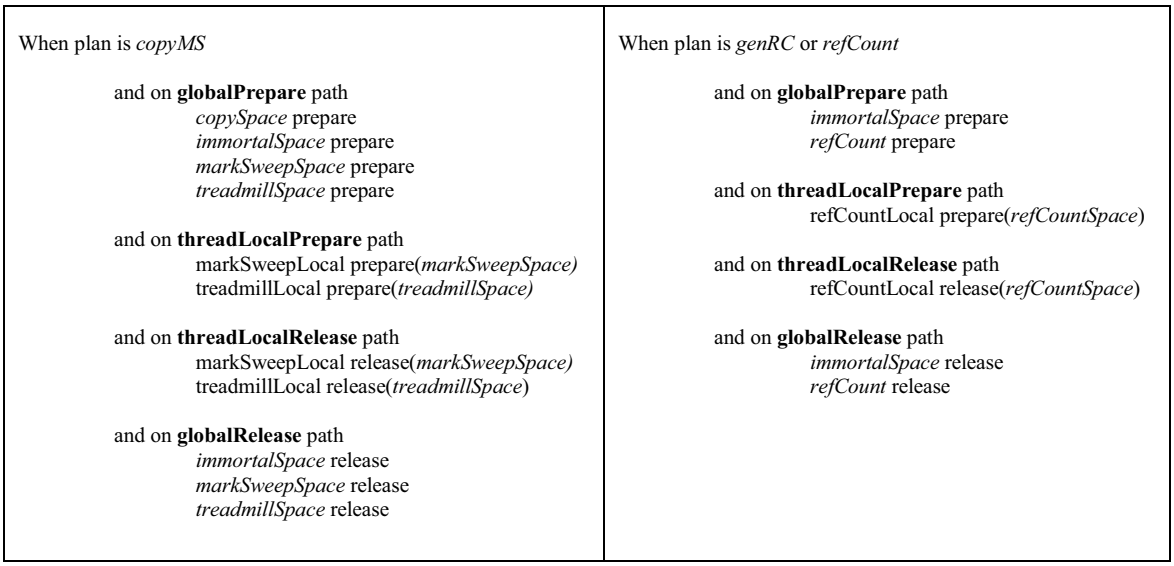


Figure 8: Structure of Prepare/Release in *copyMS, genRC* and *refCount* Plans.

```
package com.ibm.JikesRVM.memoryManagers.JMTk;

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 9: Prepare/Release aspect for *copyMS*

The Prepare/Release aspect for *copyMS* is shown in Figure 9. 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.

4 Impact on Evolution and Adaptation

The long-term goal of this work is to identify the impact of aspects as new structural elements within rapidly growing system infrastructure software. We contend that, as aspects, crosscutting concerns such as those in this case study are more naturally structured, and hence can grow more organically as the system changes over time.

This section begins by supplying a context for AOP in general, and then considers growth trends in the RVM as a context for observations regarding evolution and adaptation in general. Case study results are then considered on a broader scale: from GCSpy results, we consider dynamic analysis tools; from VM_Interruptible results, we consider synchronization mechanisms; and from Prepare/Release results, we consider explicit enforcement of otherwise implicit structure.

4.1.1 *AOP and Separation of Concerns*

The software engineering community has repeatedly demonstrated that structure plays a key role in determining the cost of change [19, 4, 17] and that structure tends to decay over time due to increasing dependencies between modules, further impairing evolution [21, 13]. Structural deficiency results in the need for non-local changes that require considerable effort associated with non-local reasoning [24, 20]. Though no single study has unequivocally shown that one programming methodology, for example a modular structure based on objects, is vastly superior to all others, widespread adoption, indicates that it works well in practice and delivers the expected benefits of good modularity – increased efficiency of the programming process and improvements in the quality of the product [14].

ALGOL 60 (ALGOrithmic language) [1] set a standard for block structure as we know it today. It supported branching, looping, delimited scope of variables, pass by value, pass by name, and recursion. Soon after, Simula67 (SIMULAtion language) [3] provided linguistic support for object-oriented programming, and CLU (function CLUsters) [15, 14] provided linguistic support for data abstraction.

Whereas Simula supported encapsulation if programmers obeyed rules, CLU offered further language enforcement, contributing to a key idea in programming methodology from this same era that focused on separation of concerns [4], organizing systems into separate parts that could be dealt with in relative isolation. The idea of what precisely constitutes a concern remains somewhat nebulous even today[18]. Yet, linguistic support for modules as a collection of operations with hidden information separating the *what* from the *how* has been established as a standard early along. For example, in languages such as C [9, 10], Modula-2 [23], and ML [16], which all supported library modules with separate compilation. In this context, we can consider AOP as another step in linguistic support for better separation of concerns.

4.1.2 Growth Trends in the RVM

Trends driving evolution and adaptation within the memory management subsystem of the Jikes RVM include increasing demand for collection strategies that are more responsive to application specific needs and able to further harness multiprocessing power. In order to match these demands, dynamic analysis tools must ascertain application requirements in the context of system-wide behaviour, synchronization strategies must be efficient and effectively managed, and the overall population and variation of plans must continue to rise.

4.1.3 Dynamic Analysis Tools

Dynamic analysis tools will be crucial for effective future system development. The memory management subsystem is sufficiently complex that even minor code modifications can have dire consequences in terms of application performance. Not only should these tools introduce a minimal amount of overhead, but they should not impact the system when not engaged, and must be easily portable to alternative strategies in a comprehensible manner.

Since AOP can be used to instrument a target system in an all-or-nothing fashion at compile time, aspects can eliminate overhead associated with a disengaged tool. Though, in a performance sense, this is only marginally better than a traditional method that introduces flag-checking (such as *if (VM_Interface.GCSPY)*), the ramifications in terms of scalability and degeneration of structural boundaries are significant.

In terms of extensibility, as GC plans continue to proliferate, the ability to explicitly leverage a well defined interface affords an increased ability to immediately bolt tools on to common infrastructure. The manual alternative, of introducing individual lines of code to selected methods, may stand to win in terms of performance, but loose in terms of portability within the system. The performance question requires further analysis before this win is determined to be conclusive however, as code from aspects could be inlined in performance critical systems.

4.1.4 Synchronization Mechanisms

The configuration of synchronization mechanisms directly impacts performance of GC strategies. This is particularly true for multiprocessor systems. In addition to the issue of interruptibility, write barriers and rendezvous points require careful consideration from a system-wide perspective. Using aspects as a locus of control for configuration allows tuning and management of these mechanisms from a more comprehensive view-point and at a higher-level of abstraction. When centrally configured, developers can explicitly see interactions that impact efficiency due to lack of potential concurrency.

4.1.5 Domain-Specific Design Patterns

Any implicit structure, whether it be design patterns, protocols, or even pre/post conditions, can be easily compromised as new plans mushroom in the system. Plan developers would be less likely to introduce

inadvertently, rule violations if these crosscutting elements were more explicitly represented in the code. Comparative user-studies must be performed in order to help draw more conclusive evidence of this benefit.

5 Future Work and Conclusions

Inevitably, one of the most proficient (and complex) approaches to garbage collection would dynamically select the best strategy appropriate for application-specific behaviour at any given time. Dynamic aspects, deployed at runtime instead of compile time, would allow developers to couple structural relationships with run-time configurability options. We believe this combination would be particularly powerful in the context of future generation collector strategies targeting feedback-based, system-wide optimization. Before dynamic adaptation using dynamic aspects however, we plan to perform further refactoring and analysis of several more static aspects, and an in-depth performance analysis of the system as a whole according to standardized benchmarks.

This work demonstrates enhanced extensibility for the GCSpy aspect, centralized configurability for the VM_Uninterruptible aspect, and increased verifiability for the Prepare/Release aspect. We argue that from these results we can infer benefits for dynamic analysis tools, synchronization mechanisms, and domain-specific design patterns in general. Evolutionary trends within the RVM include: (1) rapid evolution of new plans, (2) adaptation within and between plans, and (3) complex interactions between plans, user applications and other parts of the RVM. We believe that these trends require the structural leverage aspects provide in order to allow crosscutting concerns to grow more organically as the system changes over time.

References

[1] John W. Backus, Friedrich L. Bauer, Julien Green, C. Katz, John McCarthy, Alan J. Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph Henry Wegstein, Adriaan van Wijngaarden and Michael Woodger, *Report on the algorithmic language ALGOL 60*, Communications of the ACM, 3(5), 1960.

[2] Steve Blackburn, Perry Chung and Kathryn McKinley, *Oil and Water? High Performance Garbage Collection in Java with MMTK*, International Conference on Software Engineering (ICSE), 2004.

[3] O.J. Dahl and K. Nygaard, *SIMULA- An Algol Based Simulation Language*, Communications of the ACM, 9(9), 1966.

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

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

[6] IBM, *AspectJ Project*, http://eclipse.org/aspectj/, 2004.

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

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

[9] Brian Kernighan and Dennis Ritchie, *The C Programming Language*, Prentice Hall, First Edition, 1978.

[10] Brian Kernighan and Dennis Ritchie, *The C Programming Language*, Prentice-Hall Software Series, Second Edition, 1988.

[11] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier and John Irwin, *Aspect-Oriented Programming*, European Conference on Object-Oriented Programming (ECOOP), 1997.

[12] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold, *An overview of AspectJ*, Proceedings of 15th European Conference on Object-Oriented Programming (ECOOP), 2001.

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

[14] Barbara Liskov, *A History of CLU*, Massachusetts Institute of Technology MIT-LCS-TR-561, http://www.lcs.mit.edu/publications/pubs/pdf/MIT-LCS-TR-561.pdf, 1992.

[15] Barbara Liskov and Stephen Zilles, *Programming with Abstract Data Types*, Proceedings of ACM SIGPLAN Conference on Very High Level Languages, 1974.

[16] D. MacQueen, *Modules for Standard ML*, University of Edinburgh ECS LFCS 86-2, 1986.

[17] Gail C. Murphy, Lightweight Structural Summarization as an Aid to Software Evolution, Computer Science, University of Washington, PhD Thesis, 1996.

[18] Gail C. Murphy, Albert Lai, Robert J. Walker and Martin P. Robillard, *Separating features in source code: An exploratory study*, Proceedings of the 23rd International Conference on Software Engineering (ICSE), 2001.

[19] D.L. Parnas, *On the Criteria To Be Used in Decomposing Systems into Modules*, Communications of the ACM, 15(12), 1972.

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

[21] W.P. Stevens, G.J. Meyers and L.L. Constantine, *Structured Design*, IBM Systems Journal, Volume 13, 1974.

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

[23] Niklaus Wirth, *Programming in Modula-2*, Springer-Verlag, 3rd, 1985.

[24] W. Wulf and Mary Shaw, *Global variable considered harmful*, SIGPLAN Notices, 8(2), 1973.