MAKING RUNTIME MONITORING OF PARAMETRIC PROPERTIES PRACTICAL

BY

DONGYUN JIN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Doctoral Committee:

Associate Professor Grigore Roşu, Chair and Director of Research
Senior Research Scientist Klaus Havelund, JPL, NASA
Professor Gul Agha
Associate Professor Darko Marinov

Abstract

Software reliability has become more important than ever in recent years, as a wide spectrum of software solutions are being used on various platforms. To this end, runtime monitoring is one of the most promising and feasible solutions for enhancing software reliability. In particular, runtime monitoring of parametric properties (parametric monitoring) has been receiving growing attention for its suitability in object-oriented systems. Despite many parametric monitoring approaches that have been proposed recently, they are still not widely used in real applications, implying that parametric monitoring is not sufficiently practical yet.

In this dissertation, three perspectives for better practicality of parametric monitoring are proposed: *expressiveness*, *efficiency*, and *scalability*. A number of techniques on all three perspectives are developed and integrated to the JavaMOP framework, which is a formalism-independent, extensible runtime monitoring framework for parametric properties. One limitation in expressing parametric properties is that the first event must alway initiate all parameters. This limitation is removed in the proposed work to improve expressiveness of parametric monitoring. Further, a new logical formalism, PTCaRet, is introduced for describing properties of the call stack. As for efficiency, the ‘enable set optimization’, the ‘indexing cache’, and the ‘monitor garbage collection’ are proposed for optimizing creation of monitors, access to monitors, and termination of monitors, respectively. In addition, several scalable parametric monitoring techniques are introduced. These techniques, for the first time, allow a large number of simultaneous parametric specifications to be monitored efficiently.

The optimization techniques presented in this dissertation were implemented into the JavaMOP framework, yielding JavaMOP 3.0, the latest and most efficient version of JavaMOP. Thorough evaluations show that these techniques can improve runtime performance of JavaMOP by 3 times on average, and up to 63 times in some cases; as for memory usage, by 3 times on average. While Tracematches and the previous version of JavaMOP crashed on several cases due to out of memory errors, the newer version of JavaMOP did not crash on any case during the evaluations. Considering that the previous version of JavaMOP was one of the most efficient parametric monitoring frameworks in terms of runtime performance, the results presented in the dissertation can be argued significant.

To my family...

Acknowledgments

I have learned many things throughout my PhD. It truly was a great experience. I was very lucky to have a great advisor, Grigore Roşu, and would like to thank him first. I deeply thank him for training me as an independent researcher, not to mention all of his constructive comments on this dissertation. I would also like to thank my doctoral committee, Doctor Klaus Havelund, Professor Gul Agha, and Professor Darko Marinov, for their insightful comments for the dissertation.

I have truly enjoyed the great research environment of the Formal Systems Laboratory and thank all the past and present members of FSL, Feng Chen, Patrick Meredith, Choonghwan Lee, Chucky Ellison, Traian Şerbănuţă, Andrei Ştefănescu, Mark Hills, Michael Ilseman, Dennis Griffith, Soha Hussein, Kyle Blocher, Yaniv Eytani, and Qingzhou Luo, for all the discussions and help they offered on my research. Special thanks goes to Feng Chen for being a great mentor for my early stage in this research. Also, without his introduction of JavaMOP in 2005, this dissertation would not exist. I pray that his soul is resting in peace. I would also like to thank Patrick Meredith for his great help in presenting my research and for his direct collaboration in developing JavaMOP.

During my PhD, I have been fortunate enough to meet many good friends. I thank Donghee Om and Eunkyung Lee. They have been very close friends and good mentors of my life. I thank all my roommates, Sunjin Im and MyungJoo Ham. It was fun living with you guys. I thank Junho Huh for his great help and comments on this dissertation. I thank Sundeep Katasani and Ravinder Shankesi for previous friendship. I thank my godfather, Sangkym Kim for helping my spiritual growth. I also thank all my friends in Urbana-Champaign, especially Sangchul Lee, Yoonkyung Lee, Kanghoon Jun and Hyunduk Kim.

I would like to thank Professor U Jin Choi for his great support and advice in pursuing the Ph.D. degree. I would like to extend my thanks to all my college friends and members of GoN (the best hacking group in my college), especially Kiho Lee, Juyeong Ji, Ujin Jung, Donghyuk Im, Dongjin Seo, Sangmin Hong, Yunho Kim, and Hongkee Yoon, who have all provided great advices, and triggered creative discussions.

Finally, I would like to thank my family. My parents, Sungkyu Jin and Soonduk Choi, and my sister, Hyeyoon Jin have always encouraged me during my graduate years. None of this would have been possible without great support and love from them. My nephew, Jihwan So was born last year, and I haven't had a chance to meet him yet, but it has been a joy to watch his photos.

The research in this dissertation has been supported in part by NSF grants CCF-0916893, CNS-0720512, and CCF-0448501, by NASA contract NNL08AA23C, and by an NSA grant, a UIUC Campus Research Board Award, and a Samsung SAIT grant.

Table of Contents

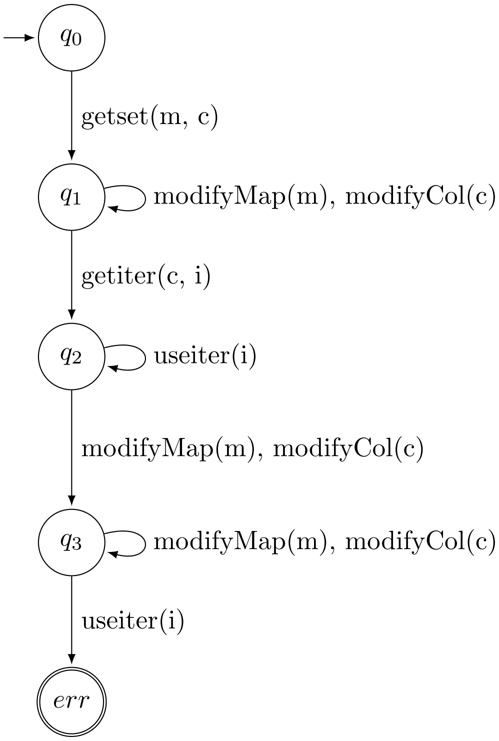
Chapter 1

Introduction

Software reliability has become increasingly more important in recent years, as a very wide spectrum of software solutions are being used on various platforms – ranging from hand-held devices like smart phones to more critical infrastructures like industrial control systems and spaceships. Some software failures result in a huge financial loss. The loss of the NASA Mars Climate Orbiter due to a simple programming error was 327.6 million dollars in total including the cost of the orbiter and the cost of lander [4]. In effect, it slowed down scientific discoveries about Mars. Even worse, some software failures can result in a tragic loss of human lives. Software errors in Therac-25, a radiation therapy machine, caused deaths and serious injuries in at least six known accidents between 1985 and 1987 [60]. Even in our daily lives, software failures in home electronics can hurt user experience, and they might result in a loss of manufacturer's market share.

Many methods from theorem proving to testing have been developed over the last few decades; each method was designed with different goals in mind, and the assurances and the costs are also different. Runtime monitoring is one of the promising methods for enhancing software reliability. Runtime monitoring (simply referred to as *monitoring* for the rest of this dissertation) observes an execution of a system for analyzing its behavior to check if it is faithful to the expected properties. Monitoring can be used not only in system development stages (e.g., debugging, testing) but also in the deployed systems as a mechanism to increase system reliability and/or security.

Parametric properties are properties that describe behaviors of objects (parameter instances), which a program should conform with during its execution. For example, they can describe use of protocols for classes, pre-conditions for using classes, prohibited activities, and so forth. Typestates [71] are a similar concept, but only allow one single parameter, while parametric properties in general can describe properties about any number of parameters. Parametric specifications are formalized parametric properties in some formal specification language with auxiliary information or definitions needed when monitoring. Thus, for the rest of this dissertation, these two terms – parametric property and parametric specification – will be used without distinction, sometimes omitting the word "parametric."



q_0
getset(m, c)
q_1
modifyMap(m), modifyCol(c)
getiter(c, i)
q_2
useiter(i)
modifyMap(m), modifyCol(c)
q_3
modifyMap(m), modifyCol(c)
useiter(i)
err

Figure 1.1: Map_UnsafeIterator property (m: `Map`, c: `Collection`, i: `Iterator`)

For example, `Map` is an interface of map data structures in the standard Java Library that map keys to values. `Map` allows one to iterate all keys/values in its mappings by providing a collection of keys/values. To use this feature, a property must be followed; the Java API documentation states:

> "If the map is modified while an iteration over the set/collection is in progress (except through the iterator's own remove operation), the results of the iteration are undefined."

Since it is allowed to modify other maps which are not being iterated, related method calls (*events*) should be parameterized in this property – meaning that each combination of `Map`, `Collection`, and `Iterator` instances must conform with this property separately. Figure 1.1 represents this parametric property, Map_UnsafeIterator from [59] as a multi-state finite state machine with parameterized events. A parameterized event comes with related parameters which bound to actual parameter instances in runtime (this term is formally defined in Chapter 2). In this property, there are five parameterized events: getset(m, c), getiter(c, i), useiter(i), modifyMap(m), and modifyCol(c). Note that the parameterized events require multiple states (one for each combination of `Map`, `Collection`, and `Iterator` instances) in the finite state machine; Typestates [71] cannot monitor this property.

Figure 1.2 formalizes this property using JavaMOP. While the detailed JavaMOP syntax is discussed in Chapter 2, here we briefly introduce what this

```
Map_UnsafeIterator(Map m, Collection c, Iterator i) {
  creation event getset after(Map m) returning(Collection c) :
    (call(Set Map+.keySet()) || call(Collection Map+.values()))
    && target(m) {}

  event getiter after(Collection c) returning(Iterator i) :
    call(Iterator Iterable+.iterator()) && target(c) {}

  event modifyMap before(Map m) :
    (call(* Map+.clear*(..)) || call(* Map+.put*(..))
    || call(* Map+.remove(..))) && target(m) {}

  event modifyCol before(Collection c) :
      (call(* Collection+.clear(..))
      || call(* Collection+.offer*(..))
      || call(* Collection+.pop(..))
      || call(* Collection+.push(..))
      || call(* Collection+.remove*(..))
      || call(* Collection+.retain*(..))) && target(c) {}

  event useiter before(Iterator i) :
    (call(* Iterator.hasNext(..))
    || call(* Iterator.next(..))) && target(i) {}

  ere : getset (modifyMap | modifyCol)* getiter useiter*
        (modifyMap | modifyCol)+ useiter

  @match {
    System.err.println("a violation detected!");
  }
}
```

Figure 1.2: Map_UnsafeIterator specification in JavaMOP

specification contains. This specification defines five parametric events with the corresponding AspectJ [54] pointcuts that pick out interesting program points. The property is formalized using an extended regular expression (ERE), as specified by the ere keyword. If a program behavior matches this pattern, and violates the property from the Java API documentation, the defined handler containing the user-defined Java code will be executed; here we simply print out an error message in the handler. Handler can be any code, from logging to recovery.

Monitoring of Parametric properties (referred to as *parametric monitoring* for the rest of this dissertation) enables us to analyze program behaviors more precisely, especially in object-oriented programs. A user or a software developer describes specifications that a program should conform with, like the one in Figure 1.2. Then, JavaMOP will generate actual monitoring code and instrumentation code in AspectJ [54], which can contain a large number of lines of code that is not easy to code manually. By using any AspectJ compiler like ajc [15], this monitoring code can be instrumented into the original program and monitor its execution.

1.1 Motivation

In spite of significant recent progress and several parametric monitoring approaches being developed, parametric monitoring is still not widely used in real-life software applications. For making parametric monitoring more practical, there are still challenges remaining that need to be addressed.

First, parametric monitoring should be *scalable* to the number of properties to monitor. To the best knowledge of the author, all earlier efforts on parametric monitoring have been focusing on monitoring a single property more efficiently and/or effectively [51, 64, 36, 19, 62, 18]. However, in real usages, there are likely to be multiple properties that have to be monitored for a program. Some monitoring systems support simultaneous parametric monitoring of multiple properties with overheads greater than the sum of overheads from monitoring each property, which can easily become too large to tolerate. To make a parametric monitoring system more practical, it is crucial to support *scalable* simultaneous parametric monitoring of multiple properties.

Second, parametric monitoring should be *efficient*, i.e., runtime and memory overheads of monitoring a property should be as small as possible. Also, we are interested not only in average performance but also in worst case performance. While many parametric monitoring systems show efficient average performance, there are still some extreme cases that many monitoring systems cannot monitor due to excessive overheads. For example, monitoring a property about `Iterator` in Java can show a huge overhead since the `Iterator` type can be heavily used in many programs. Although there are many static analysis techniques that can be used to reduce overheads, they only analyze places where runtime monitoring cost can be hidden; monitoring costs incurring from other places will still exist and will be counted as part of the overall overhead. Moreover, most of static analysis is formalism-dependent. When a new logical formalism is introduced, new static analysis have to follow for efficient monitoring of properties in the new logical formalism. Also, static analysis techniques might not be effective in some cases, depending on the nature of the monitored program and the specifications. Thus, improving runtime monitoring cost itself is definitely beneficial. Furthermore, two different types of techniques are orthogonal to each other, implying that more improvements can be achieved from using both techniques.

Third, parametric monitoring should be *expressive*. From the nature of parametric monitoring that observes behaviors of parameters separately, it introduces overheads that can easily be excessive. To the best knowledge of the author, all parametric monitoring systems such as [32, 19, 62, 43, 21] among others, have restrictions on monitoring parametric properties for this performance reason. Many of them follow a formalism-dependent approach, that is, they hardwire their parametric specification formalisms as a feasible solution to this, limiting expressiveness and leading to inefficient monitoring. Note that some properties can be monitored more efficiently in different logical formalisms.

Another approach, that the previous version of JavaMOP chose, is to restrict creation events to initiate all parameters. Although, in this way, parametric monitoring can be efficient, focusing on monitoring fully instantiated parameter instances only, this restriction hurts expressiveness in describing parametric properties. For example, the Map_UnsafeIterator property in Figure 1.1 cannot be expressed with this restriction since the creation event, getset instantiates only two parameters out of the three. Practical parametric monitoring must support various logical formalisms without putting any restrictions on expressing properties.

To apply parametric monitoring to more areas, we need to develop *scalable*, *efficient* and *expressive* parametric monitoring techniques which are capable of monitoring multiple specifications simultaneously, while supporting multiple formalisms. The ultimate goal is to provide a practical parametric runtime monitoring framework that can be used in real software development stages and even in deployed systems for enhancing reliability and security.

1.2 Contributions

This dissertation presents research for practical runtime monitoring of parametric properties, and an application to multi-threaded unit testing. The research outcomes from this dissertation have been integrated into JavaMOP and resulted in a new, more practical version. Before the advancements made through this work, JavaMOP was not capable of addressing the challenges listed in Section 1.1. The key contributions of this dissertation are as follows:

1. The practical JavaMOP framework

   (a) **Scales in the number of specifications that it monitors at the same time.** For making parametric monitoring practical, it is essential to support *efficient* monitoring of multiple simultaneous properties. If the runtime and memory overhead increases linearly (or worse), parametric monitoring easily becomes prohibitive with the existence of a large number of properties. Based on a reasonable assumption that some properties describe behaviors of the same parameter, sharing some events and parameters, parametric monitoring can be done more efficiently than linear sum of overheads that would incur from monitoring them separately. This work is described in detail in Chapter 5

   (b) **Shows the best runtime performance and competitive memory performance compared to other parametric monitoring systems.** The previous version of JavaMOP already showed a better runtime performance than other parametric monitoring systems in most cases and a reasonable memory performance. A number of optimization techniques, presented in Chapter 4, dramatically improve this runtime and memory performance of the JavaMOP framework even further – resulting in orders of magnitude faster runtime performance and competitive memory performance compared to runtime performance of other systems.

(c) **Supports multiple logical formalisms in describing properties.** The previous version of JavaMOP already supported multiple logical formalisms including extended regular expressions, finite state machines, and linear temporal logics.

In addition to those logical formalisms, another logical formalism called *past time linear temporal logic with calls and returns* (PTCaRet) is implemented, which is an extension of past time linear temporal logic; this work is described in Chapter 3. *Context-free grammar* is added by Patrick Meredith [64], in collaboration with the author, but this will not be covered in this dissertation. To ensure that multiple logical formalisms are still supported, all the work done on the JavaMOP framework chooses formalism-independent approaches.

(d) **Provides parametric monitoring without any restriction for better expressiveness in describing parametric properties.** Due to performance reasons, the previous version of JavaMOP has a restriction that all creation events must instantiate all parameters; this is so that JavaMOP can focus on monitoring fully instantiated parameter instances only. This dissertation removes this restriction, while keeping its efficiency, by using the *enable set optimization*, which tells what parameter instances need to be monitored. The outcomes from this work are described in detail in Chapters 3 and 4.

2. Application of JavaMOP

JavaMOP is used in the improved multithreaded unit testing framework for describing/monitoring events and enforcing the desired schedules. To enable this, a new logic plugin called *partial orders* is implemented and integrated with the JavaMOP framework. Thanks to the JavaMOP architecture, there were not too many technical barriers in implementing the logic plugin, yet it gets all the advantages of the JavaMOP framework.

1.3 Dissertation Overview

Chapter 2 provides the background information on JavaMOP, that is required to understanding the rest of the dissertation. We introduce the JavaMOP framework with its architecture, the syntax of JavaMOP, and provide examples. Also, we explain the indexing tree technique, which is the key technique in implementing parametric monitoring. In this chapter, we focus on the indexing tree technique that is used in the previous version of JavaMOP, while other chapters discuss the modifications and improvements that were made.

Chapter 3 describes the author's work on expanding the expressiveness of JavaMOP. The limitation in expressing parametric properties that creation events must initiate all parameters, is removed; it allows more parametric properties to be written in JavaMOP, including some examples found in Chapter 2. A new logical formalism, Past Time Linear Temporal Logic with Calls and Returns (PT-CaRet) for monitoring stack-based properties is implemented. Also a specification inheritance for reusing specifications just like the Java inheritance, is supported.

Chapter 4 offers several optimization techniques to improve the runtime and memory performance of parametric monitoring. First, the *enable set* optimization avoids monitoring parametric instances that are not need to be monitored. Then, the new data structures for the indexing tree are proposed, and unnecessary monitors can be garbage collected within the new data structures by using the *co-enable set* optimization. The indexing cache for the indexing tree is introduced for reducing the number of expensive operations that need to be performed on the indexing tree.

Chapter 5 presents scalable parametric monitoring techniques, which are capable of monitoring more than 100 parametric specifications *simultaneously*. The common part in the indexing trees is extracted into a shared resource between multiple specifications. Indexing trees within each specification are also combined when possible, reducing the space usage and the maintenance cost. Also, simple specification activators effectively suppress unnecessary overhead from inactive specifications.

Chapter 6 introduces some application of parametric monitoring. JavaMOP is used for monitoring/enforcing the desired schedules to improve the multi-threaded unit testing. Instead of using sleep statements, which are fragile and slow, in unit testings, we propose an improved multi-threaded unit testing framework that a user can explicitly describe the desired schedule for testing. Then, the framework monitors the thread scheduling and enforces the desired ones. In the framework, JavaMOP takes charge of monitoring and enforcing the schedules.

Finally, Chapter 7 discusses related work, and Chapter 8 concludes the dissertation with opportunities for future research.

1.4 Relationship to Previous Work

Monitoring-Oriented Programming [30], abbreviated MOP, is a generic monitoring framework that integrates specification and implementation by checking the former against the latter at runtime. JavaMOP which was first introduced by Chen and Rosu [31, 32], is an instance of the MOP framework specific to the Java programming language. The research on parametric monitoring in this dissertation integrates into the JavaMOP framework, resulting in a series of new versions of JavaMOP. Before the research in this dissertation, JavaMOP was showing a reasonable runtime and memory performance. This dissertation presents expressive, efficient, and scalable parametric monitoring techniques, promoting JavaMOP to one of the fastest and the most scalable parametric monitoring frameworks with competitive memory performance. The parametric monitoring algorithm without any limitation on parameters has first been proposed by Chen and Rosu [33], which is *efficiently* implemented in this dissertation. Monitoring algorithm for the Past Time Linear Temporal Logic with Calls and Returns (PTCaRet) has been proposed by Rosu et al. [69], which is also *efficiently* implemented in this dissertation, along with an optimization to handle a huge number of method calls and returns.

1.5 Related Publications

This section provides a quick overview of the author's publications that are relevant to the research presented in this dissertation. All the work in this dissertation was done in collaboration with Grigore Roşu. The work on the JavaMOP framework and all the optimizations are done in collaboration with Patrick Meredith, Feng Chen, Choonghwan Lee, and Dennis Griffith. The improved multithread unit testing framework is developed in collaboration with Vilas Jagannath, Milos Gligoric, Qingzhou Luo and Darko Marinov.

JavaMOP Framework All work on parametric monitoring in this dissertation is integrated into the JavaMOP framework, resulting in a new version of JavaMOP. The JavaMOP tool was introduced in a number of publications including *'JavaMOP: Efficient Parametric Runtime Monitoring Framework'* [52]presented at the **2012 International Conference on Software Engineering Formal Demonstrations**, and *'Monitoring Oriented Programming - A Project Overview'* [35]presented at the **2009 International Conference on Intelligent Computing and Information Systems**. Also, the detailed syntax and semantics of the JavaMOP framework, the structure of the JavaMOP framework, and the detailed syntax, semantics, and monitoring algorithm of each logical formalism that the JavaMOP framework supports are explained in *'An Overview of the MOP Runtime Verification Framework'* [65]in the **Journal on Software Tools for Technology Transfer**. Since the JavaMOP framework is used throughout the dissertation, Chapter 2 summarizes its structure, syntax, and semantics, as well as the indexing tree technique, on which many parametric monitoring systems are based on.

Parametric Monitoring Parametric Monitoring was supported in the previous version of JavaMOP framework, but with the restriction that creation events must instantiate all parameters. It is not trivial to remove this restriction since monitoring all possible combinations of parameter instances is infeasible. To address this, *'Efficient Formalism-Independent Monitoring of Parametric Properties'* [36]which is presented at the **2009 International Conference on Automated Software Engineering** , proposes *enable set optimization*, which is described in detail in Chapter 4. It enables general parametric monitoring without any restriction; it also expands expressiveness of parametric monitoring that is explained in Chapter 3. All other work on parametric monitoring described in this dissertation is done based on this.

Monitor Garbage Collection Parametric monitoring introduces a large number of monitors even after optimizations. Therefore, it is important to garbage collect unnecessary monitors to improve the memory performance as well as the runtime performance. However, it is not trivial to collect unnecessary monitors *efficiently.* The efficient monitor garbage collection, found in Chapter 4, is presented in *'Garbage Collection for Monitoring Parametric Properties'* [51]which is presented in **2011 Programming Language Design and Implementation**, improving the performance of parametric monitoring greatly.

Scalable Parametric Monitoring For monitoring multiple specifications more efficiently, Chapter 5 introduces Scalable parametric monitoring techniques, which are also presented in *'Scalable Parametric Runtime Monitoring'* [53]. These scalability techniques improve the efficiency of JavaMOP with respect to monitoring multiple simultaneous specifications; in addition, performance for single specification cases are also improved.

Improved Multithreaded Unit Testing Monitoring ability of JavaMOP is applied to unit testing in multithreaded environments for monitoring/enforcing the thread scheduling. This provides more benefits over traditional sleep-based multithreaded unit tests which are unreliable and slow. This work is published as *'Improved Multithreaded Unit Testing'* [47, 48] in **2011 Foundations of Software Engineering** and the **2010 International Workshop on Multicore Software Engineering**, and also explained in Chapter 6.

Chapter 2

Background

This chapter provides the background information necessary to follow the rest of the dissertation. We present the architecture of JavaMOP in Section 2.1, the syntax of JavaMOP in Section 2.2, and examples of JavaMOP specification in Section 2.3. We formally define parametric monitoring and give algorithm in Section 2.4. Also, we explain the indexing tree technique that JavaMOP uses for locating associated monitors upon each event, in Section 2.5. Since all the work in this dissertation is based on the JavaMOP framework, we focus on parametric monitoring and the indexing tree technique used in the previous version of JavaMOP. Then, the current version of JavaMOP is explained in the following chapters, covering the improvements made. There are many other parametric monitoring frameworks that directly or indirectly use the indexing tree technique. Many optimization techniques presented in this dissertation can also be applied to these frameworks.

2.1 JavaMOP

JavaMOP [31, 32] is an instance of the generic MOP framework specific to the Java programming language. It allows concise descriptions of parametric properties using a combination of event specifications that uses an extension of AspectJ [54] as well as properties specified over these events. From these specifications, JavaMOP generates AspectJ code for monitoring, which is weaved into the target program by any AspectJ compiler, such as the standard AspectJ compiler `ajc`. In this way, the generated monitoring code observes the program, catches the events defined by a specification, and checks whether the program is compliant to the given specification. When a specification is validated or violated, user-defined code, called handlers, are executed. User-defined code can be any Java code ranging from a code that performs logging to something that performs runtime recovery according to a user's objectives. The ability to supply actual recovery code in the handlers allows JavaMOP-generated monitors to enforce specifications within a program.

Table 2.1 lists a number of other monitoring systems and the logical formalism they support. While all monitoring systems in the table except JavaMOP follow a formalism-dependent approach, that is, they hardwire their parametric

Approach	Language	Logic	Scope	Mode	Handler
Hawk [38]	Java	Eagle	global	inline	violation
J-Lo [25]	Java	ParamLTL	global	inline	violation
Jass [23]	Java	assertions	global	inline	violation
JavaMaC [56]	Java	PastLTL	class	outline	violation
jContractor [8]	Java	contracts	global	inline	violation
JML [58]	Java	contracts	global	inline	violation
JPaX [44]	Java	LTL	class	offline	violation
P2V [61]	C, C++	PSL	global	inline	validation/ violation
PQL [62]	Java	PQL	global	inline	validation
PTQL [43]	Java	SQL	global	outline	validation
Spec# [20]	C#	contracts	global	inline/ offline	violation
RuleR [22]	Java	RuleR	global	inline	violation
Temporal Rover [40]	*several*	MiTL	class	inline	violation
Tracematches [19]	Java	Reg. Exp.	global	inline	validation

Table 2.1: Runtime monitoring breakdown

specification formalisms (limiting expressiveness and leading to inefficient monitoring, JavaMOP follows a formalism-independent approach). We believe that there is no ultimate formalism that concisely expresses every property. Users will likely use different formalisms, depending on what properties they want to express. Four logic-plugins were provided with JavaMOP before this dissertation: Java Modeling Language (JML) [58], Extended Regular Expressions (ERE), and Past-Time and Future-time Linear Temporal Logics (PTLTL and FTLTL) [34]. We have introduced two more logical formalisms, namely Context-Free Grammar (CFG) [63] and Past-Time Linear Temporal Logics with Calls and Returns (PTCaRet). However, only PTCaRet is covered in the dissertation since CFG was not one of the author's primary contributions.

Figure 2.1 shows the architecture of JavaMOP. JavaMOP communicates with the Logic Repository which generates platform-independent monitoring pseudo-code for the given property. All logical formalism plugins are contained within the Logic Repository, which is a standalone program that can be used with other instances of the MOP framework (e.g., CMOP, C#MOP, PythonMOP and so on). The JavaMOP component has several translators to interpret pseudo-code from the Logic Repository into AspectJ monitoring code. From this architecture, one can easily introduce new logical formalism into JavaMOP by adding a logic plugin – this can be done without worrying about other features and optimizations which are formalism-independent. JavaMOP provides several interfaces including a web-based interface, a command-line interface and an Eclipse-based GUI, providing the developer with different means to manage and process MOP specifications. For other usages, interfaces can be extended as well sharing the core implementation of JavaMOP.

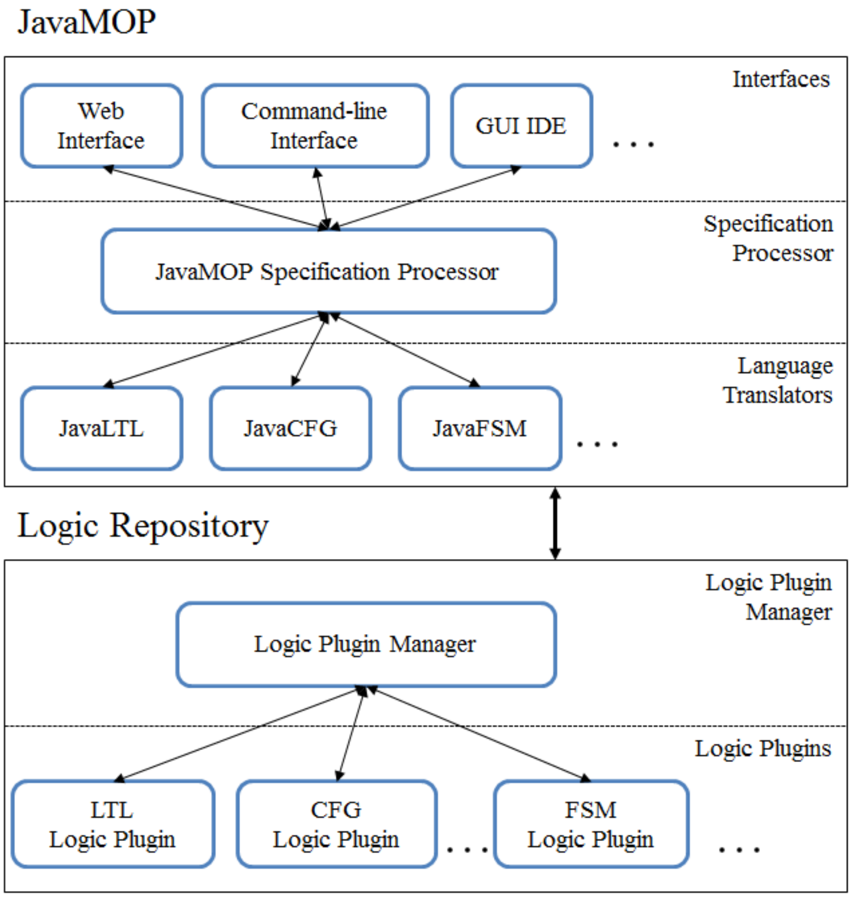


Figure 2.1: JavaMOP architecture

For performance reasons, the previous version of JavaMOP had a limitation that the creation event was required to initiate all parameters. If all parameters are always initiated at the creation event, we know exactly what parameter instances to monitor. Otherwise, we should monitor all possible parameter instances, including combinations of multiple parameter instances. It was not possible to do this *efficiently* before this dissertation, thus the previous version of JavaMOP chose an easy way around this, only allowing the first case. The research to eliminate this limitation can be found in Chapter 3.

Since JavaMOP relies on an external AspectJ compiler for weaving monitoring code, JavaMOP only allows events which can be defined in the standard AspectJ. There are many extended components of AspectJ in different tools and they are shown to be useful ([26, 39, 46, 11] among others). However, it is unknown which AspectJ compiler is best suited for everything. There is unlikely to be a best compiler as such. Conservatively, JavaMOP only supports components from the standard AspectJ, allowing users to choose their AspectJ compiler. Also, weaving-time optimization should be done independently from JavaMOP since the resulting code from JavaMOP is program-independent as well.

2.2 JavaMOP Syntax

As there can be multiple instances of MOP (e.g., BusMOP [66] for monitoring system buses using FPGA-based monitors), the MOP syntax is structured to keep consistency. Figure 2.2 shows the structured MOP syntax. All of the grammars used to define MOP syntax in this section use Extended Backus-Naur Form (EBNF) [2]. Non-terminals in the grammars are surrounded by "⟨" and "⟩". Braces ("{" and "}") enclose portions of the grammar that may appear zero or more times. Brackets ("[" and "]") enclose portions of the grammar that are optional (i.e., it may or may not appear). Concrete examples of the syntax defined below can be found in Section 2.3. There are the shared syntax for every MOP instance, the instance specific syntax, and the logic plugin specific syntax that each logic can freely define. The syntax of any instance of MOP can be generated by defining certain syntactic categories (non-terminals) of the MOP syntax.

The following syntax constructs are shared by different MOP instances:

- ⟨*Specification*⟩ — It describes the generic MOP specification syntax which can be instantiated for MOP language instances and MOP logic plugins.

- ⟨*Event*⟩ — The ⟨*Event*⟩ declaration code allows for the definition of events, which may then be referred to in the property (see ⟨*Property*⟩ below). Event declarations can also have arbitrary code associated with them (⟨*Instance Action*⟩), which is run when the event is observed (⟨*Instance Event Definition*⟩), e.g. code to modify the program or the monitor state. For manual indication of events that can start a trace, the keyword creation is used at the beginning of each declaration.

- ⟨*Property*⟩ — Every MOP specification may contain zero or more properties. A ⟨*Property*⟩ consists of a named formalism (⟨*Logic Name*⟩), followed by a colon, followed by a property specification using the named formalism (see ⟨*Logic Syntax*⟩ below) and usually referring to the declared events. If the property is missing, then the MOP specification is called *raw*. Raw specifications are useful when no existing logic plugin is powerful or efficient enough to specify the desired property; in that case, one embeds the custom monitoring code manually within the ⟨*Instance Action*⟩ code.

- ⟨*Property Handler*⟩ — Handlers contain arbitrary code from the instance source language, and are invoked when a certain logic state (see ⟨*Logic State*⟩ below) or category is reached, e.g., match, fail, or a particular state in a finite state machine description.

The following constructs are based on the particular instance of MOP used for a particular specification. JavaMOP should define these six components in the instance specific syntax. Figure 2.3 shows the JavaMOP specific syntax, which defines six components listed below and auxiliary components for them.

```
Shared syntax
            ⟨Specification⟩  ::=  {⟨Instance Modifier⟩} ⟨Id⟩ ⟨Instance Parameters⟩ “{”
                                     {⟨Instance Declaration⟩}
                                     {⟨Event⟩}
                                     {⟨Property⟩
                                        {⟨Property Handler⟩}
                                     }
                                  “}”
                    ⟨Event⟩  ::=  [“creation”] “event” ⟨Id⟩ ⟨Instance Event Def⟩ “{”
                                     ⟨Instance Action⟩
                                  “}”
                 ⟨Property⟩  ::=  ⟨Logic Name⟩ “ : ” ⟨Logic Syntax⟩
         ⟨Property Handler⟩  ::=  “@” ⟨Logic State⟩ ⟨Instance Handler⟩

Instance-specific syntax
        ⟨Instance Modifier⟩  ::=  ⟨Id⟩
      ⟨Instance Parameters⟩  ::=  ⟨JavaMOP Parameters⟩
                               |  ⟨BusMOP Parameters⟩
                               |  ...
     ⟨Instance Declaration⟩  ::=  ⟨JavaMOP Declaration⟩
                               |  ⟨BusMOP Declaration⟩
                               |  ...
       ⟨Instance Event Def⟩  ::=  ⟨JavaMOP Event Definition⟩
                               |  ⟨BusMOP Event Definition⟩
                               |  ...
          ⟨Instance Action⟩  ::=  ⟨JavaMOP Event Action⟩
                               |  ⟨BusMOP Event Action⟩
                               |  ...
         ⟨Instance Handler⟩  ::=  ⟨JavaMOP Event Handler⟩
                               |  ⟨BusMOP Event Handler⟩
                               |  ...

Logic-plugin-specific syntax
               ⟨Logic Name⟩  ::=  ⟨Id⟩
             ⟨Logic Syntax⟩  ::=  ⟨FSM Syntax⟩ | ⟨ERE Syntax⟩ | ⟨LTL Syntax⟩
                               |  ⟨PTLTL Syntax⟩ | ⟨CFG Syntax⟩
                               |  ⟨PTCaRet Syntax⟩ | ...
              ⟨Logic State⟩  ::=  ⟨FSM State⟩ | ⟨ERE State⟩ | ⟨LTL State⟩
                               |  ⟨PTLTL State⟩ | ⟨CFG State⟩
                               |  ⟨PTCaRet State⟩ | ...
```

Figure 2.2: MOP syntax

⟨*JavaMOP Modifier*⟩	::=	“*full–binding*” | “*maximal–binding*”
	|	“*any–binding*” | “*connected*” | “*unsynchronized*”
	|	“*decentralized*” | “*perthread*” | “*suffix*”
⟨*JavaMOP Parameters*⟩	::=	“(” [⟨*JavaMOP Type*⟩ ⟨*Id*⟩ {“,” ⟨*JavaMOP Type*⟩ ⟨*Id*⟩}] “)”
⟨*JavaMOP Declaration*⟩	::=	syntax of declarations in Java
⟨*JavaMOP Event Def*⟩	::=	⟨*AspectJ AdviceSpec*⟩ “ : ” ⟨*AspectJ Pointcut*⟩ [“&&” ⟨*JavaMOP Pointcut*⟩]
⟨*JavaMOP Action*⟩	:=	Java statements, which may refer to monitor local variables
⟨*JavaMOP Handler*⟩	:=	Java statements with additional keywords
⟨*JavaMOP Type*⟩	:=	Any valid Java type
⟨*AspectJ AdviceSpec*⟩	::=	syntax of AdviceSpec in AspectJ
⟨*AspectJ Pointcut*⟩	::=	syntax of Pointcut in AspectJ
⟨*JavaMOP Pointcut*⟩	::=	“*thread*” “(” ⟨*Id*⟩ “)”
	|	“*condition*” “(” ⟨*Boolean Exp*⟩ “)”
	|	“*endProgram*” “(” “)”
	|	“*endObject*” “(” ⟨*Id*⟩ “)”
	|	“*endThread*” “(” “)”
	|	⟨*AspectJ Pointcut*⟩
	|	⟨*JavaMOP Pointcut*⟩ “&&” ⟨*JavaMOP Pointcut*⟩
⟨*Boolean Exp*⟩	::=	⟨*Id*⟩ | “!” ⟨*Boolean Exp*⟩
	|	⟨*Boolean Exp*⟩ ⟨*Boolean Operator*⟩ ⟨*Boolean Exp*⟩
	|	“(” ⟨*Boolean Exp*⟩ “)”
⟨*Boolean Operator*⟩	::=	“ || ” | “&&” | “ | ” | “&” | “ == ” | “! = ”

Figure 2.3: JavaMOP syntax

- ⟨*Instance Modifier*⟩ — ⟨*Instance Modifier*⟩s are specific to each language instance of MOP. Syntactically, they can be any valid identifier restricted by the given language. They change the behavior of the monitoring code.

- ⟨*Instance Parameters*⟩ — allow one to define the parameters of a parametric specification using the language corresponding to the MOP instance. Not all MOP instances are parametric (e.g., BusMOP), however, so this non-terminal may be empty.

- ⟨*Instance Declaration*⟩ — ⟨*Instance Declaration*⟩s are specific to each instance of MOP. They allow for the declaration of monitor local variables.

- ⟨*Instance Event Definition*⟩ — ⟨*Instance Event Definition*⟩s are specific to each language instance of MOP. They define the conditions under which an event is triggered.

- ⟨*Instance Action*⟩ — An event can have arbitrary code associated with it, called an action. The action is run when the event is observed. An action can modify the program or the monitor state, and the syntax of the allowed statements are dependent upon the MOP instance in question. Typically the statements used in actions have different variables and functions that may be referred to than handlers. This is why different non-terminals are used for actions and handlers.

- ⟨*Instance Handler*⟩ — ⟨*Instance Handler*⟩s are arbitrary code that is executed when a property handler is triggered.

The following constructs are based on the logic plugin(s) used in a particular specification.

- ⟨*Logic Name*⟩ — ⟨*Logic Name*⟩ is an identifier to indicate in which logic a property is defined.
- ⟨*Logic Syntax*⟩ — This refers to the syntax of the actual property definition, and is defined in the syntax section for each plugin.
- ⟨*Logic State*⟩ — ⟨*Logic State*⟩s are constants defined for each plugin. They state for which monitor states or categories (match, fail, etc.) a handler may be written.

2.3 Examples

In this section, we explain a few examples of JavaMOP specifications. More examples can be found in the JavaMOP website (`http://javamop.com`) and the project page of categorizing the Java API (`http://annotated-java-api.googlecode.com`).

2.3.1 Iterator_HasNext

Figure 2.4 shows a JavaMOP specification for the unsafe use of `Iterator` which is a simple data structure that iterates elements of the underlying data structure. This specification is also modified for demonstration purposes. While the original specification does not have any event action, it prints out the event details. Note that these event actions can be any code. `Iterator` has only three methods: hasNext() to check if there is any remaining element to iterate, next() to retrieve the element at the current cursor, and remove() to remove the element at the current cursor from the underlying data structure. So, it is recommended to call hasNext() before calling next() unless the size of the data structure is for sure. When the next() method is called and there is no element to return, the Java Virtual Machine (JVM) usually throws a runtime exception, but the exception is not guaranteed to be thrown in a multi-threaded environment.

This specification is parameterized by only one parameter ⟨*i*⟩, where *i* stands for `Iterator`. There are three parametric events defined in this specification: hasnexttrue ⟨*i*⟩, hasnextfalse ⟨*i*⟩, and next ⟨*i*⟩. Since this specification has only one parameter, Typestates can monitor this specification as well. In this specification, a new pointcut "condition(⟨*BooleanExpression*⟩)" is used and it is not in the standard AspectJ. This pointcut is an extended pointcut supported by JavaMOP, for checking a boolean expression. The difference between this and the pointcut "if(⟨*BooleanExpression*⟩)" from the standard AspectJ is that this

```
Iterator_HasNext(Iterator i) {
  event hasnexttrue after(Iterator i) returning(boolean b) :
    call(* Iterator+.hasNext())
    && target(i) && condition(b) {
    System.out.println("hasNext() returns true.");
  }

  event hasnextfalse after(Iterator i) returning(boolean b) :
    call(* Iterator+.hasNext())
    && target(i) && condition(!b) {
    System.out.println("hasNext() returns false.");
  }

  event next before(Iterator i) :
    call(* Iterator+.next())
    && target(i) {
    System.out.println("next() is called.");
  }

  ltl: [](next => (*) hasnexttrue)

  @violation {
    System.out.println("Iterator.hasNext() was not called before calling next().");
  }
}
```

Figure 2.4: Modified Iterator_HasNext specification in JavaMOP using the LTL plugin

extended pointcut can use user-defined monitor variables and even the return value of the method call that the event is defined for, which are not in the scope of the if pointcut. JavaMOP supports this extended pointcut by removing it in the resulting AspectJ code, and checking the condition at the beginning of the event so that the resulting AspectJ code only contains standard pointcuts.

2.3.2 Collection_UnsafeIterator

Figure 2.5 shows a JavaMOP specification for the unsafe use of `Collection` and `Iterator`. This specification can be found in [59], but it is modified in this section for demonstration purposes. In the original specification, there is no modifier and there is only one property in an extended regular expression (ERE). Here, we added two modifiers and another property which is the same to the original property but represented in a different logical formalism, linear temporal logic (LTL). The *decentralized* modifier is for using the decentralized indexing tree, which is faster than the centralized indexing tree but it might require instrumenting the Java library depending on the types of specification parameters. Since the first parameter in this specification is `Collection` from the Java API, it requires instrumentation of the Java library. The other modifier, *unsynchronized*, means that this specification does not require any synchronization between events, indicating that the program never generates events from different threads at the same time. For example, there can be only one thread or there can be already an external synchronization means.

```
decentralized unsynchronized Collection_UnsafeIterator(Collection c, Iterator i) {
  creation event create after(Collection c) returning(Iterator i) :
    call(Iterator Iterable+.iterator()) && target(c) {}

  event modify before(Collection c) :
    (
      call(* Collection+.add*(..)) ||
      call(* Collection+.clear(..)) ||
      call(* Collection+.offer*(..)) ||
      call(* Collection+.pop(..)) ||
      call(* Collection+.push(..)) ||
      call(* Collection+.remove*(..)) ||
      call(* Collection+.retain*(..))
    ) && target(c) {}

  event useiter before(Iterator i) :
    (
      call(* Iterator.hasNext(..)) ||
      call(* Iterator.next(..))
    ) && target(i) {}

  ere : create useiter* modify+ useiter

  @match {
    System.err.println("The collection was modified while an iterator is being used.");
  }

  ltl : [](useiter => (not modify S create))

  @violation {
    System.err.println("The collection was modified while an iterator is being used.");
  }

}
```

Figure 2.5: Modified Collection_UnsafeIterator specification in JavaMOP using the ERE plugin and LTL plugin

This specification is parameterized by those two parameters $\langle c, i \rangle$, where c stands for `Collection` and i stands for `Iterator`. Three parametric events are defined: create $\langle c, i \rangle$, modify $\langle c \rangle$, and useiter $\langle i \rangle$. Among events, the create event is the only event that can create monitors, which is called a creation event. All events on the same parameter instance before this event will be simply ignored. Since the create event creates `Iterator`, only possible event before this event is the modify event. Therefore, any modification on the `Collection` before the creation of `Iterator` is allowed and it will not create any monitor. Each event has a pointcut of AspectJ that picks out program points as events. For example, the create event has "call(Iterator Iterable+.iterator()) && target(c)" as its pointcut. Every call of iterator() to an object compatible to `Iterable` is considered to be a create event, and the target object is captured as the parameter c. Thus, the object should be also compatible to `Collection`.

The properties flag them as an error if an `Iterator` is created, its underlying `Collection` is modified, and then the `Iterator` is used again. In this specification, we use the extended regular expression (ERE) formalism and the linear temporal logic (LTL) formalism, as specified by the ere and ltl keywords, respectively. We wish to catch this behavior because Java `Collections` do not allow concurrent

```
PipedStream_SingleThread(PipedInputStream i, PipedOutputStream o, Thread t) {
  creation event create after(PipedOutputStream o) returning(PipedInputStream i) :
    call(PipedInputStream+.new(PipedOutputStream+)) && args(o) {}

  creation event create before(PipedInputStream i, PipedOutputStream o) :
    call(* PipedInputStream+.connect(PipedOutputStream+)) && target(i) && args(o) {}

  creation event create after(PipedInputStream i) returning(PipedOutputStream o) :
    call(PipedOutputStream+.new(PipedInputStream+)) && args(i) {}

  creation event create before(PipedOutputStream o, PipedInputStream i) :
    call(* PipedOutputStream+.connect(PipedInputStream+)) && target(o) && args(i) {}

  event write before(PipedOutputStream o, Thread t) :
    call(* PipedOutputStream+.write(..)) && target(o) && thread(t) {}

  event read before(PipedInputStream i, Thread t) :
    call(* PipedInputStream+.read(..)) && target(i) && thread(t) {}

  ere: create (write* | read*)

  @fail {
    System.err.println("A single thread attempted to use both a PipedInputStream instance and a
         PipedOutputStream instance, which may deadlock the thread.");
  }
}
```

Figure 2.6: PipedStream_SingleThread specification in JavaMOP using the ERE plugin

modification while iterating elements using Iterator. A runtime exception will be thrown when this occurs, but it is not guaranteed in a multi-threaded environment. When the properties are violated, it simply prints out an error message in this specification, but it can be any user-defined Java code.

2.3.3 PipedStream_SingleThread

Figure 2.6 shows a JavaMOP specification for the potentially unsafe use of PipedInputStream and PipedOutputStream in each Thread. According to the Java API documentation, once a PipedInputStream and a PipedOutputStream are connected, attempting to use both objects from a single thread is not recommended since it may lead to a deadlock. However, JVM does not detect this usage at all. The violation of this property may not mean an actual deadlock, but it can catch potential deadlocks, advising developers to review the code.

This specification is parameterized by three parameters $\langle i, o, t\rangle$, where i stands for PipedInputStream, o stands for PipedOutputStream, and t stands for Thread. There are three parametric events defined: create $\langle i, o\rangle$, write $\langle o, t\rangle$, and read $\langle i, t\rangle$. Unlike to the previous examples, this specification has four definitions of the creation event: one for each possible connection between two kinds of objects. They have different parameter mapping styles, resulting in four definitions. However, all four event definitions generate the same event, create; monitors will consider them as the same event. Among pointcuts in the event definitions, there is a new pointcut “thread(t)”, which is not in the standard AspectJ. This

```
Map_UnsafeIterator(Map m, Collection c, Iterator i) {
  creation event getset after(Map m) returning(Collection c) :
    (
      call(Set Map+.keySet()) ||
      call(Collection Map+.values())
    ) && target(m) {}

  event getiter after(Collection c) returning(Iterator i) :
    call(Iterator Iterable+.iterator()) && target(c) {}

  event modifyMap before(Map m) :
    (
      call(* Map+.clear*(..)) ||
      call(* Map+.put*(..)) ||
      call(* Map+.remove(..))
    ) && target(m) {}

  event modifyCol before(Collection c) :
    (
      call(* Collection+.clear(..)) ||
      call(* Collection+.offer*(..)) ||
      call(* Collection+.pop(..)) ||
      call(* Collection+.push(..)) ||
      call(* Collection+.remove*(..)) ||
      call(* Collection+.retain*(..))
    ) && target(c) {}

  event useiter before(Iterator i) :
    (
      call(* Iterator.hasNext(..)) ||
      call(* Iterator.next(..))
    ) && target(i) {}

  ere : getset (modifyMap | modifyCol)* getiter useiter* (modifyMap | modifyCol)+ useiter

  @match {
    System.err.println("The map was modified while an iteration over the set is in progress.");
  }
}
```

Figure 2.7: Map_UnsafeIterator specification in JavaMOP using the ERE plugin

pointcut is an extended pointcut supported by JavaMOP, for capturing the current thread of the event. JavaMOP will simply remove this pointcut in the resulting AspectJ code, and assign the current thread to the parameter when the event occurs.

Note that, in this specification, the creation event create does not initiate all parameters. The create event only initiates i and o, since the property is not concerned about which thread connects them. Therefore, the previous version of JavaMOP cannot monitor this specification.

2.3.4 Map_UnsafeIterator

Figure 2.7 shows a JavaMOP specification for the unsafe use of Map, Collection, and Iterator. This specification is similar to Collection_UnsafeIterator except that, in this specification, a Collection comes from a Map. A Collection is either the values or the key set of the Map. While iterating elements of the Collection, both the Map and the Collection should not be modified according to the Java API documentation. This causes an undefined behavior of the iterator and should be avoided.

This specification is parameterized by three parameters $\langle m, c, i \rangle$, where m stands for `Map`, c stands for `Collection`, and i stands for `Iterator`. There are five parametric events defined: getset $\langle m, c \rangle$, getiter $\langle c, i \rangle$, modifyMap $\langle m \rangle$, modifyCol $\langle c \rangle$, and useiter $\langle i \rangle$. This specification also has the creation event getset which does not initiate all parameters. It only initiates `Map` and `Collection`; again, the previous version of JavaMOP cannot monitor this specification.

2.4 Parametric Properties and Monitoring

In this section, we formally define parametric properties and parametric monitoring, starting with a high-level overview. The notations and terminologies introduced in this section are used consistently throughout the dissertation.

2.4.1 Overview

Monitoring a program execution generates an execution trace consisting of events that the user is interested in. An event can be any monitorable program activity such as a method call or field access/update. A property is a formula in a logical formalism where events are atomic formulas. It describes behaviors that the program should or should not follow. A specification contains event definitions, properties, event actions (Java codes to be executed upon events), and handlers (Java codes to be executed upon validation/violation). When an event occurs, the event action for the event will be executed, and when an execution trace validates/violates the given property, the appropriate handler will be executed.

A *parameter instance* is a mapping from parameters (e.g., classes in Java) to parameter values (e.g., objects in Java). For example, for the parameters $\langle c, i \rangle$, a parameter instance could be $\langle c \mapsto c_1, i \mapsto i_1 \rangle$, where c stands for `Collection`, i stands for `Iterator`, and c_1 and i_1 are instances of `Collection` and `Iterator`, respectively. A *event* is a program point which can be monitored (e.g., a method call or a field access). A *parametric event* is an event that comes with a parameter instance. For example, the event next of method call next() to `Iterator` can be parameterized by `Iterator`. Instead of next, we have next $\langle i \mapsto i_1 \rangle$, where i_1 is an instance of `Iterator`. A parametric property is a formula in a logical formalism where parametric events are atomic formulas.

A *parametric specification* is a specification whose events and properties are parameterized, along with specification parameters. *Specification parameters* are the free variables for the parametric specification. Therefore, we can describe behaviors of objects using parametric specifications. In parametric monitoring, an execution trace consisting of parametric events is generated. We slice the execution trace into trace slices according to parameter instances so that each parameter instance associates with one trace slice. We monitor the parametric specification for the trace slice of each parameter instance.

2.4.2 Definitions and Algorithms

We begin by introducing the notions of event, trace, and property, first non-parametric and then parametric. Trace slicing is then defined as a reduct operation that forgets events that are unrelated to the given parameter instance.

Definition 1. ***(Base events).*** *Let $\mathcal{E}$ be a finite set of (non-parametric) events, called* ***base events*** *or simply* ***events****. An* ***$\mathcal{E}$-trace****, or simply a (non-parametric)* ***trace*** *when $\mathcal{E}$ is understood or not important, is any finite sequence of events in $\mathcal{E}$, that is, an element in $\mathcal{E}^*$. If event $e \in \mathcal{E}$ appears in trace $w \in \mathcal{E}^*$ then we write $e \in w$. ϵ is the empty trace.*

For Collection_UnsafeIterator (referred to as Col_UnsafeIter) in Section 2.3, the set of events $\mathcal{E}$ is {create, modify, useiter}, and a possible trace is "create useiter modify useiter".

Definition 2. ***(Properties).*** *An* ***$\mathcal{E}$-property*** *P, or simply a (base or non-parametric)* ***property****, is a function $P{:}\mathcal{E}^* \to \mathcal{C}$ partitioning the set of traces into* ***(verdict) categories*** *$\mathcal{C}$. In general, $\mathcal{C}$ may be any set.*

Consider again Col_UnsafeIter. The match traces are those matching the pattern, e.g., "create useiter modify useiter". There are also traces that have not matched yet, but may still match in the future, such as "modify create", which we call ? (unknown) traces. Lastly, there are traces that may never match again, such as "create modify useiter useiter", which we refer to as fail traces. Thus we pick $\mathcal{C}$ to be the set $\{\mathsf{match}, \mathsf{fail}, ?\}$, and define its property $P_{\mathsf{Col\_}UnsafeIter}{:}\mathcal{E}^* \to \mathcal{C}$ as follows: $P_{\mathsf{Col\_}UnsafeIter}(w) = \mathsf{match}$ if w is in the language of the Col_UnsafeIter ere, $P_{\mathsf{Col\_}UnsafeIter}(w) = ?$ if w is a prefix of a string in the language of the ere, and $P_{\mathsf{Col\_}UnsafeIter}(w) = \mathsf{fail}$ otherwise.

We next extend the above definitions to the parametric case. Let $[A \to B]$ be the set of total functions, and let $[A \rightharpoonup B]$ be the set of partial functions from A to B.

Definition 3. ***(Parametric instances, events and traces).*** *Let X be a finite set of* ***parameters*** *and let V be a set of corresponding* ***parameter values****. Partial functions θ in $[X \rightharpoonup V]$ are called* ***parameter instances****. Let $\mathcal{E}$ be a set of base events, then $\mathcal{E}\langle X\rangle$ is the set of* ***parametric events*** *$e\langle\theta\rangle$, where e is a base event in $\mathcal{E}$ and θ is a parameter instance. A* ***parametric trace*** *is a trace with events in $\mathcal{E}\langle X\rangle$, that is, a word in $\mathcal{E}\langle X\rangle^*$.*

A parametric trace for Col_UnsafeIter could be "modify$\langle c \mapsto c_1\rangle$ modify$\langle c \mapsto c_2\rangle$ create$\langle c \mapsto c_1, i \mapsto i_1\rangle$ useiter$\langle i \mapsto i_1\rangle$". To simplify writing we often assume the parameter set implicit, as in the following, which is the same trace: "modify$\langle c_1\rangle$ modify$\langle c_2\rangle$ create$\langle c_1, i_1\rangle$ useiter$\langle i_1\rangle$".

Definition 4. ***(Parametric event definition).*** *Let X be a finite set of parameters. Given a set of base events $\mathcal{E}$, we define a* ***parametric event definition****, or* ***event definition*** *for short, as a function $\mathcal{D}{:}\,\mathcal{E} \to \mathcal{P}(X)$, where $\mathcal{P}$ is the power set, that maps each event e to a set of parameters $\mathcal{D}(e)$ that will be instantiated by e at runtime. $\mathcal{D}$ is extended to $\mathcal{E}^*$ as $\mathcal{D}(\epsilon) = \varnothing$ and $\mathcal{D}(ew) = \mathcal{D}(e) \cup \mathcal{D}(w)$, and to $\mathcal{P}(\mathcal{E})$ as $\mathcal{D}(\varnothing) = \varnothing$ and $\mathcal{D}(\{e\} \cup E) = \mathcal{D}(e) \cup \mathcal{D}(E)$. Parametric event $e\langle\theta\rangle$ is* ***$\mathcal{D}$-consistent*** *if $\mathsf{dom}(\theta) = \mathcal{D}(e)$. Parametric trace τ is* ***$\mathcal{D}$-consistent*** *if $e\langle\theta\rangle$ is $\mathcal{D}$-consistent for each $e\langle\theta\rangle \in \tau$.*

The Col_UnsafeIter property contains the parametric event definition $\mathcal{D}(\mathsf{create}) = \{c,\ i\}$, $\mathcal{D}(\mathsf{modify}) = \{c\}$, $\mathcal{D}(\mathsf{useiter}) = \{i\}$. It states that, for example, parameters c and i will be instantiated at runtime when a parametric event $\mathsf{create}\langle\theta\rangle$ is received. For a trace "create modify", $\mathcal{D}(\mathsf{create\ modify})$ is $\{c, i\}$.

Definition 5. ***(Compatibility).*** *$\theta, \theta' \in [A \rightharpoonup B]$ are* ***compatible*** *if for any $x \in \mathsf{dom}(\theta) \cap \mathsf{dom}(\theta')$, $\theta(x) = \theta'(x)$. We can* ***combine*** *compatible instances θ and θ', written $\theta \sqcup \theta'$, as follows:*

$$(\theta \sqcup \theta')(x) = \begin{cases} \theta(x) & \text{if } \theta(x) \text{ is defined} \\ \theta'(x) & \text{if } \theta'(x) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$\theta \sqcup \theta'$ is also called the ***least upper bound (lub)*** *of θ and θ'. θ is* ***less informative*** *than θ', written $\theta \sqsubseteq \theta'$, if for any $x \in X$, if $\theta(x)$ is defined then $\theta'(x)$ is also defined and $\theta(x) = \theta'(x)$. $\sqcup$ is extended to $\mathcal{P}_f([X \rightharpoonup V])$ in the natural way. Here $\mathcal{P}_f$ is the finite power set.*

Definition 6. ***(Trace slicing).*** *Given parametric trace $\tau \in \mathcal{E}\langle X\rangle^*$ and θ in $[X \rightharpoonup V]$, let the* ***θ-trace slice*** *$\tau{\restriction}_\theta \in \mathcal{E}^*$ be the non-parametric trace defined as:*

- *$\epsilon{\restriction}_\theta = \epsilon$ (recall that ϵ is the empty trace)*
- *$(\tau\, e\langle\theta'\rangle){\restriction}_\theta = \begin{cases} (\tau{\restriction}_\theta)\, e & \text{if } \theta' \sqsubseteq \theta \\ \tau{\restriction}_\theta & \text{otherwise} \end{cases}$*

The trace slice $\tau{\restriction}_\theta$ first filters out all the parametric events that are not relevant for the instance θ, i.e., which contain instances of parameters that θ does not care about, and then, for the remaining events relevant to θ, it forgets the parameters so that the trace can be checked against base, non-parametric properties. It is crucial to discard events from parameter instances that are not relevant to θ during the slicing, including those more informative than θ. Referring back to our parametric trace from above, the non-parametric trace slice for parameter instance $\langle c_2\rangle$ is "modify", that for $\langle c_1\rangle$ is "modify", the slice for $\langle c_1, i_1\rangle$ is "modify useiter", and the slice for $\langle i_1\rangle$ is "useiter".

Definition 7. ***(Parametric properties).*** *Let X be a finite set of parameters together with their corresponding parameter values V, and let $P{:}\,\mathcal{E}^* \to \mathcal{C}$ be a non-parametric property like in Definition 2. Then we define the* ***paramet-***

ric property *$\Lambda X.P$ as the property (over traces $\mathcal{E}\langle X\rangle^*$ and verdict categories $[[X \rightharpoonup V] \to \mathcal{C}]$)*

$$\Lambda X.P{:}\,\mathcal{E}\langle X\rangle^* \to [[X \rightharpoonup V] \to \mathcal{C}]$$

as $(\Lambda X.P)(\tau)(\theta) = P(\tau{\restriction_\theta})$ for each $\tau \in \mathcal{E}\langle X\rangle^*$, $\theta \in [X \rightharpoonup V]$.

A parametric property is therefore similar to a normal property, but one partitioning parametric traces in $\mathcal{E}\langle X\rangle^*$ into verdict categories in $[[X \rightharpoonup V] \to \mathcal{C}]$, that is, original (as in the non-parametric property) verdict categories indexed by parameter instances. This allows the parametric property to associate an original category for each parameter instance from $[X \rightharpoonup V]$.

Next we define monitors and parametric monitors. Like for parametric properties, which are just properties over parametric traces, parametric monitors are also just monitors, but for parametric events and with instance-indexed states and verdict categories.

Definition 8. ***(Monitors).*** *A* ***monitor*** *M is a tuple $(S, \mathcal{E}, \mathcal{C}, \imath, \sigma, \gamma)$, where S is the set of states, $\mathcal{E}$ is the set of input events, $\mathcal{C}$ is the set of verdict categories, $\imath \in S$ is the initial state, $\sigma{:}\, S \times \mathcal{E} \to S$ is the transition function, and $\gamma{:}\, S \to \mathcal{C}$ is the verdict function. The transition function is extended to handle traces, i.e., $\sigma{:}\, S \times \mathcal{E}^* \to S$ where $\sigma(s, \epsilon) = s$ and $\sigma(s, ew) = \sigma(\sigma(s, e), w)$. $M = (S, \mathcal{E}, \mathcal{C}, \imath, \sigma, \gamma)$ is a* ***monitor for property*** *$P{:}\,\mathcal{E}^* \to \mathcal{C}$ if $\gamma(\sigma(\imath, w)) = P(w)$ for each $w \in \mathcal{E}^*$. Monitor M defines the property $P_M{:}\,\mathcal{E}^* \to \mathcal{C}$ with $P_M(w) = \gamma(\sigma(\imath, w))$. Monitors M and M' are equivalent iff $P_M = P_{M'}$.*

We next define parametric monitors starting with a base monitor and a set of parameters: the corresponding parametric monitor can be thought of as a set of base monitors running in parallel, one for each parameter instance.

Definition 9. ***(Parametric monitors).*** *Given parameters X with corresponding values V and monitor $M = (S, \mathcal{E}, \mathcal{C}, \imath, \sigma, \gamma)$, the* ***parametric monitor*** *$\Lambda X.M$ is the monitor $([[X \rightharpoonup V] \to S], \mathcal{E}\langle X\rangle, [[X \rightharpoonup V] \to \mathcal{C}], \lambda\theta.\imath, \Lambda X.\sigma, \Lambda X.\gamma)$, with*

- $\Lambda X.\sigma{:}\, [[X \rightharpoonup V] \to S] \times \mathcal{E}\langle X\rangle \to [[X \rightharpoonup V] \to S]$
- $\Lambda X.\gamma{:}\, [[X \rightharpoonup V] \to S] \to [[X \rightharpoonup V] \to \mathcal{C}]$

defined as

$$(\Lambda X.\sigma)(\delta, e\langle\theta'\rangle)(\theta) = \begin{cases} \sigma(\delta(\theta), e) & \textit{if } \theta' \sqsubseteq \theta \\ \delta(\theta) & \textit{otherwise} \end{cases}$$
$$(\Lambda X.\gamma)(\delta)(\theta) = \gamma(\delta(\theta))$$

for each $\delta \in [[X \rightharpoonup V] \to S]$ and each $\theta, \theta' \in [X \rightharpoonup V]$.

Therefore, a parametric monitor $\Lambda X.M$ maintains a state $\delta(\theta)$ of M for each parameter instance θ, takes parametric events as input, and outputs categories

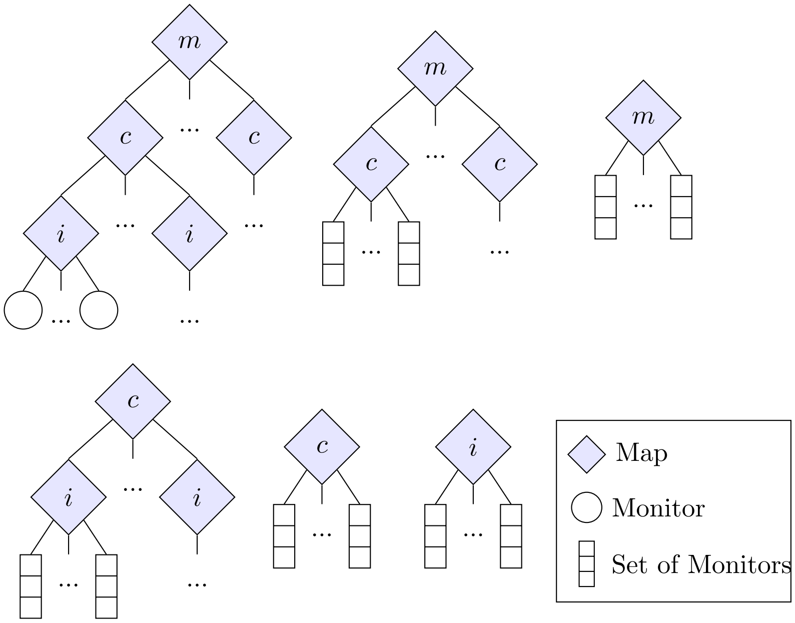


Figure 2.8: Indexing trees for Map_UnsafeIterator before combining

indexed by parameter instances (one category of M per instance). Intuitively, one can think of a parametric monitor as a collection of *"monitor instances"*. Each monitor instance, which is indexed by a parameter instance, keeps track of the state of one trace slice. The rule for $\Lambda X.\sigma$ can be read as stating that when an event with parameter instance θ' is evaluated, it updates the state for all monitor instances more informative than the instance for θ', and the instance for θ' itself, leaving all other monitor instances untouched. The rule for $\Lambda X.\gamma$ simply states that γ is applied to a state, as normal, but the state is found by looking up the state of the monitor instance for θ. One of the major results in [33] states that if M is a monitor for P, parametric monitor $\Lambda X.M$ is a monitor for the parametric property $\Lambda X.P$.

2.5 Indexing Tree

As shown in Section 2.4, JavaMOP slices the program execution trace for each parameter instance so that a monitor for each parameter instance can forget about the parameters and focus on the property. In this way, a monitor can be independent from parameters, resulting in a formalism-independent parametric monitoring system. For slicing the program execution trace, the monitoring code needs to dispatch each event to the related monitors for the parameter instance of the event.

The indexing tree is an efficient means to locate the monitors for a given parameter instance. The indexing tree is implemented as a multi-level map that, at each level, indexes each parameter object of the parameter instance. For example, Figure 2.8 shows indexing trees for the Map_UnsafeIterator specification from Section 2.3.4. The indexing tree for ⟨`Map`, `Collection`, `Iterator`⟩ (top-left tree) is a 3-level map. With a map, a collection, and an iterator, we can retrieve the related monitor. The indexing tree for ⟨`Map`, `Collection`⟩ (top-middle tree) is a 2-level map. For a map and a collection, this indexing tree returns a set of monitors because there can be multiple monitors for the given map and collection (one monitor for each iterator).

If an indexing tree stores all parameter objects directly, it will block them from being garbage collected, leading to a memory leak. Instead of storing parameter objects directly, the indexing tree uses the `WeakReference` class from the Java API. `WeakReference` is a reference to an object that will not disallow garbage collection for said object. When the object is garbage collected, the JVM changes the referent field of all weak references setting it to `null`. In effect, parameter objects can be garbage collected without any interference from monitoring.

Mappings in the indexing tree can be broken when parameter objects are garbage collected and their weak references point to `null`. The Java API provides a way to queue weak references of garbage collected objects into a `ReferenceQueue` object. By using this feature, broken mappings can be easily removed from the indexing tree. However, using this feature also slows down the system significantly because queuing weak references involves synchronization.

The previous version of JavaMOP was using `ReferenceIdentityMap`, one of the general data structures from the Apache Commons Collections Library [42]. While `ReferenceIdentityMap` is based on the `WeakReference`, allowing monitors to be garbage collected when they can be, `ReferenceIdentityMap` uses `ReferenceQueue` for cleaning up broken mapping, which shows a performance degradation as explained. Our new indexing tree implementation in Chapter 4 does not use `ReferenceQueue` for these performance reasons. Instead of using `ReferenceQueue`, we iterate through mappings and remove the broken ones. Surprisingly, iterating through mappings is significantly faster than using the queuing feature from the Java API. This self-cleaning feature of our indexing tree also allows for efficient garbage collection of unnecessary monitors in Chapter 4; when we iterate through the mappings, we simply check whether any of the monitors have become unnecessary. By doing this, we can reduce memory usage and unnecessary updates to those collected monitors.

Chapter 3

Expressive Parametric Monitoring

There is no doubt that a parametric monitoring system becomes more practical if it can express more properties and monitor them. The expressiveness of a parametric monitoring system is determined not only by logical formalisms that it supports, but also by how it supports parameters, what it can do upon an event and a match of the property, what kinds of events it can capture, and so fourth. There are many challenges in supporting multiple formalisms without any limitation on parameters, yet keeping its efficiency and freedom in the user-given code to be executed upon an event and a match of the property. As a feasible solution to this, many parametric monitoring systems hardwire their logical formalism, restrict parameters, or give up efficiency and/or freedom in the user-given code.

The previous JavaMOP supported multiple logical formalisms and it was extensible so that any logical formalism can be easily introduced. However, there was an restriction in expressing parametric properties that the creation events must initiate all parameters. In this chapter, we expand the expressiveness of JavaMOP by removing the restriction in expressing parametric properties, adding a new logical formalism, and introducing inheritance into specifications.

3.1 Non-Restrictive Parametric Monitoring

In this Section, we introduce a generic, in terms of specification formalism, solution to monitoring parametric specifications without any limitation on parameters. Our solution is based on a general, theoretical solution for handling parametric traces [33]. We implement this algorithm with the *enable set* optimization using static knowledge about the desired property. While the *enable set* optimization can be found in Chapter 4, this section discusses the generic parametric monitoring algorithm and some examples of real specifications from the Java API documentation, that the previous JavaMOP was not able to express/monitor, but the new JavaMOP can.

3.1.1 Approach Outline

The Map_UnsafeIterator specification from Chapter 2 expresses a property for the unsafe use of `Map`, `Collection`, and `Iterator`. `Map` and `Collection` implement data structures for mappings and collections, respectively. `Iterator` is an interface used to enumerate elements in a collection-typed object. One can also enumerate elements in a `Map` object using `Iterator`. But, since a `Map` object contains key-value pairs, one needs to first obtain a collection object that represents the contents of the map, e.g., the set of keys or the set of values stored in the map, and then create an iterator from the obtained collection. An intricate safety property in this usage, according to the Java API documentation, is that when the iterator is used to enumerate elements in the map, the contents of the map should not be changed, or unexpected behaviors may occur. A *violating* behavior with regards to this property can be naturally specified using future time linear temporal logic (FTLTL) with parameters. Given that m, c, i are objects of `Map`, `Collection` and `Iterator`, respectively:

$$\forall m, c, i.\, \diamond\left(\mathsf{getset}\langle m, c\rangle \wedge \diamond(\mathsf{getiter}\langle c, i\rangle\right.$$

$$\left.\wedge \diamond\left((\mathsf{modifyMap}\langle m\rangle \vee \mathsf{modifyCol}\langle c\rangle) \wedge \diamond\mathsf{useiter}\langle i\rangle\right)\right))$$

Where getset is creating a collection from a map, getiter is creating an iterator from a collection, modifyMap is updating the map, modifyCol is updating the collection, and useiter is using the iterator; $\diamond$ means eventually in the future.

The formula describes the following sequence of actions: `Collection` c is obtained from a `Map` m, an iterator i is created from c, m and/or c are changed, and then i is accessed. When an observed execution satisfies this formula, the Map_UnsafeIterator property is broken. The violating behavior can also be specified as an extended regular expression (ERE) that is more understandable for programmers but less concise:

$$\forall m, c, i.\mathsf{getset}\langle m, c\rangle\ \left(\mathsf{modifyMap}\langle m\rangle|\mathsf{modifyCol}\langle c\rangle\right)^*\ \mathsf{getiter}\langle c, i\rangle$$

$$\mathsf{useiter}\langle i\rangle^*\ \left(\mathsf{modifyMap}\langle m\rangle|\mathsf{modifyCol}\langle c\rangle\right)^+\ \mathsf{useiter}\langle i\rangle$$

Note that, the first event in both formulas, getset, which creates a monitor for the parameter instance, initiates only part of the parameters, e.g., $\langle m_1, c_1\rangle$ that binds parameters m and c with a map object m_1 and a collection c_1, respectively. Then, we create a monitor for the parameter instance $\langle m_1, c_1\rangle$. When the next event, $\mathsf{modifyMap}\langle c_1, i_1\rangle$ comes, obviously, we create a monitor for $\langle c_1, i_1\rangle$. Also, we need to create another monitor for $\langle m_1, c_1, i_1\rangle$ and this monitor should be updated with both events; it can be done by copying the monitor for $\langle m_1, c_1\rangle$ and updating the copied monitor with the second event. We need all three monitors because there might be another upcoming event that expands the existing parameter instance; then, we need to copy the monitor for the parameter instance that the event expands.

It becomes more complex, when the second event is $\mathsf{useiter}\langle i_2\rangle$. Without the knowledge that there is only one underlying `Collection` for an `Iterator`, which

#	Event	#	Event
1	getset$\langle m_1, c_1\rangle$	7	modifyMap$\langle m_1\rangle$
2	getset$\langle m_1, c_2\rangle$	8	useiter$\langle i_2\rangle$
3	getiter$\langle c_1, i_1\rangle$	9	getset$\langle m_2, c_3\rangle$
4	getiter$\langle c_1, i_2\rangle$	10	getiter$\langle c_3, i_4\rangle$
5	useiter$\langle i_1\rangle$	11	useiter$\langle i_4\rangle$
6	getiter$\langle c_2, i_3\rangle$		

Table 3.1: Possible execution trace over for Map_UnsafeIterator

is not available in general, we do not know, in advance, whether i_2 is going to expand the parameter instance $\langle m_1, c_1\rangle$. Therefore, we need to just expand it into $\langle m_1, c_1, i_2\rangle$ just in case, which is actually unnecessary in monitoring Map_UnsafeIterator. This can generate a huge number of unnecessary monitors in an actual monitoring, making it prohibitive to use. It is highly non-trivial to monitor such parametric specifications efficiently. This is why the previous JavaMOP only allows the first event that initiates all parameters.

Our approach to monitoring parametric traces against parametric properties is based on the observation that each parametric trace actually contains multiple *non-parametric trace slices*, each for a particular parameter binding instance. Intuitively, a slice of a parametric trace for a particular parameter binding consists of names of all the events that have *less informative* parameter bindings. Consider the example parametric trace of eleven events in Table 3.1 over the events from Map_UnsafeIterator. The # column gives the numbering of the events for easy reference. Every event in the trace starts with the name of the event, e.g., getset, followed by the parameter binding information, e.g., $\langle m_1, c_1\rangle$. Then, Table 3.2 shows the trace slices and their corresponding parameter bindings contained in the trace in Table 3.1. The Status column denotes the output category that the slice falls into (for ERE). In this case, the slice for $\langle m_1, c_1, i_2\rangle$, which matches the property, is in the "match" category, the slices for $\langle m_1, c_1, i_4\rangle$, $\langle m_1, c_2, i_1\rangle$, $\langle m_1, c_2, i_2\rangle$, and $\langle m_1, c_2, i_4\rangle$ are in the "fail" category, and all others are in "?" (undecided) category. Note that all events before the creation event getset, are ignored. The trace for the binding $\langle m_1, c_1\rangle$ contains getset modifyMap (the first and seventh events in the trace) and the trace for the binding $\langle m_1, c_1, i_2\rangle$ is getset getiter modifyMap useiter (the first, fourth, seventh, and eighth events in the trace).

Based on this observation, our approach creates a set of monitor instances during the monitoring process, each handling a trace slice for a parameter binding. Figure 3.1 shows the set of monitors created for the trace in Table 3.1, each monitor labeled by the corresponding parameter binding. This way, the monitor *does not need to handle the parameter information* and can employ any existing technique for ordinary, non-parametric traces, including state machines and push-down automata, providing a formalism-independent way to check parametric properties. When an event comes, our algorithm will dispatch it to related monitors, which will update their states accordingly. For example, the seventh

Instance	Slice	Status
$\langle m_1 \rangle$	modifyMap	?
$\langle i_1 \rangle$	useiter	?
$\langle i_2 \rangle$	useiter	?
$\langle i_4 \rangle$	useiter	?
$\langle m_1, c_1 \rangle$	getset modifyMap	?
$\langle m_1, c_2 \rangle$	getset modifyMap	?
$\langle m_1, i_1 \rangle$	useiter modifyMap	?
$\langle m_1, i_2 \rangle$	modifyMap useiter	?
$\langle m_1, i_4 \rangle$	modifyMap useiter	?
$\langle m_2, c_3 \rangle$	getset	?
$\langle c_1, i_1 \rangle$	getiter useiter	?
$\langle c_1, i_2 \rangle$	getiter useiter	?
$\langle c_2, i_3 \rangle$	getiter	?
$\langle c_3, i_4 \rangle$	getiter useiter	?
$\langle m_1, c_1, i_1 \rangle$	getset getiter useiter modifyMap	?
$\langle m_1, c_1, i_2 \rangle$	getset getiter modifyMap useiter	match
$\langle m_1, c_1, i_4 \rangle$	getset modifyMap useiter	fail
$\langle m_1, c_2, i_1 \rangle$	getset useiter modifyMap	fail
$\langle m_1, c_2, i_2 \rangle$	getset modifyMap useiter	fail
$\langle m_1, c_2, i_3 \rangle$	getset getiter modifyMap	?
$\langle m_1, c_2, i_4 \rangle$	getset modifyMap useiter	fail
$\langle m_1, c_3, i_4 \rangle$	modifyMap getiter useiter	?
$\langle m_2, c_3, i_1 \rangle$	useiter getset	?
$\langle m_2, c_3, i_2 \rangle$	useiter getset	?
$\langle m_2, c_3, i_4 \rangle$	getset getiter useiter	?

Table 3.2: Slices for the trace in Table 3.1

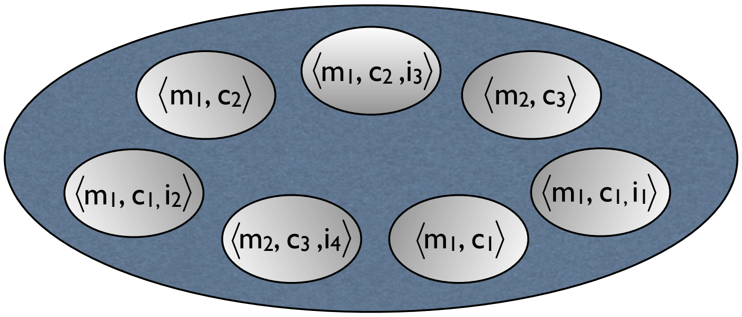


Figure 3.1: Monitors for the trace in Table 3.1

```
Algorithm M$onitor(M = (S, \mathcal{E}, \mathcal{C}, \imath, \sigma, \gamma))$
function main($\tau$)
$\Delta \leftarrow \perp$; $\Delta(\perp) \leftarrow \imath$; $\Theta \leftarrow \{\perp\}$
foreach $e\langle\theta\rangle$ $in\ order\ in$ $\tau$ do
: foreach $\theta' \in \{\theta\} \sqcup \Theta$ do
: : $\Delta(\theta') \leftarrow \sigma(\Delta(\max\{\theta'' \in \Theta \mid \theta'' \sqsubseteq \theta'\}), e)$
: : $\Gamma(\theta') \leftarrow \gamma(\Delta(\theta'))$
: endfor
: $\Theta \leftarrow \{\perp, \theta\} \sqcup \Theta$
endfor
```

Figure 3.2: Generic Parametric Monitoring algorithm

event in Table 3.1, modifyMap$\langle m_1 \rangle$, will be dispatched to monitors for $\langle m_1, c_1 \rangle$, $\langle m_1, c_2 \rangle$, $\langle m_1, c_1, i_1 \rangle$, $\langle m_1, c_1, i_2 \rangle$, and $\langle m_1, c_2, i_3 \rangle$. New monitor instances will be created if the event contains new parameter instances. For example, when the third event in Table 3.1, getiter$\langle c_1, i_1 \rangle$, is received, a new monitor will be created for $\langle m_1, c_1, i_1 \rangle$ by combining $\langle m_1, c_1 \rangle$ in the first event with $\langle c_1, i_1 \rangle$.

An algorithm to build parameter instances from observed events, like the one introduced in [33], often creates many useless monitor instances leading to prohibitive runtime overheads. For example, Table 3.2 does not need to contain the binding $\langle m_1, c_3, i_4 \rangle$ even though it can be created by combining the parameter instances of modifyMap$\langle m_1 \rangle$ (the seventh event) and getiter$\langle c_3, i_4 \rangle$ (the tenth event). It is safe to ignore this binding here because m_1 is not the underlying map for c_3, i_4. It is critical to minimize the number of monitor instances created during monitoring. The advantage is twofold: (1) it reduces the needed memory space, and (2), more importantly, monitoring efficiency is improved because fewer monitors are triggered for each received event. In Chapter 4, the *enable set* optimization is discussed for reducing the number of monitor instances.

3.1.2 Generic Parametric Monitoring Algorithm

Figure 3.2 shows the basic abstract monitoring algorithm for parametric properties from [33]. Given parametric property $\Lambda X.P$ and M a monitor for P, M$onitor(M)$ yields a monitor that is equivalent to $\Lambda X.M$, that is, a monitor for $\Lambda X.P$. The functions $[[X \rightharpoonup V] \to S]$ and $[[X \rightharpoonup V] \to \mathcal{C}]$ of $\Lambda X.M$ are encoded by M$onitor(M)$ as tables Δ and Γ with entries indexed by parameter instances in $[X \rightharpoonup V]$ and with contents states in S and verdict categories in $\mathcal{C}$, respectively. Such tables will have finite entries because each event e binds only a finite number of parameters defined by $\mathcal{D}(e)$.

1. Begin with a monitor instance for the empty parameter instance $\perp$ initialized to the start state of the monitor, $\imath$.

2. As each event, $e\langle\theta\rangle$, arrives there are two possibilities:

   - There is already a monitor instance for θ. In this case the instance is simply updated with e.
   - There is not already a monitor instance for θ. In this case an instance is created for θ. It is initialized to the state of the *most informative* θ' less informative than θ. Such a θ is guaranteed to exist because we begin with a monitor instance for $\perp$, which is less informative than all other possible θ's. We also create monitor instances for every parameter instance that may be created by combining θ with previously seen parameter instances. Each of these created instances is initialized similarly to the instance for θ, using the most informative instance less than itself. All created monitor instances are updated with e after initialization.

3. e is then used to update the monitor instances for all θ' that are strictly more informative than θ.

The monitoring algorithm first clears Δ, which contains the monitor state for each parameter instance, then assigns $\imath$, the initial state, to $\Delta(\perp)$. Θ, which contains all known parameter instances, is initialized to contain only the empty partial function $\perp$. For each event $e\langle\theta\rangle$ that arrives during program execution (line 2), M$onitor(M)$ generates every compatible parameter instance by combining θ with all the previously known compatible parameter instances (line 3). It then updates the state of every one of these compatible parameter instances (θ') with the state, transitioned by event e, of the "monitor instance" corresponding to the "largest" parameter instance less than or equal to θ' (line 4). At the same time we also calculate the verdict category corresponding to that monitor instance and store it in table Γ (line 5). Rather than storing a whole slice as in Definition 6 in Chapter 2, the knowledge of the slice is encoded in the state of the monitor instance for θ'. After the algorithm completes, Γ contains the verdict category for each possible trace slice. An actual implementation is free to report a verdict category of interest (e.g., match or fail) as soon as it is discovered.

3.2 Past-Time Linear Temporal Logic with Calls and Returns

Past time linear temporal logic with calls and returns (PTCaRet) [69] is a specialization of CaRet [10], an extension of LTL with calls and returns, for safety properties and their monitoring. Essentially, PTCaRet is PTLTL extended by adding abstract variants of temporal operators. Matching call/return events in traces allows one to express program trace properties not expressible using plain LTL. One can express properties related to the contents of the program execution stack, such as "function g is always called from within function f", or one can

⟨*PTCaRet Name*⟩	::=	“*ptcaret*”
⟨*PTCaRet Syntax*⟩	::=	“*true*” | “*false*” | ⟨*Event Name*⟩
	|	⟨*Unary Operator*⟩ ⟨*PTCaRet Syntax*⟩
	|	⟨*PTCaRet Syntax*⟩ ⟨*Binary Operator*⟩ ⟨*PTCaRet Syntax*⟩
⟨*Unary Operator*⟩	::=	“*not*” | “[∗]” | “<∗>” | “(∗)”
	|	“[∗*a*]” | “<∗*a*>” | “(∗*a*)”
	|	“@*b*” | “@*c*”
	|	“[∗*s*@*b*]” | “[∗*s*@*c*]” | “[∗*s*@*bc*]”
	|	“<∗*s*@*b*>” | “<∗*s*@*c*>” | “<∗*s*@*bc*>”
⟨*Binary Operator*⟩	::=	“*and*” | “*or*” | “*implies*” | “S” | “Sa”
	|	“*Ss*@*b*” | “*Ss*@*c*” | “*Ss*@*bc*”
⟨*PTCaRet State*⟩	::=	“*validation*” | “*violation*”

Figure 3.3: PTCaRet syntax

express properties that are allowed to be temporarily validated/violated, such as “a user u may never directly access a password file (but may access it through system procedures)” [69]. Motivated by practical reasons, PTCaRet distinguishes call and return points from begin and end points: the former take place in the method caller’s context and the latter take place in the callee’s context. This distinction allows more flexible and elegant expressions of properties.

3.2.1 Syntax

Figure 3.3 shows the syntax for our PTCaRet plugin. PTCaRet includes all operators from PTLTL: the standard boolean operators and the temporal operators. PTCaRet also has *abstract* temporal operators. The semantics of abstract operators is defined exactly as the semantics of their concrete counterpart operators, but they operate on the abstract version of the trace from which all the intermediate events of terminated method or function executions deeper in the call stack are erased [69]. In other words, abstract operators refer only to the trace of the current call stack level. In the syntax of the *abstract* temporal operators, “∗” and “S” are followed by “a”, meaning that the operator is an abstract variant of the concrete counterpart operator. The operators “[∗a]”, “<∗a>”, “(∗a)”, and “Sa” stand for “abstract always in the past”, “abstract eventually in the past”, “abstract previously in the past”, and “abstract since”, respectively.

PTCaRet also includes several derived operators which are convenient in practice, both for temporal and for stack operators. The operators “@b” and “@c”, read “at begin” and “at call” respectively, are derived temporal operators meaning that the formula they take as an argument must hold “at the Beginning of the execution of the current function” and “at the context when the current function was Called”, respectively. The semantics of the derived stack operators are defined exactly as the semantics of their *abstract* counterpart operators, but they operate only on the begin/call points on the abstract version of the trace. For example, derived stack operators defined on “begin” operate on a trace where

```
enter_phase_2 implies (
        not (not enter_phase_1 Sa begin)
        and (not acquire Sa enter_phase_1
             or not(not release Sa acquire))
        and @c (has_phase_2_pass)
        and <*s@b>(safe_exec))
```

Figure 3.4: PTCaRet example

we have filtered out all events except events in “begin” contexts from the abstract trace. Similar to the abstract temporal operators, in the syntax of the derived stack operators, “*” and “S” are followed by “s”, meaning that the operator is a derived stack variant of the concrete counterpart operator. In addition to this keyword, either of “@b”, “@c”, or “@bc” follows right after “s”, to indicate that the derived stack operator is defined on “begins”, “calls”, or “both begins and calls”, respectively. In particular, the operators “[*s@b]”, “<*s@b>”, and “Ss@b” are the derived stack operators on the beginnings of method calls, meaning “always on begin contexts on abstract traces”, “eventually on begin contexts on abstract traces”, and “since on begin contexts on abstract traces”, respectively. The derived stack operators for “calls” and “begins” are defined similarly.

3.2.2 Example

Figure 3.4 shows an example PTCaRet property from [69], which states that a program carrying out a critical multi-phase task should satisfy the following safety properties when execution enters the second phase:

- Execution entered the first phase in the same procedure;
- Resources acquired within the same procedure since the first phase must be released;
- The caller of the current procedure must have had approval for the second phase;
- Task is executed directly or indirectly by the procedure safe_exec.

Since the operators “Sa”, “@c”, and “<*s@b>” are abstract temporal operators, the example abstracts out events that happened in the procedure calls from within the current procedure.

3.2.3 Monitoring Algorithm

The monitor synthesis algorithm presented in [69], generalizes the synthesis algorithm from plain PTLTL formulae, found in [65]. To summarize it, the PTLTL synthesis algorithm uses a bitvector to keep the state of each temporal

```
if begin then {
  push(beta)
  exit
}
if end then {
  pop(beta)
  exit
}
beta[0] ← beta[0] or begin and safe_exec
beta[1] ← enter_phase_1 or not acquire
           and beta[1]
beta[2] ← acquire or not release and beta[2]
beta[3] ← begin or beta[3] and
           (not alpha[3] or alpha[2])
beta[4] ← begin or not enter_phase_1 and beta[4]
output(not enter_phase_2 or not beta[4]
        and beta[0]
        and (begin or beta[3])
        and (not begin or alpha[0])
        and (not beta[2] or beta[1])))
alpha[3] ← begin
alpha[2] ← alpha[1]
alpha[1] ← has_phase_2_pass
alpha[0] ← has_phase_2_pass
```

Figure 3.5: PTCaRet example output

operator in the formula. A series of sequential assignments updates the bitvector as each event arrives. If one of the operands to a temporal formula is itself a temporal formula, it will appear as a bitvector index in the assignment. It is, then, essential to generate the assignments in the proper order (depth-first).

The difference in generating monitors from PTCaRet formulae is that two bitvectors are kept for PTCaRet monitors, alpha[] and beta[]. The former plays the same role as the bitvector b from PTLTL. The other bitvector, beta[], stores the validity status of the subformulae corresponding to abstract temporal operators. When a new function or method is called, a copy of the abstract bitvector is pushed onto the top of a stack. When the function or method ends, the bitvector is popped from the stack, effectively erasing all updates that happened during the called function or method.

Figure 3.5 shows the output for Figure 3.4. PTCaRet uses the sequential assignments as explained above. In this example, we use the bitvector names and roles from previously, alpha[] and beta[]. Note that all elements in alpha[] and beta[] are initialized to false. When a new function or method is called, a copy of beta[] is pushed onto the top of a stack and when the function or method returns, the bitvector is popped from the stack, replacing beta[], while alpha[] stays as it is. Updating bitvectors before output is related to "since" operators, processing inner "since" operators before the outer ones. Updating bitvectors after `output` is related to “previously” operators, processing outer “previously” operators before the inner ones. Thus, bitvector updates before `output` are in order and bitvector updates after `output` are in reverse order. More detailed monitor synthesis algorithm can be found in [69].

3.2.4 Optimization

Although the monitor synthesis algorithm is already presented in [69], it is not easy to efficiently monitor properties in PTCaRet because, at each method return, we need to update states of all monitors. This is so that they forget about all the intermediate events of terminated method. A naive approach that keeps a stack of states and updates the stack at each method call/return will easily cause a huge overhead simply because there can be a huge number of method calls/returns. Static analysis to remove unnecessary stack maintenance at calls/returns of methods (that do not contain any event) can greatly improve monitoring PTCaRet in many cases. However, it is still important to improve the performance without help of static analysis because static analysis might not be available for some monitored programs, or more importantly, we cannot expect much improvement from static analysis when many methods are involved in the events. Also, our optimization technique is orthogonal to static analysis, so both of them can be used together for even more improvements.

Monitoring a PTCaRet formula involves pushing and popping of the bitvector beta[] at each method call and return. We observe that not all push and pop operations are necessary in monitoring it. For example, if a method is called and returned and there was no event during the method execution, then beta[] will be pushed into the stack and popped without any change; we can ignore this method, doing no operation on the stack.

Our static analysis can remove those unnecessary method begin and end events. If a method does not contain any point that can generate an event, begin and end of the method will be just ignored. If a method contains such a point, then we do stack operations on begin and end of the method although the method might not generate any event after all since an event point is not guaranteed to be reached. However, this static analysis requires one more step in generating the monitoring code for the given specification. Also, the resulting monitoring code is program specific so that it cannot be re-used for monitoring other programs.

We also implement a dynamic optimization which does not require any static analysis. The main idea is to maintain the stack of states as lazy as possible so that unnecessary maintenance can be offset by accumulated calls and returns. At each call and return, instead of updating stacks of all monitors, we maintain one global vector to represent how many times the executed program visits each depth of the call stack. By comparing the global vector with each monitor’s own vector, we can maintain the stack of states for each monitor upon every event.

Event	Call Depth	Global Version Vector	Monitor Version Vector
begin	1	(1)	
begin	2	(1, 1)	
end	2	(1, 1)	
begin	2	(1, 2)	
enter_phase_1	3	(1, 2)	
end	2	(1, 2)	(1, 2)
begin	2	(1, 3)	(1, 2)
begin	3	(1, 3, 1)	(1, 2)
end	3	(1, 3, 1)	(1, 2)
begin	3	(1, 3, 2)	(1, 2)
end	3	(1, 3, 2)	(1, 2)
begin	3	(1, 3, 3)	(1, 2)
acquire	4	(1, 3, 3)	(1, 2)
release	4	(1, 3, 3)	(1, 3, 3)
end	3	(1, 3, 3)	(1, 3, 3)
end	2	(1, 3, 3)	(1, 3, 3)
end	1	(1, 3, 3)	(1, 3, 3)

Table 3.3: An example trace of the PTCaRet property in Figure 3.4

For example, Table 3.3 shows an example trace of the PTCaRet property in Figure 3.4, with the changes in call depth and version vector. When the first event except begin and end events, enter_phase_1 comes, we compare the version vectors. Since two levels are new, we push twice. Then, at the event acquire, again we compare the version vectors. There is one common prefix, one different verion, and one new version. Therefore, we pop once to reach the depth of the common prefix, and push twice to reach the current depth. At the event release, there is nothing to push or pop. In total, we have five push or pop operations, while there are 14 push or pop operations without optimizations. In a real monitoring, there are likely to be more method calls without actual events; this optimization works more effectively.

This algorithm improves the performance of monitoring PTCaRet immensely because we can cancel out method calls and returns if there is no event between them, and we do not have to maintain stacks for monitors that are not used afterwards. Note that this optimization is orthogonal to static analysis, implying that it is possible to have further improvements by using static analysis based on this optimization algorithm.

3.3 Specification Inheritance

There are many specifications which share the same events, especially when expressing multiple properties on the same object in Java. Thus, it is natural to reuse specifications. Specification inheritance allows one to write a specification based on existing ones by adding/overriding/disabling events, properties, and

```
Improved_Map_UnsafeIterator(Map m, Collection c, Iterator i) includes Map_UnsafeIterator {
  event useiter before(Iterator i) :
    (
      call(* Iterator.hasNext(..)) ||
      call(* Iterator.next(..))
      call(* Iterator.remove(..))
    ) && target(i) {
      System.err.println("iterator has been used.");
    }

  ere Map_UnsafeIterator.prop : getset+ (modifyMap | modifyCol)* getiter useiter* (modifyMap |
       modifyCol)+ useiter

  @Map_UnsafeIterator.prop@match {
    System.err.println("The map was modified while an iteration over the set is in progress at:
         " + __LOC);
  }
}
```

Figure 3.6: Improved_Map_UnsafeIterator specification inheriting Map_UnsafeIterator in Chapter 2

handlers to them. In addition to specification inheritance, we support a way to capture an event or a handler in other specification. This allows meta monitoring that observes other monitoring and takes actions based on that. Our work on specification inheritance not only makes reuse of specification possible, but also allows one to describe more complex behaviors related to other specifications that we monitor at the same time.

Currently, JavaMOP supports primitive level of specification inheritance, that is, a sub-specification should contain all parameters from the parent specifications and there is no access control that Java has in its class inheritance. Due to the similarity between Java, AspectJ and the JavaMOP Framework, many features and concepts can be adopted into JavaMOP for more benefits. We leave this as a future work.

3.3.1 Example

Figure 3.6 shows an example of inherited specification for explaining how the specification inheritance can be used and demonstrating the syntax. This specification inherits Map_UnsafeIterator to improve and change the behavior of the original specification. In the original specification, the useiter event does not capture remove() method calls as iterator usages, thus the improved specification overrides the useiter to capture this method call as well. Also, it prints out a logging message for each useiter event, while the original specification does nothing. The property is also modified so that it allows multiple getset events at the beginning. When the property matches, it additionally prints out the source code location of the last event. In this way, modified specifications can be easily achieved without changing the original specifications.

3.3.2 Syntax

Figure 3.7 shows the syntax for the prototype of specification inheritance in JavaMOP. Note that modified or new constructs are highlighted. A specification can include parent specifications if they have the same parameters. It means that all the events, the properties and the handlers in the parent specifications are included into the current specification. Then, new syntax allows a user to modify them. The following syntax constructs are modified or added:

- ⟨*Parents*⟩ — ⟨*Parents*⟩ describes a list of specification names that this specification includes. The included specifications must have the same parameter type pattern with the current specification.

- ⟨*Event Modifier*⟩ — ⟨*Event Modifier*⟩ changes the behavior of the event. Only creation events can create monitors, starting monitoring, and abstract events are only for expanding it further in child specifications.

- ⟨*Property Handler*⟩ — ⟨*Property Handler*⟩ now can indicate the specification and property names so that it can modify an existing one.

- ⟨*JavaMOP Pointcut*⟩ — Two new pointcuts are added to this construct. handler pointcut catches an execution of the specified handler of the specified specification and property. If specification and property names are not given, it indicates the current specification and its only property by default. When there are multiple properties in the current specification, then the property name must be given.

⟨*JavaMOP Specification*⟩	::=	{⟨*JavaMOP Modifier*⟩} ⟨*Id*⟩ ⟨*JavaMOP Parameters*⟩
		[⟨***Parents***⟩]
		“{”
		{⟨*JavaMOP Declaration*⟩}
		{⟨*Event*⟩}
		{⟨*Property*⟩
		{⟨*Property Handler*⟩}
		}
		“}”
⟨***Parents***⟩	::=	**“includes”** ⟨***Id***⟩ [⟨***JavaMOP ParamList***⟩]
		{**“,”** ⟨***Id***⟩ [⟨***JavaMOP ParamList***⟩]}
⟨*Event*⟩	::=	{⟨***Event Modifier***⟩} “*event*” ⟨***Event Id***⟩
		⟨*JavaMOP Event Def*⟩ “{”
		⟨*JavaMOP Action*⟩
		“}”
⟨***Event Modifier***⟩	::=	**“creation”** | **“abstract”**
⟨*Property*⟩	::=	⟨*Logic Name*⟩ [⟨***Super Id***⟩] “ : ” ⟨*Logic Syntax*⟩
⟨*Property Handler*⟩	::=	[**“@”** ⟨***Super Id***⟩] “@” ⟨*Logic State*⟩
		⟨*JavaMOP Handler*⟩
⟨***Super Id***⟩	::=	[⟨***Spec Id***⟩ **“.”**] ⟨***Property Id***⟩
⟨***Spec Id***⟩	::=	⟨***Id***⟩
⟨***Property Id***⟩	::=	⟨***Id***⟩
⟨***Event Id***⟩	::=	⟨***Id***⟩
⟨*JavaMOP Modifier*⟩	::=	“*full–binding*” | “*maximal–binding*”
	|	“*any–binding*” | “*connected*” | “*unsynchronized*”
	|	“*decentralized*” | “*perthread*” | “*suffix*”
⟨*JavaMOP Parameters*⟩	::=	“(” [⟨*JavaMOP Type*⟩ ⟨*Id*⟩
		{“,” ⟨*JavaMOP Type*⟩ ⟨*Id*⟩}] “)”
⟨*JavaMOP Declaration*⟩	::=	syntax of declarations in Java
⟨***JavaMOP ParamList***⟩	::=	“(” [⟨***Id***⟩ {**“,”** ⟨***Id***⟩}] “)”
⟨*JavaMOP Event Def*⟩	::=	⟨*AspectJ AdviceSpec*⟩ “ : ”
		⟨*AspectJ Pointcut*⟩ [“&&” ⟨*JavaMOP Pointcut*⟩]
⟨*JavaMOP Action*⟩	:=	Java statements, which may refer to monitor local variables
⟨*JavaMOP Handler*⟩	:=	Java statements with additional keywords
⟨*JavaMOP Type*⟩	:=	Any valid Java type
⟨*AspectJ AdviceSpec*⟩	::=	syntax of AdviceSpec in AspectJ
⟨*AspectJ Pointcut*⟩	::=	syntax of Pointcut in AspectJ
⟨*JavaMOP Pointcut*⟩	::=	“*thread*” “(” ⟨*Id*⟩ “)”
	|	“*condition*” “(” ⟨*Boolean Exp*⟩ “)”
	|	“*endProgram*” “(” “)”
	|	“*endObject*” “(” ⟨*Id*⟩ “)”
	|	“*endThread*” “(” “)”
	|	***“handler”*** **“(”** [⟨***Super Id***⟩] **“@”** ⟨***Logic State***⟩ **“)”**
	|	***“event”*** **“(”** [⟨***Spec Id***⟩ **“.”**] ⟨***Event Id***⟩ **“)”**
	|	⟨*AspectJ Pointcut*⟩
	|	⟨*JavaMOP Pointcut*⟩ “&&” ⟨*JavaMOP Pointcut*⟩
⟨*Boolean Exp*⟩	::=	⟨*Id*⟩ | “!” ⟨*Boolean Exp*⟩
	|	⟨*Boolean Exp*⟩ ⟨*Boolean Operator*⟩ ⟨*Boolean Exp*⟩
	|	“(” ⟨*Boolean Exp*⟩ “)”
⟨*Boolean Operator*⟩	::=	“ || ” | “&&” | “ | ” | “&” | “ == ” | “! = ”

Figure 3.7: Specification inheritance syntax (newly introduced syntax is highlighted)

Chapter 4

Efficient Parametric Monitoring

Our work on efficiency of monitoring has resulted in a runtime monitoring framework that is the most efficient in terms of runtime overhead and competitive with respect to memory usage. Section 4.1 discusses the enable set optimization for formalism-independent parametric monitoring without any limitation. Section 4.2 introduces indexing cache to reduce the number of expensive operations that need to be performed on the indexing tree. Section 4.3 presents the monitor garbage collection that efficiently collects monitors which become unnecessary during monitoring. Finally, Section 4.4 evaluate how those techniques improve performance of parametric monitoring in terms of runtime and memory.

Our efficient parametric monitoring techniques presented in this chapter are orthogonal to other optimization techniques. More precisely, our techniques are aimed at improving the base performance of parametric monitoring by means of keeping the number of monitor instances low *without* relying on (expensive) knowledge about the source program. Other optimizations can be applied on top of our techniques and thus start from this base performance and improve it. For example, staged indexing (or decentralized indexing), which has been proposed and implemented in [19, 32, 16], piggy-backs indexing trees onto parameter instances. Also, significant runtime overhead reductions have been achieved using program static analysis [62, 29, 28, 41], by removing unnecessary instrumentation. JavaMOP supports both staged indexing and program static analysis via the Clara approach [29]. Nevertheless, we deliberately disabled these orthogonal optimizations in our evaluation, to properly measure the effectiveness of the proposed techniques. Enabling these orthogonal optimizations would only hide the inefficiency of base monitoring.

4.1 Efficient Formalism-Independent Monitoring of Parametric Properties

The generic parametric monitoring, found in Chapter 3, can monitor parametric properties without any limitation on parameters. However, there is a challenge in this approach that a huge number of monitor instances can be created during monitoring. For example, if the property is about `Collection` and `Iterator` in Java, there are as many parameter instances as the number of combinations of `Collection` and `Iterator`. It is not uncommon to see hundreds and thousands of iterators being used during one execution of a program, which would lead to hundreds and thousands of parameter instances in a specification about those iterators. However, not all combinations of them need to be monitored. Each `Iterator` has its own underlying `Collection`, therefore only related combinations should be monitored. To minimize the created monitor instances by using this observation, we use static knowledge about the desired property. By ignoring parameter instances that can never reach the target states, we can reduce the number of monitor instances to create, reducing runtime and memory overhead greatly.

It is worth mentioning that one may reduce the number of needed monitors using static program analysis, e.g., the one introduced in [28]. However, such techniques are based on the program targeted for monitoring, leading to two drawbacks: (1) it is a more complex and thus slower analysis and (2) the analysis must be run for every target program, making the approach non-modular. For example, if the property to monitor is related to some library, one will have to run the analysis for every program using the library, which can be expensive, and often infeasible. The analysis needed by our approach, on the other hand, is usually much quicker[1], because properties tend to be much smaller than the programs they are designed to monitor. Moreover, our optimization technique requires no additional analysis when used in a situation, like for a library, where a property is checked for different programs, because the enable set is derived only from the property.

In Section 4.1.1, enable sets are formally defined with intuitive explanations and examples. In Section 4.1.2, the algorithms to compute enable sets for finite-state machine (FSM) and context free grammars (CFG) are given. Then, Section 4.1.3 presents the parametric monitoring algorithm with enable sets and Section 4.1.4 discusses other possible optimizations.

4.1.1 Enable Sets

An enable set is constructed for each event, say e, defined for a particular property. The enable set associated with e is a set of sets of parameters. Each of these sets of parameters denotes parameters that must have been seen before the arrival of event e, for e to be acceptable by a monitor instance. Consider the event modifyMap from the Map_UnsafeIterator specification in Chapter 2 and the example parametric trace in Table 3.1 in Chapter 3, it may occur anywhere in a matching trace, *except* for as the first event. Because the first event must be getset in a matching trace, and because getset instantiates both m and c, one of the sets in the enable set for modifyMap must be $\{m, c\}$. However, modifyMap may (in fact, must, to match the pattern) occur after the getiter event.

[1] The analysis is bounded above by the number of acyclic paths from the start state/symbol through a finite state machine/context free grammar, because convergence is achieved through one cycle. Finite state machines and context free grammars for properties tend to be small.

Because getiter may not occur before getset, we also have the set $\{m, c, i\}$ in the enable set for modifyMap. The final result for the enable set for modifyMap is thus: $\{\{m, c\}, \{m, c, i\}\}$. Therefore, when modifyMap$\langle m_1 \rangle$ arrives (seventh event in Table 3.1), the instance monitors for $\langle m_1, c_1 \rangle$ and $\langle m_1, c_2 \rangle$ must be updated because they bind $\{m, c\}$, and the instance monitors for $\langle m_1, c_1, i_1 \rangle$, $\langle m_1, c_1, i_2 \rangle$, and $\langle m_1, c_2, i_3 \rangle$ must be updated because they bind $\{m, c, i\}$, and have the same value for m (m_1). In this example all of the instances to update have already been created by the time the event arrives, while no new instances can be created because at least m and c must be bound before modifyMap can occur.

When monitoring a program against a specific property, usually only a certain subset of property categories, ($\mathcal{C}$ in *Definition 2*), is checked. For example, in the Map_UnsafeIterator specification in Chapter 2, the regular expression specifies a defective interaction among related `Map`, `Collection` and `Iterator` objects. To find an error in the program using monitoring is thus to detect matches of the specified pattern during the execution. In other words, we are only interested in the validation category of the specified pattern. Obviously, to match the pattern, for a parameter instance of parameter set $\{m, c, i\}$, getset and getiter should be observed before useiter is encountered for the first time in monitoring. Otherwise, the trace slice for $\{m, c, i\}$ will never match the pattern. Based on this information, we next show that creating the monitor state for $\langle m_1, c_2, i_1 \rangle$ in Table 3.2 in Chapter 3 is not needed. When event useiter$\langle i_1 \rangle$ is encountered, if the monitor state for a parameter instance $\langle m_1, c_2 \rangle$ exists without the monitor state for $\langle m_1, c_2, i_1 \rangle$, it can be inferred that in the trace slice for $\langle m_1, c_2, i_1 \rangle$, only events getset and/or modifyMap occur before useiter because, otherwise, if getiter also occurred before useiter, the monitor state for $\langle m_1, c_2, i_1 \rangle$ should have been created. Therefore, we can infer, when event useiter$\langle i_1 \rangle$ is observed and before the execution continues, that no match of the specified pattern can be reached by the trace slice for $\langle m_1, c_2, i_1 \rangle$, that is to say, the monitor for $\langle m_1, c_2, i_1 \rangle$ will never reach the match state.

This observation shows that the knowledge about the specified property can be applied to avoid unnecessary creation of monitor states. We next formalize the information needed for the optimization and argue that it is not specific to the underlying specification formalism. How this information is used is discussed in Section 4.1.3.

Definition 10. Given $\tau \in \mathcal{E}^*$ and $e, e' \in \tau$, we denote that e' occurs before an occurrence of e in τ as $e' \rightsquigarrow_\tau e$. Let the **trace enable set** of $e \in \mathcal{E}$ be the function $\mathsf{enable}_\tau : \mathcal{E} \to \mathcal{P}_f(\mathcal{E})$, defined as: $\mathsf{enable}_\tau(e) = \{e' \mid e' \rightsquigarrow_\tau e\}$.

Note that if $e \notin \tau$ then $\mathsf{enable}_\tau(e) = \varnothing$. The trace enable set can be used to examine whether the execution under observation may generate a particular trace of interest, or not: if event e is encountered during monitoring but some event $e' \in \mathsf{enable}_\tau(e)$ has not been observed, then the (incomplete) execution being monitored will *not* produce the trace τ when it finishes. This observation

Event	$\mathsf{enable}_{\mathcal{G}}^{\mathcal{E}}$	$\mathsf{enable}_{\mathcal{G}}^{X}$
getset	$\{\varnothing\}$	$\{\varnothing\}$
getiter	{{getset}, {getset, modifyMap}, {getset, modifyCol}, {getset, modifyMap, modifyCol}}	$\{\{m, c\}\}$
modifyMap	{{getset}, {getset, modifyCol}, {getset, getiter}, {getset, modifyCol, getiter}, {getset, getiter, useiter}, {getset, modifyCol, getiter, useiter}}	$\{\{m, c\},$ $\{m, c, i\}\}$
modifyCol	{{getset}, {getset, modifyMap}, {getset, getiter}, {getset, modifyMap, getiter}, {getset, getiter, useiter}, {getset, modifyMap, getiter, useiter}}	$\{\{m, c\},$ $\{m, c, i\}\}$
useiter	{{getset, getiter}, {getset, getiter, modifyCol}, {getset, getiter, modifyMap}, {getset, getiter, modifyMap, modifyCol}}	$\{\{m, c, i\}\}$

Table 4.1: Property and parameter enable sets for Map_UnsafeIterator

can be extended to check, before an execution finishes, whether the execution can generate a trace belonging to some designated property categories. The designated categories are called the *goal* of the monitoring.

Definition 11. Given $P : \mathcal{E}^* \to \mathcal{C}$ and a set of categories $\mathcal{G} \subseteq \mathcal{C}$ as the goal, the **property enable set** is defined as a function $\mathsf{enable}_{\mathcal{G}}^{\mathcal{E}} : \mathcal{E} \to \mathcal{P}_f(\mathcal{P}_f(\mathcal{E}))$ with $\mathsf{enable}_{\mathcal{G}}^{\mathcal{E}}(e) = \{\mathsf{enable}_{\tau}(e) \mid P(\tau) \in \mathcal{G}\}$.

Intuitively, if event e is encountered during monitoring but none of event sets $\mathsf{enable}_{\mathcal{G}}^{\mathcal{E}}(e)$ has been completely observed, the (incomplete) execution being monitoring will not produce a trace τ s.t. $P(\tau) \in \mathcal{G}$. For example, given the regular expression specifying the Map_UnsafeIterator specification, where $\mathcal{G}$ contains only the match, fail, and ? categories, the second column in Table 4.1 shows the property enable sets of events in Map_UnsafeIterator.

The property enable set provides a sound and fast way to decide whether an incomplete trace slice has the possibility of reaching the desired categories by looking at the events that have already occurred. In the above example, if a trace slice starts with getset useiter, it will never reach the match category, because getset $\notin$ $\mathsf{enable}_{\mathcal{G}}^{\mathcal{E}}(\mathsf{useiter})$. In such case, no monitor state need be created even when the newly observed event may lead to new parameter instances. For example, suppose that the observed (incomplete) trace is getset useiter from before. At the second event, useiter, a new parameter instance can be constructed, namely, $\langle m_1, c_1, i_1\rangle$, and a monitor state s will be created for $\langle m_1, c_1, i_1\rangle$ if the generic algorithm (Figure 3.2) is applied. However, since the trace slice for s is getset useiter, we immediately know that s cannot reach state match. So there is no need to create and maintain s during monitoring if match is the goal.

A direct application of the above idea to optimize the generic algorithm (Figure 3.2) requires maintaining observed events for every created monitor and comparing event sets when a new parameter instance is found, reducing the improvement of performance. Therefore, we extend the notion of the enable set to be based on parameter sets instead of event sets.

Definition 12. Given a property $P : \mathcal{E}^* \to \mathcal{C}$, a set of categories $\mathcal{G} \subseteq \mathcal{C}$ as the goal, a set of parameters X and a function $\mathcal{D} : \mathcal{E} \to \mathcal{P}_f(X)$ mapping an event to its parameters, the **property parameter enable set** of event $e \in \mathcal{E}$ is defined as a function $\mathsf{enable}_{\mathcal{G}}^{X} : \mathcal{E} \to \mathcal{P}_f(\mathcal{P}_f(X))$ as follows: $\mathsf{enable}_{\mathcal{G}}^{X}(e) = \{\cup\{\mathcal{D}(e') \mid e' \in \mathsf{enable}_{\tau}(e)\} \mid P(\tau) \in \mathcal{G}\}$.

From now on, we use "enable set" to refer to "property parameter enable set" for simplicity. For example, given the regular pattern for the Map_UnsafeIterator specification in Chapter 2 and $\mathcal{G} = \{\mathsf{match}\}$; the third column in Table 4.1 shows the parameter enable sets of events in Map_UnsafeIterator. Then, given again the trace $\mathsf{getset}\langle m_1, c_1\rangle\ \mathsf{useiter}\langle i_1\rangle$, no monitor state need be created at the second event for $\langle m_1, c_1, i_1\rangle$ since the parameter instance used to initialize the new monitor state, namely, $\langle m_1, c_1\rangle$, is not in $\mathsf{enable}_{\mathcal{G}}^{X}(\mathsf{useiter})$. In other words, one may simply compare the parameter instance used to initialize the new parameter instance with the enable set of the observed event to decide whether a new monitor state is needed or not. Note that in JavaMOP, the property parameter enable sets are generated from the property enable sets provided by the formalism plugin. This allows the plugins to remain totally parameter agnostic.

4.1.2 Computing Enable Sets

As we mentioned, the definition of the enable set is general and does not depend on a specific formalism to write the property. We next show two algorithms to compute enable sets for finite-state machine (FSM) based monitors and context-free grammars (CFG), respectively.

Case 1: FSM The algorithm in Figure 4.1 computes the property enable sets for a finite state machine. We use this algorithm to compute the enable sets for any logic that is reducible to a finite state machine, including ERE, FTLTL, and PTLTL (past time linear temporal logic). The algorithm assumes a finite state machine, defined as $FSM = (\mathcal{E}, S, s_0 \in S, \delta : S \times \mathcal{E} \overset{\circ}{\to} S)$. $\mathcal{E}$ is the alphabet, traditionally listed as Σ but changed for consistency, because the alphabets of our FSMs are event sets. s_0 is the start state, corresponding to $\imath$ in the definition of a monitor. δ is the transition partial function, taking a state and an event and potentially mapping to a next state for the machine. We assume that all states not reachable from the initial state and not coreachable from the states of interest (states of interest being those states s such that $\gamma(s) \in \mathcal{G}$) are pruned

```
Algorithm $\mathcal{EN}_{fsm}(FSM = (\mathcal{E}, S, s_0, \delta))$

Globals: mapping $\mathcal{V}_\mu : S \to \mathcal{P}_f(\mathcal{P}_f(\mathcal{E}))$
         mapping $\mathsf{enable}_{\mathcal{G}}^{\mathcal{E}} : \mathcal{E} \to \mathcal{P}_f(\mathcal{P}_f(\mathcal{E}))$

Initialization: $\mathcal{V}_\mu(s) \leftarrow \varnothing$ for any $s \in s$
                $\mathsf{enable}_{\mathcal{G}}^{\mathcal{E}}(e) \leftarrow \varnothing$ for any $e \in \mathcal{E}$

function main()
  1 compute_enables($s_0, \varnothing$)

function compute_enables($s, \mu$)
  1 foreach defined $\delta(s, e)$ do
  2 : $\mathsf{enable}_{\mathcal{G}}^{\mathcal{E}}(e) \leftarrow \mathsf{enable}_{\mathcal{G}}^{\mathcal{E}}(e) \cup \{\mu\}$
  3 : let $\mu' \leftarrow \mu \cup \{e\}$
  4 : if $\mu' \notin \mathcal{V}_\mu(s)$
  5 : : $\mathcal{V}_\mu(s) \leftarrow \mathcal{V}_\mu(s) \cup \{\mu'\}$
  6 : : compute_enables($\delta(s, e), \mu'$)
  7 : endif
  8 endfor
```

Figure 4.1: FSM $\mathsf{enable}_{\mathcal{G}}^{\mathcal{E}}$ computation algorithm.

from the FSM before running the algorithm, leaving the transitions that pointed to them undefined. $\mathcal{V}_\mu$ is a mapping from states to sets of events; it is used to check for algorithm termination. $\mathsf{enable}_{\mathcal{G}}^{\mathcal{E}}$ is the output property enable set, which is converted into a parameter enable set by JavaMOP.

Function compute_enables is first called from main with $\mu = \varnothing$ and the initial state s_0. If we think of the FSM as a graph, μ represents the set of edges we have seen at least once in a given traversal path. For each defined $\delta(s, e)$ (line 1), we add the current μ to the $\mathsf{enable}_{\mathcal{G}}^{\mathcal{E}}(e)$ (line 2) because this means we have seen a viable prefix set (as all non-viable paths in the machine have been pruned). This follows from the definition of $\mathsf{enable}_{\mathcal{G}}^{\mathcal{E}}$. Line 3 begins the recursive step of the algorithm. We let $\mu' = \mu \cup \{e\}$, because we have traversed another edge, and that edge is labeled as e. The map $\mathcal{V}_\mu$ tells us which μ have been seen in previous recursive steps, in a given state. If a μ has been seen before, in a state, taking a recursive step can add no new information. Because of this, line 4 ensures that we only call the recursive step on line 6, if new information can be added. Line 5 keeps $\mathcal{V}$ consistent. Thus the algorithm terminates only when every viable μ has been seen in every reachable state, effectively computing a fixed point. Thus, the algorithm is bounded above by the number of one cycle paths through the graph (and is faster in practice, because most paths will have repeated events, collapsing them into smaller μ's).

Event	$e_1\langle p_1\rangle$	$e_2\langle p_2\rangle$	$e_3\langle p_1, p_2\rangle$
Δ	$\langle p_1\rangle{:}\sigma(i, e_1)$	$\langle p_1\rangle{:}\sigma(i, e_1)$	$\langle p_1\rangle{:}\sigma(i, e_1)$ $\langle p_1, p_2\rangle{:}\sigma(\sigma(i, e_1), e_3)$

Table 4.2: Unsound usage of the enable set

Case 2: CFG We also provide an algorithm to compute the *match* enable set for a context-free pattern, which has an infinite monitor state space, as briefly explained in what follows[2]. This is a modification of the algorithm in Figure 4.1.

Let $\mathcal{G} = \{\mathsf{match}\}$. For $\mathsf{enable}_{\mathcal{G}}^{\mathcal{E}}$ and a given CFG $G = (NT, \mathcal{E}, P, S)$ we begin with all productions $S \to \gamma$ and the set $\mu_0 = \varnothing \in \mathcal{P}_f(\mathcal{E})$. For each production, we investigate each $s \in \gamma$ (where $\in$ is, by abuse of notation, used to denote a symbol in a right hand side) from left to right. If $s \in \mathcal{E}$ we add μ_i to $\mathsf{enable}_{\mathcal{G}}^{\mathcal{E}}(s)$, thus if s is the first symbol in γ we add μ_0. We then add s to μ_i forming μ_{i+1}. If $s \in NT$ we recursively invoke the algorithm, but rather than use μ_0, we use μ_i, and each production investigated will be of the form $s \to \gamma'$. We keep track of which $s \in NT$ have been processed, to ensure termination of the algorithm.

4.1.3 Monitoring with Enable Sets

We next integrate the concept of enable sets and creation events with the generic algorithm (Figure 3.2), to improve performance and memory usage. Given a set of desired value categories $\mathcal{G}$, we can optimize the monitoring process by omitting creating monitor states for certain parameter instances when an event is received using the enable set without missing any trace belonging to $\mathcal{G}$. However, skipping the creation of monitor states may result in false alarms, i.e., a trace that is not in $\mathcal{G}$ can be reported to belong to $\mathcal{G}$. Let us consider the following example. We monitor to find matching of a regular pattern $e_1 e_3$. Relevant events and their parameters are $e_1(p_1), e_2(p_2), e_3(p_1, p_2)$. The observed trace is $e_1\langle p_1\rangle e_2\langle p_2\rangle e_3\langle p_1, p_2\rangle$. Also, suppose e_1 is the only creation event. Obviously, the trace does not match the pattern. Figure 4.2 shows the run using the enable set optimization (i.e., not creating monitor states for parameter instances disallowed by the enable sets). At e_1, a monitor state is created for $\langle p_1\rangle$ since it is the creation event. At e_2, no action is taken since $\mathsf{enable}_{\mathcal{G}}^{X}(e_2) = \varnothing$. At e_3, a monitor state will be created for $\langle p_1, p_2\rangle$ using the monitor state for $\langle P_1 \mapsto p_1\rangle$ since $\mathsf{enable}_{\mathcal{G}}^{X} e_3 = \{P_1\}$. This way, e_2 is forgotten and a match of the pattern is reported incorrectly.

To avoid unsoundness, we introduce the notion of $\mathsf{disable}$ stamps of events. $\mathsf{disable} : [[X \rightharpoonup V] \rightharpoonup \mathsf{integer}]$ maps a parameter instance to an integer timestamp. $\mathsf{disable}(\theta)$ gives the time when the last event with θ was received. We maintain timestamps for monitors using a mapping $\mathcal{T} : [[X \rightharpoonup V] \rightharpoonup \mathsf{integer}]$. $\mathcal{T}$ maps a parameter instance for which a monitor state is defined to the time when the

[2] We assume a certain familiarity with context free patterns; definitions can be found in [63], together with explanations on CFG monitoring.

Event	$e_1\langle p_1\rangle$	$e_2\langle p_2\rangle$	$e_3\langle p_1, p_2\rangle$
Δ	$\langle p_1\rangle$:$\sigma(i, e_1)$	$\langle p_1\rangle$:$\sigma(i, e_1)$	$\langle p_1\rangle$:$\sigma(i, e_1)$
$\mathcal{T}$	$\langle p_1\rangle$:1	$\langle p_1\rangle$:1	$\langle p_1\rangle$:1
disable	$\langle p_1\rangle$:2	$\langle p_1\rangle$:2 $\langle p_2\rangle$:3	$\langle p_1\rangle$:2 $\langle p_2\rangle$:3 $\langle p_1, p_2\rangle$:4

Table 4.3: Sound monitoring using timestamps

original monitor state is created from a creation event. Specifically, if a monitor state for θ is created using the initial state when a creation event is received, $\mathcal{T}(\theta)$ is set to the time of creation; if a monitor state for θ is created from the monitor state for θ', $\mathcal{T}(\theta')$ is passed to $\mathcal{T}(\theta)$. Table 4.3 shows the evolution of disable and $\mathcal{T}$ while processing the trace in Table 4.2.

disable and $\mathcal{T}$ can be used together to track "skipped events": when a monitor state for θ is created using the monitor state for θ', if there exists some $\theta'' \sqsubset \theta$ s.t. $\theta'' \not\sqsubseteq \theta'$ and $\mathsf{disable}(\theta'') > \mathcal{T}(\theta')$ then the trace slice for θ does not belong to the desired value categories $\mathcal{G}$. Intuitively, $\mathsf{disable}(\theta'') > \mathcal{T}(\theta')$ implies that an event $e\langle\theta''\rangle$ has been encountered after the monitor state for θ' was created. But θ'' was not taken into account ($\theta'' \not\sqsubseteq \theta'$). The only possibility is that e is omitted due to the enable set and thus the trace slice for θ does not belong to $\mathcal{G}$ according to the definition of the enable set. Therefore, in Table 4.3, no monitor instance is created for $\langle p_1, p_2\rangle$ at e_3 because $\mathsf{disable}(\langle p_2\rangle) > \mathcal{T}(\langle p_1\rangle)$.

The above discussion applies when the skipped event occurs after the initial creation of the monitor state. The other case, i.e., an event is omitted before the initial monitor state is created, can also be handled using timestamps. If the skipped event is not a creation event, it does not affect the soundness of the algorithm because of the definition of creation events. In the above example, if the observed trace is $e_2\langle p_2\rangle e_1\langle p_1\rangle e_3\langle p_1, p_2\rangle$, we will ignore e_2 and report the matching at e_3 since e_1 is the only creation event. It is more sophisticated (but not much different) when the skipped event is a creation event.

Based on the above discussion, we develop a new parametric monitoring algorithm that optimizes the generic algorithm using the enable set and timestamps, as shown in Figure 4.2. This algorithm makes use of the mappings discussed above, namely, $\mathsf{enable}^X_{\mathcal{G}}$, Δ, disable and $\mathcal{T}$, and maintains an integer variable to track the timestamp. It also introduces $\mathcal{U}$ which maps a parameter instance θ to *all the parameter instances* that have been defined and are properly more informative than θ. When event $e\langle\theta\rangle$ is received, algorithm $\mathbb{D}\langle X\rangle$ first checks whether $\Delta(\theta)$ is defined or not (line 1 in main). If not, monitor states may be generated for new encountered parameter instances, which is achieved by function createNewMonitorStates in algorithm $\mathbb{D}\langle X\rangle$. Unlike in the generic algorithm, where all the parameter instances less informative than θ are searched to

```
Algorithm $\mathbb{D}\langle X\rangle(M = (S, \mathcal{E}, \mathcal{C}, \imath, \sigma, \gamma))$

Input: mapping $\mathsf{enable}_{\mathcal{G}}^{X} : [\mathcal{E} \to \mathcal{P}_f(\mathcal{P}_f(X))]$

Globals: mapping $\Delta : [[X \to V] \to S]$
         mapping $\mathcal{T} : [[X \to V] \to \text{integer}]$
         mapping $\mathcal{U} : [X \to V] \to \mathcal{P}_f([X \to V])$
         mapping disable : $[[X \to V] \to \text{integer}]$
         integer timestamp

Initialization: $\mathcal{U}(\theta) \leftarrow \emptyset$ for any $\theta$, timestamp $\leftarrow 0$

function main($e\langle\theta\rangle$)
  if $\Delta(\theta)$ undefined then
    createNewMonitorState($e\langle\theta\rangle$)
    if $\Delta(\theta)$ undefined and $e$ is a creation event then
      defineNew($\theta$)
    endif
    disable($\theta$) $\leftarrow$ timestamp; timestamp $\leftarrow$ timestamp $+ 1$
  endif
  foreach $\theta' \in \{\theta\} \cup \mathcal{U}(\theta)$ s.t. $\Delta(\theta')$ defined do
    $\Delta(\theta') \leftarrow \sigma(\Delta(\theta'), e)$
  endfor

function createNewMonitorStates($e\langle\theta\rangle$)
  foreach $X_e \in \mathsf{enable}_{\mathcal{G}}^{X}(e)$ (in reversed topological order) do
    if dom($\theta$) $\notin X_e$ then
      $\theta_m \leftarrow \theta'$ s.t. $\theta' \sqsubset \theta$ and dom($\theta'$) = dom($\theta$) $\cap X_e$
      foreach $\theta'' \in \mathcal{U}(\theta_m) \cup \{\theta_m\}$ s.t. dom($\theta''$) = $X_e$ do
        if $\Delta(\theta'')$ defined and $\Delta(\theta'' \sqcup \theta)$ undefined then
          defineTo($\theta'' \sqcup \theta, \theta''$)
        endif
      endfor
    endif
  endfor

function defineNew($\theta$)
  foreach $\theta'' \sqsubset \theta$ do
    if $\Delta(\theta'')$ defined then return endif
  endfor
  $\Delta(\theta) \leftarrow \imath$; $\mathcal{T}(\theta) \leftarrow$ timestamp; timestamp $\leftarrow$ timestamp $+ 1$
  foreach $\theta'' \sqsubset \theta$ do $\mathcal{U}(\theta'') \leftarrow \mathcal{U}(\theta'') \cup \{\theta\}$ endfor

function defineTo($\theta, \theta'$)
  foreach $\theta'' \sqsubseteq \theta$ s.t. $\theta'' \not\sqsubseteq \theta'$ do
    if disable($\theta''$) $> \mathcal{T}(\theta')$ or $\mathcal{T}(\theta'') < \mathcal{T}(\theta')$ then
      return
    endif
  endfor
  $\Delta(\theta) \leftarrow \Delta(\theta')$; $\mathcal{T}(\theta) \leftarrow \mathcal{T}(\theta')$
  foreach $\theta'' \sqsubset \theta$ do $\mathcal{U}(\theta'') \leftarrow \mathcal{U}(\theta'') \cup \{\theta\}$ endfor
```

Figure 4.2: Optimized monitoring algorithm $\mathbb{D}\langle X\rangle$.

find all the compatible parameter instances, $\mathsf{createNewMonitorStates}$ enumerates parameter sets in $\mathsf{enable}^X_{\mathcal{G}}(e)$ and looks for parameter instances whose domains are in $\mathsf{enable}^X_{\mathcal{G}}(e)$ and which are compatible with θ, using $\mathcal{U}$. The inclusion check at line 2 in $\mathsf{createNewMonitorStates}$ is to omit unnecessary search since if $\mathsf{dom}(\theta) \subseteq X_e$ then no new parameter instance will be created from θ. This way, $\mathsf{createNewMonitorStates}$ creates all the parameter instances from θ whenever the enable set of e is satisfied using fewer lists in $\mathcal{U}$.

If e is a creation event then a monitor state for θ is initialized (lines 3 - 5 in main). Note that $\Delta(\theta)$ can be defined in function $\mathsf{createNewMonitorStates}$ if $\Delta(\theta')$ has been defined for some $\theta' \sqsubset \theta$. $\mathsf{disable}(\theta)$ is set to the current timestamp after all the creations and the timestamp is increased (line 6 in main). Then, all the relevant monitor states are updated according to e. Function $\mathsf{defineNew}$ in $\mathbb{D}\langle X\rangle$ first searches for a defined sub-instance of θ. If such instance exists, θ should be defined using it; otherwise, $\Delta(\theta)$ is set to the initial state. Then $\mathcal{T}(\theta)$ is set to the current timestamp, and the timestamp is incremented. Function $\mathsf{defineTo}$ in $\mathbb{D}\langle X\rangle$ checks $\mathsf{disable}$ and $\mathcal{T}$ as discussed above to decide whether $\Delta(\theta)$ can be defined using $\Delta(\theta')$. If $\Delta(\theta)$ is defined using $\Delta(\theta')$, $\mathcal{T}(\theta)$ is set to $\mathcal{T}(\theta')$. Both functions then add θ to the sets in table $\mathcal{U}$ for the bindings less informative than θ.

4.1.4 Discussion

The general definition of the enable set allows us to separate the concerns of generating efficient monitoring code. On the framework level, such as the algorithms discussed in this dissertation, we can focus on applying the information encoded in the enable set to generate an efficient monitoring process for parametric properties, while on the logic level, where a monitor is generated for a given on-parametric property written in a specific formalism, one can focus on creating the fastest monitor that verifies the input trace against the property and also on producing the enable set information. The enable set represents static information about the given property and only need be generated once. As mentioned, the static analysis presented in [28], while effective, requires a complex analysis of the target program, which must be performed for every program one wants to monitor.

We discuss two other possibilities for optimization. The first is to make use of the semantics of the program. In the example of the Map_UnsafeIterator specification, we know that an i object is created from a c object and does not relate to other c objects. Hence, we can avoid creating a combination of $\langle m_2, c_2\rangle$ and $\langle i_1\rangle$ because i_1 is created from c_1. However, such semantic information is very difficult to achieve automatically and may require human input. The enable set, on the contrary, can be easily computed by statically analyzing the specification without analyzing any program or human interferences; indeed, the specified property already indicates some semantics of the involved parameters.

Nevertheless, we believe that static analysis on the program to monitor, such as that in [28], can and should be applied in conjunction with enable sets to further reduce the monitoring overhead, whenever it is feasible.

Other optimizations are based on heuristics. One reasonable heuristic which can be applied here is that we may only combine parameter instances that are connected to one another through some events which have been observed (we cannot rely on future events in online monitoring). For example, in the possible execution trace for Map_UnsafeIterator (Table 3.1), $\langle i_1 \rangle$ and $\langle m_1, c_1 \rangle$ need to be combined to build a new parameter instance because c_1 and i_1 are connected in the third event, getiter$\langle c_1, i_1 \rangle$, but $\langle i_1 \rangle$ and $\langle m_1, c_2 \rangle$ should not combined due to the heuristic. The intuition is that if two parameter instances do not interact in any event, it may imply that they are not relevant to each other even if they are compatible. However, because no information about future events available, such a heuristic can break, for example, an event connecting the two parameter instances comes afterward. The enable set provides a sound optimization, and we believe that it performs as well as, if not better than, such heuristics in most cases.

4.2 Indexing Caching

In our parametric monitoring approach that we keep a monitor for each parameter instance, we need to locate relevant monitors for each received event to update their states. As an efficient solution to this, many monitoring systems including ours use the indexing tree technique presented in Chapter 2. Indexing trees locate the relevant monitors by using multi-level mappings. Although indexing trees are fairly efficient – it is the most frequently used data structure in monitoring – it can cause a noticeable overhead in the presence of a large number of events. The major overhead in retrieving monitor(s) through an indexing tree comes from the cost of hashing. Hashing should happen at each level of the tree since an indexing tree consists of `HashMap`-based data structures. Thus, retrieving monitors through an indexing tree is much more expensive than a regular access to local variables or object fields.

To increase the performance of indexing trees even further, we introduce 'indexing caching' that caches the last retrieved monitor(s) at each indexing tree to serve them right away without any hashing cost. By doing this, we can utilize the temporal locality among monitors. For example, in monitoring Map_UnsafeIterator, after creating an `Iterator` from a `Collection`, the `Iterator` will likely to be used many times in succession, and thus we will process many events about `Iterator` usage. Many other programs and properties have events that occur as part of a long string of repeated events. Thus, if we cache the monitor(s) that need to be updated, we only need to pay the hashing cost once.

Once we have introduced a cache, we have an opportunity to tweak many parameters (e.g., number of entries, eviction schemes). In order to find reasonable

Cache Entries	1	2	3	4
Lookups	157011738	157875749	156297661	157047943
Cache Hits	150147057	152707467	151257225	152047105
Cache Hit Rate	95.63%	96.73%	96.78%	96.82%

Table 4.4: Cache hit rate when monitoring bloat against Map_UnsafeIterator

monitor(s) for these parameters, we tried monitoring a program bloat from the DaCapo benchmark suite [24] against UnsafeIter using different cache sizes. Table 4.4 shows our results when using a least recently used (LRU) eviction algorithm. Because we do not perform the search in parallel, the search time grows linearly with the size of the cache. More entries only offer marginal improvements to hit rate, if any, so we maintain a cache with only one element. With only one cache entry the question of which entry to evict becomes moot, it is unnecessary to investigate this further.

With this simple idea, we can increase the performance of indexing trees greatly. In our evaluation in Section 4.4, this is one of major factors for the performance improvements. This indexing cache technique is further improved in Chapter 5, for monitoring multiple simultaneous specifications.

4.3 Garbage Collection for Monitoring Parametric Properties

In parametric monitoring systems, the parameters are dynamically bound to objects at runtime, thus resulting in a potentially unlimited number of monitor instances, one for each parameter instance. The main challenge underlying the monitoring of parametric properties is therefore how to effectively manage these monitor instances, in particular how to efficiently retrieve all the monitor instances interested in an event when it takes place, and how to efficiently garbage collect monitor instances which have become unnecessary.

The previous version of JavaMOP was only able to collect monitor instances when all the bound parameters are garbage collected, which ensures that no event can happen to the corresponding monitor instance. The problem with this method of garbage collection can be clearly seen in the Map_UnsafeIterator specification in Chapter 2. Because it is the `next` event at the end of the pattern that actually causes the error, there is no way to ever match the pattern if the `Iterator` bound to a given monitor instance is garbage collected. However, the previous version of JavaMOP is only able to collect the associated monitor instance if all the three `Map`, `Collection` and the `Iterator` are garbage collected. Unfortunately, in most realistic programs, `Map` and `Collection` objects have much longer lifetimes than the `Iterator` objects created from them. Because of this, the previous version of JavaMOP would have large numbers of monitor instances

– when monitoring most programs – that could never possibly match the pattern because their bound `Iterators` had been collected.

In this section, we present the monitor garbage collection technique, a means to prune unnecessary monitor instances based on a static analysis of the monitored property. The results of the static analysis, which we refer to as coenable sets, are used at runtime to determine when a monitor instance can no longer reach a triggering state, and can thus be garbage collected. Although we know what monitor instances to collect, removing such unnecessary monitors is still an expensive task, and in the interest of making it as efficient as possible, a lazy garbage collection scheme is used. There are two performance benefits to garbage collecting unnecessary monitors: reduced memory usage, and reduction in the time needed to update monitor instances because many of the monitor instances that would be updated are no longer necessary.

Section 4.3.1 formally describes coenable sets, and Section 4.3.2 presents the efficient monitor garbage collection algorithm, based on coenable sets, that lazily propagates the information of garbage collections of parameter objects and lazily removes unnecessary monitors.

4.3.1 Coenable Sets

When monitoring parametric properties, it is easy to generate a large number of monitor instances. For example, as seen in Section 4.4, the program bloat generates 1.9 million monitor instances when monitored for the UnsafeIter specification. After some time, some of these monitor instances may become unnecessary, e.g., because they have no hope of reaching a verdict category in $\mathcal{G}$. Indeed, as seen in Section 4.4, the garbage collection technique flags 1.8 million of these monitor instances as unnecessary. In Section 4.1 proposed the enable set optimization, to avoid needlessly *creating* monitors that will never trigger. In this section, we show how a dual method can be derived to avoid needlessly *retaining* monitors that will never trigger. Computing the coenable sets is expected to be a quick static operation in practice, because they are a function of the specification to monitor (which is expected to be small) and not of the program (which is expected to be large).

Definition 13. *Given $w \in \mathcal{E}^*$ and $e, e' \in w$, we let $e \leadsto_w e'$ denote that e' occurs after e in w. Let $\textsc{Coenable}_w(e) = \{e' \mid e \leadsto_w e'\}$ be the **trace coenable set** of e. Given property $P\colon \mathcal{E}^* \to \mathcal{C}$ and a subset of verdict categories of interest (or goal) $\mathcal{G} \subseteq \mathcal{C}$, the **property coenable set** is defined as the map $\textsc{Coenable}_{P,\mathcal{G}}\colon \mathcal{E} \to \mathcal{P}(\mathcal{P}(\mathcal{E}))$ where $\textsc{Coenable}_{P,\mathcal{G}}(e) = \{\textsc{Coenable}_w(e) \mid w \in \mathcal{E}^* \text{ s.t. } P(w) \in \mathcal{G}, e \in w,\ \textsc{Coenable}_w(e) \neq \emptyset\}$ for each $e \in \mathcal{E}$.*

Intuitively, if event e is encountered during monitoring, but none of the event sets of $\textsc{Coenable}_{P,\mathcal{G}}(e)$ are possible in the future, it is impossible to reach any verdict category in $\mathcal{G}$, so a monitor for P observing e will never trigger. We drop all $\varnothing$s from $\textsc{Coenable}_{P,\mathcal{G}}$ because they can cause monitor instances to be retained that are unnecessary. An $\varnothing$ in $\textsc{Coenable}_{P,\mathcal{G}}(e)$ means that the trace suffix consisting of only the event e can lead to a category in $\mathcal{G}$ for some trace prefix. However, our interest is in the ability to reach $\mathcal{G}$ again in the future. If there is a trace suffix that can lead to a state in $\mathcal{G}$ from e, then its events will be added to $\textsc{Coenable}_{P,\mathcal{G}}(e)$. If there is no trace suffix that can lead back to a state in $\mathcal{G}$, there is no reason to maintain the monitor instance after it has executed the proper handler due to the occurrence of e.

FSM Example We define finite state machines in the spirit of Definition 8. A finite state machine is a tuple $(S, \mathcal{E}, \mathcal{C}, \imath, \sigma, \gamma)$ where $\mathcal{E}$ is a finite alphabet, S is a finite set of states, $\imath \in S$ is the initial state, $\sigma: S \times \mathcal{E} \rightharpoonup S$ a partial transition function, $\mathcal{C}$ a set of verdict categories, and $\gamma: S \to \mathcal{C}$ the verdict function. The property monitored by an FSM classifies a trace w into $\gamma(\sigma(\imath, w))$, where σ is extended to strings in the natural way, and fail if $\sigma(\imath, w)$ is undefined.

We can find $\textsc{Coenable}_{P,\mathcal{G}}$, for the property monitored by an FSM, by the least fixed point of the following equations. Recall that $\mathcal{G} \subseteq \mathcal{C}$ is the set of verdict categories of interest:

$$\textsc{Seeable}(s) = \bigcup_{\sigma(s,e)=s'} \{\{e\} \cup T \mid T \in \textsc{Seeable}(s')\}$$

$$\textsc{Coenable}_{P,\mathcal{G}}(e) = \bigcup_{\sigma(s,e)=s'} \textsc{Seeable}(s')$$

For the Collection_UnsafeIterator specification from Chapter 2, we can use the equations above to generate coenable sets; it requires generating a finite state machine from the property's ERE, which is simple enough. For $P = \mathsf{C}ollection\_UnsafeIterator$ and $\mathcal{G} = \{\mathsf{match}\}$, the $\textsc{Coenable}_{P,\mathcal{G}}$ sets are:

$$\textsc{Coenable}_{P,\mathcal{G}}(\mathsf{create}) = \{\{\mathsf{useiter}, \mathsf{modify}\}\}$$

$$\textsc{Coenable}_{P,\mathcal{G}}(\mathsf{modify}) = \{\{\mathsf{useiter}\}, \{\mathsf{useiter}, \mathsf{modify}\}\}$$

$$\textsc{Coenable}_{P,\mathcal{G}}(\mathsf{useiter}) = \{\{\mathsf{useiter}, \mathsf{modify}\}\}$$

Note that if we did not remove $\varnothing$s, $\textsc{Coenable}_{P,\mathcal{G}}(\mathsf{useiter})$ would contain $\varnothing$. Each inner set can be thought of as a conjunction of events that must occur at least once for a verdict category in $\mathcal{G}$ to still be reachable, while the outer sets are a disjunction (see Section 4.3.2). For example, if the event seen by monitor instance M is modify and useiter can still be seen at some future point, then M is still necessary. Likewise, if the event seen by M is useiter, then both useiter *and* modify must be possible for M to ever match. In particular, if the corresponding Collection object instance is already dead then we know that the event modify will never be possible, so we can safely garbage collect M. Definition 14 formalizes this notion.

CFG Example A CFG is a tuple $(N, \mathcal{E}, \mathsf{S}, \Pi)$ where N is a finite set of nonterminals, $\mathcal{E}$ is a finite set of terminals, $\mathsf{S} \in N$ is the initial nonterminal, and Π

is a set of productions of the form $\mathtt{A} \to \beta$ where $\mathtt{A} \in N$ and $\beta \in (N \cup \mathcal{E})^*$. The monitor for a CFG classifies traces that are in the language of the grammar into the verdict category match.

For a CFG, to compute $\textsc{Coenable}_{P,\{\mathsf{match}\}}$ we find the least fixed point of the following equations:

$$\mathrm{G}(\epsilon) = \{\varnothing\} \quad \mathrm{G}(e) = \{\{e\}\} \quad \mathrm{G}(\mathtt{A}) = \textstyle\bigcup_{\mathtt{A}\to\beta} \mathrm{G}(\beta)$$
$$\mathrm{G}(\beta_1\beta_2) = \{T_1 \cup T_2 \mid T_1 \in \mathrm{G}(\beta_1), T_2 \in \mathrm{G}(\beta_2)\}$$
$$C(x) = \left\{ T_1 \cup T_2 \;\middle|\; \begin{array}{l} \mathtt{A} \to \beta_1 x \beta_2, \\ T_1 \in C(\mathtt{A}), T_2 \in \mathrm{G}(\beta_2) \end{array} \right\}$$
$$\textsc{Coenable}_{P,\{\mathsf{match}\}}(e) = C(e)$$

Informally, $\mathrm{G}(\mathtt{A})$ is the set of events generated by the CFG, if the symbol $\mathtt{A}$ were used as the initial nonterminal of the CFG. The equation $\mathrm{G}(\beta_1\beta_2) = \{T_1 \cup T_2 \mid T_1 \in \mathrm{G}(\beta_1), T_2 \in \mathrm{G}(\beta_2)\}$ generalizes this notion to entire traces of symbols (where symbols are either events or non-terminals). C is the coenable sets function generalized to traces that include both non-terminals and events. For a production, $\mathtt{A} \to \beta_1\mathtt{B}\beta_2$, $C(\mathtt{B})$ needs to cope with the fact that $\mathtt{A}$ has its own coenable sets. Thus its definition unions possible coenable sets of $\mathtt{A}$ with the sets of symbols that are generated by β_2. The rest of JavaMOP only needs to know coenable sets for events so coenables is just the restriction of C to events.

Definition 14. *Given property $P{:}\,\mathcal{E}^* \to \mathcal{C}$, goal $\mathcal{G} \subseteq \mathcal{C}$, set of parameters X and event definition $\mathcal{D}{:}\,\mathcal{E} \to \mathcal{P}(X)$ (see Definition 4), the* ***property parameter coenable set*** *is defined as the map $\textsc{Coenable}^X_{P,\mathcal{G}}{:}\,\mathcal{E} \to \mathcal{P}(\mathcal{P}(X))$ where $\textsc{Coenable}^X_{P,\mathcal{G}}(e) = \{\mathcal{D}(E) \mid E \in \textsc{Coenable}_{P,\mathcal{G}}(e)\}$ for each $e \in \mathcal{E}$.*

The $\textsc{Coenable}^X_{P,\mathcal{G}}$ sets tell us which parameter objects must be alive for a verdict category in $\mathcal{G}$ to be reachable. For $P = \mathsf{C}ollection\_UnsafeIterator$, $\mathcal{G} = \{\mathsf{match}\}$, and $X = \{c, i\}$, the $\textsc{Coenable}^X_{P,\mathcal{G}}$ sets are:

$$\textsc{Coenable}^X_{P,\mathcal{G}}(\mathsf{create}) = \{\{c, i\}\}$$
$$\textsc{Coenable}^X_{P,\mathcal{G}}(\mathsf{modify}) = \{\{i\}, \{c, i\}\}$$
$$\textsc{Coenable}^X_{P,\mathcal{G}}(\mathsf{useiter}) = \{\{c, i\}\}$$

Now with the $\textsc{Coenable}^X_{P,\mathcal{G}}$ sets we can explicitly decide when a monitor instance may be collected. For example, in Collection_UnsafeIterator we know that if, at *any time*, the `Iterator` bound to i is garbage collected, then a match can never occur because i occurs in every one of the inner sets. This makes sense because the event that causes a match in the Collection_UnsafeIterator pattern is use of the `Iterator`. This situation could produce a very large memory leak in the previous version of JavaMOP where long living `Collections` would cause monitor instances for dead `Iterators` to be retained because it could not remove a monitor instance unless all bound parameter objects were collected. We prove this concept by showing that certain parameters specified by $\textsc{Coenable}^X_{P,\mathcal{G}}(e)$ for a trace wew' must be able to occur in w' for a verdict category to be reached.

Theorem 1. *Consider the same assumptions as in Definition 14, and a trace slice $wew' \in \mathcal{E}^*$. If for each $Y \in \textsc{Coenable}^X_{P,\mathcal{G}}(e)$ there exists some $y \in Y$ such that $y \notin \mathcal{D}(w')$ then $P(wew') \notin \mathcal{G}$.*

Proof. Suppose, for the sake of contradiction, that $P(wew') \in \mathcal{G}$ and that each $Y \in \textsc{Coenable}^X_{P,\mathcal{G}}(e)$ contains a y such that $y \notin \mathcal{D}(w')$. By Definition 13, because $P(wew') \in \mathcal{G}$ there must be some $E \in \textsc{Coenable}_{P,\mathcal{G}}(e)$ that contains exactly those events in w'. Then, by Definition 14, there must be $Y \in \textsc{Coenable}^X_{P,\mathcal{G}}(e)$ containing exactly the parameters in $\mathcal{D}(w')$. Contradiction. □

Discussion The $\textsc{Coenable}^X_{P,\mathcal{G}}$ sets are a conservative approximation of the situations in which a monitor instance may be collected. From Definition 6 we know that an event e where $x \in \mathcal{D}(e)$ can only occur in a trace-slice $\tau\upharpoonright_\theta$ if $\theta(x)$ is still alive in the system. If $\theta(x)$ has been garbage collected, there is no way for any e with $x \in \mathcal{D}(e)$ to occur in trace slice for θ. This is precisely how monitoring arrives in the situation presented in Theorem 1, where all possible suffixes w' of the trace slice wew' do not contain at least one parameter in each set of the $\textsc{Coenable}^X_{P,\mathcal{G}}(e)$, and it becomes impossible to reach a verdict category in $\mathcal{G}$. Clearly, if it is impossible for the θ trace slice to ever reach a verdict category in $\mathcal{G}$, there is no reason to keep the monitor instance for θ.

The Tracematches system uses a more precise formulation, which is similar, but based on the *state* of the monitor. Intuitively, the Tracematches garbage collection technique can be thought of as coenables sets indexed by state rather than events, but the formulation as presented in [19] is considerably different. While theirs is more precise, our empirical results, presented in Section 4.4, show that the coenable set technique is able to reduce memory usage in the JavaMOP framework to comparable levels with Tracematches, while the JavaMOP framework has considerably lower runtime overhead. More importantly, the Tracematches garbage collection technique is limited to finite logics, such as the regular expressions of Tracematches. However, our coenable approach is extensible to any underlying monitor implementation. We have a coenables sets generation algorithm for the context-free grammar plugin. A static state-based technique, such as the one used by Tracematches, could not be used for context-free properties because the state space is unbounded.

The coenables technique reclaims much more memory than the garbage collection of the previous version of JavaMOP, which, as already explained, has to wait for all bound parameter objects to be collected (see Section 4.4).

4.3.2 Monitor Garbage Collection

By using coenable sets, we can decide whether a monitor is unnecessary. However, removing unnecessary monitors efficiently from data structures is not trivial.

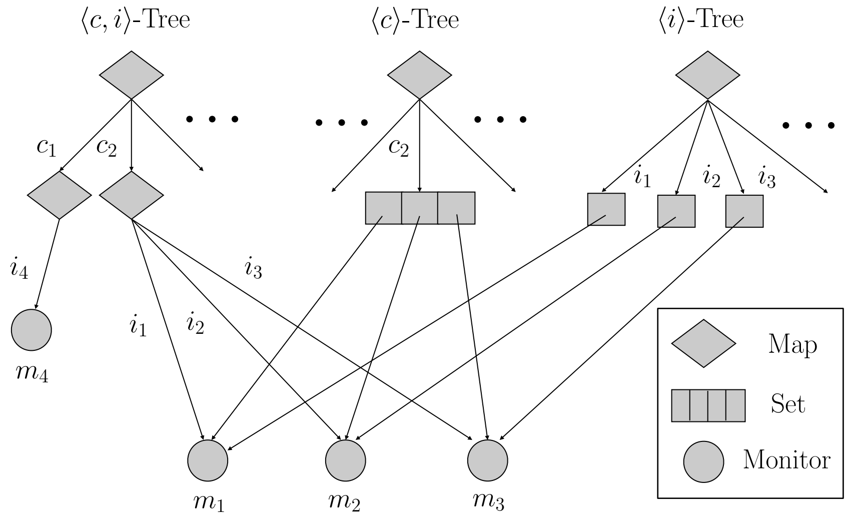


Figure 4.3: Indexing trees for Collection_UnsafeIterator

Consider the Collection_UnsafeIterator specification. Figure 4.3 shows the indexing trees for this specification. When i_1 is garbage collected, m_1 will be removed from $\langle c, i \rangle$-tree and $\langle i \rangle$-tree automatically because i_1 links are broken now. But m_1 will not be removed from $\langle c \rangle$-tree since its `Collection` c_2 is still alive. To remove the monitor m_1 from the $\langle c \rangle$-tree, either we should retrieve the set which contains m_1, causing more runtime overhead, or we should keep the set reference in m_1, increasing memory usage. After retrieving the set anyhow, we should remove m_1 from the set, which is expensive to do repeatedly. If we remove monitors actively like above (eager collection), the overhead of monitor removal easily overwhelms the benefit of having fewer monitors. This is because eager collection requires propagating the information regarding liveness of parameter objects to monitors far too frequently. Additionally, eager collection can result in removing monitors from some data structures that will never be used again.

Therefore, we use a lazy garbage collection scheme. We iterate monitor instances and propagate the information of garbage collections of parameter objects *lazily*, and we remove unnecessary monitors *lazily*. When an indexing tree containing a garbage collected parameter object is accessed and the tree detects this, it informs all the relevant monitors that it contains. Note that this is later than the actual garbage collection of the parameter object. Then, the monitor decides if it can still possibly reach a target state in the absence of the parameter object that has been garbage collected. Later when more space is needed in the data structure or when monitors are updated, we remove monitors from the accessed data structure but not from other data structures. A monitor is garbage collected when it is removed from all data structures. This is similar to mark-and-sweep garbage collection. If a data structure itself is garbage collected, the contained monitors do not have to be garbage collected separately.

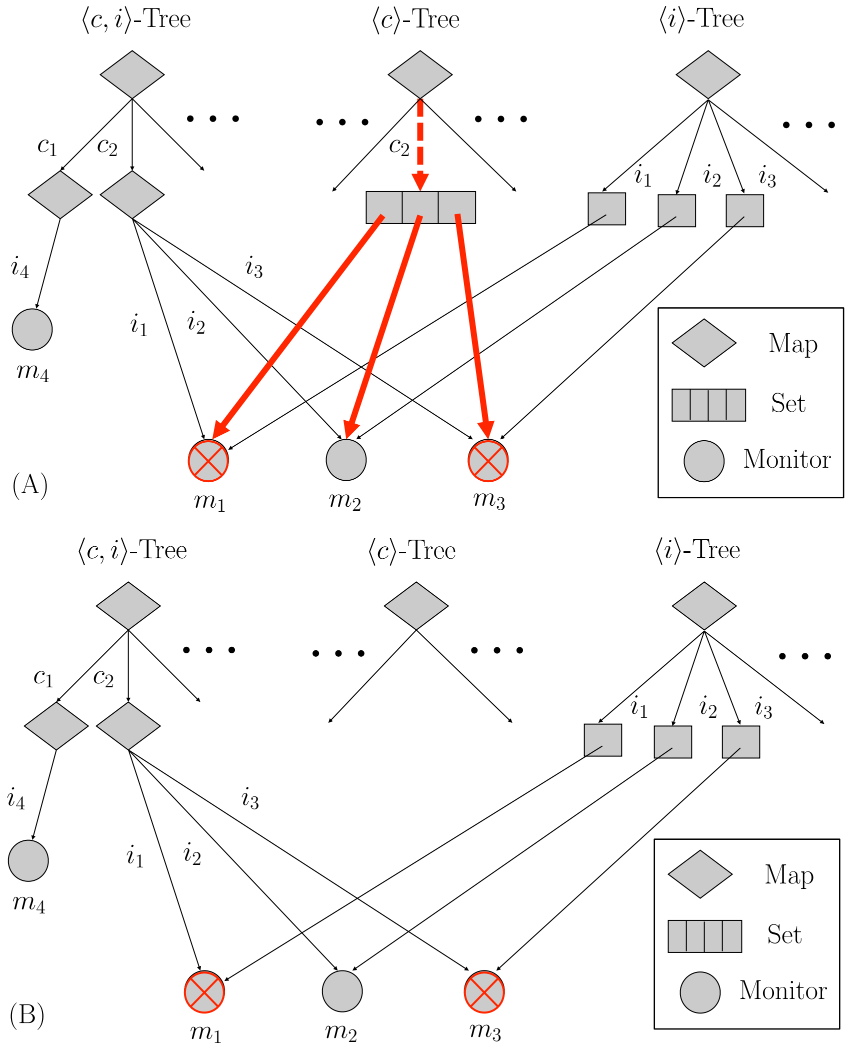


Figure 4.4: (A) Notifying monitors for garbage collected $\langle c_2 \rangle$ in the $\langle c \rangle$-tree. (B) Cleaning up the broken mapping in the $\langle c \rangle$-tree

The data structures used by previous runtime monitoring systems [9, 38, 32] are not sufficient for this lazy mechanism of monitor garbage collection. The challenge is how to efficiently garbage collect unnecessary monitor instances that are contained in the data structures. Using the standard data structures of previous systems, the overhead of instance removal easily overwhelms the benefit of having fewer monitor instances. Our specialized data structures, introduced here, track the garbage collection of parameter objects and remove unnecessary monitor instances when discovered using coenable sets (Section 4.3.1). In this section, we present the modified indexing trees as well as the mechanism by which unnecessary monitors are garbage collected.

Parameter Object Garbage Collection Notification

Propagation of parameter object garbage collection information starts from the mappings in the indexing tree. The mappings used in indexing trees of JavaMOP are implemented as a class called `MOPMap`. `MOPMap` uses `WeakReferences` for its keys as explained in Chapter 2. A `WeakReference` in Java does not stop the garbage collector from collecting its referent; when the referent is garbage collected, the `WeakReference` points to `null`. Whenever an operation (`put` or `get`) is performed on an `MOPMap` – or the hash table underlying the map needs to be expanded to store more entries – it looks through a subset of its entries for keys with `null` referents. When there is a key with a `null` referent due to a garbage collection, `MOPMap` notifies all of the monitor instances below itself in the indexing tree. For example, Figure 4.4 (A) shows a possible scenario where $\langle c_2 \rangle$ is garbage collected and the $\langle c \rangle$-tree is accessed. The $\langle c \rangle$-tree notifies all of the monitor instances below $\langle c_2 \rangle$.

Determining When Monitor Instances are Unnecessary

When a monitor is notified of a newly garbage collected parameter object, it decides whether it can still reach a verdict category of interest in the absence of garbage collected parameter objects by using the coenable sets introduced in Section 4.3.1. Each monitor instance stores the last event it receives, e, so that it may check $\textsc{Coenable}^{X}_{P,\mathcal{G}}(e)$, when this notification takes place. The monitor instance need simply check if all the parameter objects of any set in $\textsc{Coenable}^{X}_{P,\mathcal{G}}(e)$ are alive. JavaMOP statically translates $\textsc{Coenable}^{X}_{P,\mathcal{G}}(e)$ to a minimized boolean formula to make this check as efficient as possible:

$$\textsc{Aliveness}(e) = \bigvee_{S \,\in\, \textsc{Coenable}^{X}_{P,\mathcal{G}}(e)} \left(\bigwedge_{x \,\in\, S} \mathit{live}_x \right)$$

where live_x is a boolean that is true only if the parameter object of parameter x has not been garbage collected. Then, $\textsc{Aliveness}(e)$ is true only if the monitor is necessary. Maintaining live_x variables in a given monitor instance for each parameter and checking the generated boolean expression at runtime is sufficient for determining when said instance becomes unnecessary.

The monitor instances notified of garbage collected parameters in Figure 4.4 (A) check their $\textsc{Aliveness}$ to determine if they are unnecessary. Here, m_1 and m_3 are unnecessary and therefore marked. Note that the set under $\langle c_2 \rangle$ is not altered because other `MOPMaps` in the index tree still point to it. In Figure 4.4 (B), the `MOPMap` removed the broken map entry index by c_2. m_1 and m_3 will be removed at some future time when the $\langle c, i \rangle$-tree or $\langle i \rangle$-tree are accessed or expanded, as we explain in the next.

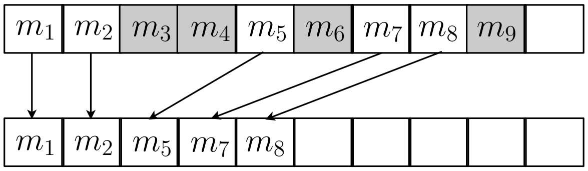


Figure 4.5: A compaction in `MOPSet` when some monitor instances are collectable

Removing Unnecessary Monitor Instances

Monitor instances are removed lazily because in many cases the maps and sets containing monitor instances flagged for removal may be garbage collected themselves. Eager removal would result in unnecessary work in such cases. For example, in Figure 4.4 (B), if the $\langle c_2 \rangle$-subtree in the $\langle c, i \rangle$-tree is going to be garbage collected, there is no reason to remove flagged monitor instances from it.

Unnecessary monitor instances are only removed when an indexing tree is accessed. Whenever an `MOPMap` looks for keys with `null` referents it also checks the values of mappings which do not have `null` referents. The value can be either a monitor instance, a set, or a lower level map. If the value is a flagged monitor instance or an empty data structure, it removes the mapping. If it is a set, it must be checked for internal monitor instances that have been flagged for removal. When a set is checked for unnecessary monitor instances, all of the instances are collected, and the remaining necessary monitor instances are compacted in one pass, as can be seen in Figure 4.5.

4.4 Evaluation

We evaluate our techniques for efficient parametric monitoring that integrate into a new version of JavaMOP. The new version of JavaMOP implements the generic parametric monitoring with the enable set optimization (Section 4.1), the indexing cache (Section 4.2), and formalism-independent monitor garbage collection (Section 4.3). Also, we compare the performance to the previous version of JavaMOP, and Tracematches, two of the most optimized monitoring systems in runtime and memory, respectively.

4.4.1 Experimental Settings

For our experiments, we used a Pentium 4 2.66GHz / 2GB RAM / Ubuntu 9.10 machine and version 9.12 of the DaCapo (DaCapo 9.12) benchmark suite [24]. We also present the result from the previous version, 2006-10 MR2 of DaCapo (DaCapo 2006-10), but only for the benchmarks that are not included in the new version of DaCapo: antlr, bloat, chart, hsqldb, and jython. Among deprecated benchmarks that DaCapo 9.12 does not provide any more, we favor the bloat benchmark from the DaCapo 2006-10 because it generates large overheads when monitoring Iterator-based properties. The bloat benchmark with the UnsafeIter specification causes 11258% runtime overhead (i.e., 113 times slower) and uses 7.8MB of heap memory in Tracematches, and causes 769% runtime overhead and uses 175MB in the previous version of JavaMOP, while the original program uses only 4.9MB. Also, although the DaCapo 9.12 provides jython, Tracematches cannot instrument jython due to an error, while all versions of JavaMOP can instrument it. Thus, we present the result of jython from the DaCapo 2006-10. The default data input for DaCapo was used and the -converge option to obtain the numbers after convergence within ±3%. We also looked into other benchmarks including Java Grande [70] and SPECjvm 2008 [5], and saw little to no overhead even with our Iterator-based properties; we omit the result in the dissertation. Instrumentation introduces a different garbage collection behavior in the monitored program, sometimes causing the program to slightly outperform the original program; this accounts for the negative overheads seen in both runtime and memory.

We used the Sun JVM 1.6.0 for the entire evaluation. The AspectJ compiler (ajc) version 1.6.4 is used for weaving the aspects generated by JavaMOP into the target benchmarks. Another AspectJ compiler, abc [17] 1.3.0, is used for weaving Tracematches properties because Tracematches is part of abc and does not work with ajc. For the previous version of JavaMOP, we used the release version, 2.1.2, from the JavaMOP website [3], but with the -noopt1 option to turn off the enable set optimization. For the new version of JavaMOP, we used the release version, 2.3.2, from the JavaMOP website, as well. For Tracematches, we used the release version, 1.3.0, from [7], which is included in the abc compiler as an extension. To figure out the reason that some examples do not terminate when using Tracematches, we also used the abc compiler for weaving aspects generated from JavaMOP properties. Note that JavaMOP is AspectJ compiler independent. JavaMOP shows similar overheads and terminates on all examples when using the abc compiler for weaving as when ajc is used. Because the overheads are similar, we do not present the results of using abc to weave JavaMOP generated aspects in the dissertation. However, using abc to weave JavaMOP properties confirms that the high overhead and non-termination come from Tracematches itself, not from the abc compiler.

The following properties are used in our experiments. They were borrowed from [28, 27, 64, 36].

- HasNext: Do not use the next element in an Iterator without checking for the existence of it;
- UnsafeIter: Do not update a Collection when using the Iterator interface to iterate its elements;
- UnsafeMapIter: Do not update a Map when using the Iterator interface to iterate its values or its keys;

- UnsafeSyncColl: If a `Collection` is synchronized, then its iterator also should be accessed synchronously;
- UnsafeSyncMap: If a `Collection` is synchronized, then its iterators on values and keys also should be accessed in a synchronized manner.

All of them are tested on Tracematches, and the previous and new versions of JavaMOP for comparison. We have tested several non-`Iterator` based properties: HashSet, SafeEnum, SafeFile, and SafeFileWriter [28, 27, 64, 36]. None of these properties produce overheads above 5% in any of the DaCapo benchmarks, thus their results are not presented in the dissertation.

4.4.2 Results and Discussions

Tables 4.5, 4.6, 4.7, 4.8, 4.9, and 4.10 summarize the results of the evaluation. Note that the structure of the DaCapo 9.12 allows us to instrument all of the benchmarks plus all supplementary libraries that the benchmarks use, which was not possible for DaCapo 2006-10. Therefore, fop and pmd show higher overheads than the benchmarks using DaCapo 2006-10 from [36]. While other benchmarks show overheads less than 80% in the previous version of JavaMOP, bloat, avrora, batik, and pmd show prohibitive overhead in both runtime and memory performance. This is because they generate many iterators and all properties in this evaluation are intended to monitor iterators. For example, bloat creates 1,625,770 collections and 941,466 iterators in total while 19,605 iterators coexist at the same time at peak, in an execution. avrora and pmd also create many collections and iterators. Also, they call hasNext() 78,451,585 times, 1,158,152 times and 4,670,555 times and next() 77,666,243 times, 352,697 times and 3,607,164 times, respectively. Therefore, in this section, we mainly discuss those examples that have shown most overhead for the previous version of JavaMOP, although the new version of JavaMOP shows improvements for other examples as well.

Tables 4.5, 4.6, 4.7 show the percent runtime overhead of Tracematches and the previous and new versions of JavaMOP. The previous version of JavaMOP shows, on average, 54% runtime overhead, but the optimized JavaMOP shows only 21% runtime overhead (16% except the cases where the previous version crashed for out of memory). This is less than half of the average runtime overhead that the previous version of JavaMOP showed. Compared to Tracematches, the optimized JavaMOP shows orders of magnitude less runtime overhead; Tracematches shows, on average, 309% runtime overhead. Even if we ignore the fact that Tracematches and the previous version of JavaMOP crashed on several cases, it clearly shows the improvements in runtime overhead when our optimization techniques were used. In the worst case benchmark program, bloat, the optimized JavaMOP managed its runtime overhead under 260%, while the previous JavaMOP shows more than 440% runtime overhead and Tracematches

		HasNext			UnsafeIter		
	ORIG (sec)	TM	Old	New	TM	Old	New
antlr	3.6	-1	4	-2	0	0	-2
bloat	14.4	2119	448	116	11258	769	251
chart	12.2	0	0	-2	11	5	-1
hsqldb	8.4	15	0	-3	17	-1	-3
jython	9.0	13	0	0	11	-4	1
avrora	13.9	45	48	55	637	298	118
batik	3.5	3	4	3	355	11	8
eclipse	79.5	-2	1	-1	0	2	-1
fop	2.0	200	57	48	350	23	13
h2	18.7	89	17	13	128	7	4
luindex	2.9	0	1	1	0	1	1
lusearch	25.3	-1	7	0	1	0	2
pmd	8.4	176	89	59	1423	162	123
sunflow	32.5	47	5	3	7	0	0
tomcat	14.1	8	-1	1	37	-1	1
tradebeans	45.7	0	1	1	1	0	2
tradesoap	95.0	1	0	0	2	-2	1
xalan	20.9	4	-2	2	27	2	2

Table 4.5: Average *Percent* Runtime Overhead for Tracematches(TM), Previous JavaMOP(Old), and optimized JavaMOP(New) against HasNext and UnsafeIter (convergence within 3%, OOM = Out of Memory)

		UnsafeMapIter			UnsafeSyncColl		
	ORIG (sec)	TM	Old	New	TM	Old	New
antlr	3.6	-2	5	1	-1	2	-1
bloat	14.4	OOM	OOM	178	1359	735	212
chart	12.2	-1	4	-2	-2	1	-1
hsqldb	8.4	29	0	-3	9	0	-2
jython	9.0	150	11	3	11	-4	1
avrora	13.9	OOM	OOM	42	75	140	80
batik	3.5	OOM	65	5	208	444	9
eclipse	79.5	5	-1	0	-4	-1	1
fop	2.0	OOM	OOM	14	OOM	OOM	25
h2	18.7	1350	OOM	6	868	69	4
luindex	2.9	1	0	1	1	1	1
lusearch	25.3	2	2	0	4	0	1
pmd	8.4	OOM	OOM	188	1818	OOM	76
sunflow	32.5	9	6	1	13	6	5
tomcat	14.1	3	-1	1	2	-1	1
tradebeans	45.7	5	-1	-1	-1	-1	2
tradesoap	95.0	2	0	1	0	0	1
xalan	20.9	10	1	2	3	1	3

Table 4.6: Average *Percent* Runtime Overhead for Tracematches(TM), Previous JavaMOP(Old), and optimized JavaMOP(New) against UnsafeMapIter and UnsafeSyncColl (convergence within 3%, OOM = Out of Memory)

		UnsafeSyncMap		
	ORIG (sec)	TM	Old	New
antlr	3.6	0	2	0
bloat	14.4	1942	858	130
chart	12.2	-2	3	-2
hsqldb	8.4	7	-1	-3
jython	9.0	10	-4	0
avrora	13.9	54	73	16
batik	3.5	5	7	0
eclipse	79.5	OOM	2	-1
fop	2.0	OOM	OOM	19
h2	18.7	83	25	5
luindex	2.9	2	2	0
lusearch	25.3	3	1	1
pmd	8.4	OOM	OOM	26
sunflow	32.5	17	8	6
tomcat	14.1	2	-1	3
tradebeans	45.7	3	2	5
tradesoap	95.0	2	0	5
xalan	20.9	4	-2	3

Table 4.7: Average *Percent* Runtime Overhead for Tracematches(TM), Previous JavaMOP(Old), and optimized JavaMOP(New) against UnsafeSyncMap (convergence within 3%, OOM = Out of Memory)

		HasNext			UnsafeIter		
	ORIG (MB)	TM	Old	New	TM	Old	New
antlr	4.3	4.4	4.1	3.8	4.8	4.0	4.5
bloat	4.9	40.3	19.3	13.9	7.8	175.4	79.0
chart	17.0	17.4	17.3	17.0	16.9	16.5	17.2
hsqldb	136.5	136.1	136.7	137.6	139.1	136.8	137.6
jython	4.9	5.1	4.7	4.8	5.5	5.1	5.0
avrora	4.7	4.6	12.1	9.1	4.4	114.0	15.8
batik	77.3	79.2	81.9	79.3	75.2	93.4	86.6
eclipse	101.0	100.8	104.0	97.1	98.3	100.3	110.3
fop	23.9	97.4	47.1	52.5	24.3	25.6	29.4
h2	267.1	267.8	588.8	565.2	267.2	267.5	262.4
luindex	6.8	5.6	6.7	5.6	6.3	7.4	6.8
lusearch	4.6	4.7	4.6	4.8	4.6	4.3	4.2
pmd	22.3	56.9	65.5	48.5	17.2	147.2	86.4
sunflow	4.5	4.5	4.8	4.9	4.8	4.6	4.7
tomcat	11.7	11.4	11.6	11.4	12.5	11.8	11.5
tradebeans	62.9	62.9	62.4	62.1	63.7	63.9	64.1
tradesoap	63.9	61.8	64.8	63.3	63.4	64.7	64.4
xalan	4.9	4.9	5.0	5.1	4.9	5.0	4.9

Table 4.8: Peak memory usage (in MB) for Tracematches(TM), Previous JavaMOP(Old), and optimized JavaMOP(New) against HasNext and UnsafeIter (during 5 iterations, OOM = Out of Memory)

		UnsafeMapIter			UnsafeSyncColl		
	ORIG (MB)	TM	Old	New	TM	Old	New
antlr	4.3	4.1	4.0	4.6	4.1	4.2	4.2
bloat	4.9	OOM	OOM	56.7	6.7	100.0	48.3
chart	17.0	16.6	15.9	19.2	17.0	16.4	17.2
hsqldb	136.5	136.0	140.0	136.8	136.1	146.2	146.3
jython	4.9	6.1	20.9	5.1	5.3	4.9	5.4
avrora	4.7	OOM	OOM	8.5	4.3	18.4	12.6
batik	77.3	OOM	173.8	79.6	78.2	180.7	85.1
eclipse	101.0	106.9	198.9	101.1	100.4	115.1	90.1
fop	23.9	OOM	OOM	28.1	OOM	OOM	24.8
h2	267.1	312.4	OOM	268.2	271.4	1456.7	265.5
luindex	6.8	7.4	6.8	6.9	7.4	7.5	7.5
lusearch	4.6	4.0	4.2	4.8	4.5	4.3	4.6
pmd	22.3	OOM	OOM	93.6	20.3	OOM	84.6
sunflow	4.5	4.7	4.6	4.4	5.1	4.4	4.9
tomcat	11.7	11.9	12.0	11.0	11.3	11.9	11.3
tradebeans	62.9	63.3	62.4	62.7	63.2	62.8	62.0
tradesoap	63.9	64.1	65.4	62.0	60.7	64.1	65.9
xalan	4.9	4.9	4.9	4.9	5.0	4.7	5.0

Table 4.9: Peak memory usage (in MB) for Tracematches(TM), Previous JavaMOP(Old), and optimized JavaMOP(New) against UnsafeMapIter and UnsafeSyncColl (during 5 iterations, OOM = Out of Memory)

		UnsafeSyncMap		
	ORIG (MB)	TM	Old	New
antlr	4.3	4.6	4.4	4.9
bloat	4.9	6.9	25.8	12.3
chart	17.0	17.4	16.4	17.1
hsqldb	136.5	142.1	136.4	137.0
jython	4.9	5.8	5.0	5.1
avrora	4.7	4.4	12.4	4.9
batik	77.3	79.9	84.8	76.7
eclipse	101.0	OOM	102.3	98.7
fop	23.9	OOM	OOM	25.2
h2	267.1	271.0	688.2	270.0
luindex	6.8	7.1	7.3	11.0
lusearch	4.6	4.6	4.8	4.7
pmd	22.3	OOM	OOM	32.9
sunflow	4.5	4.5	4.8	4.5
tomcat	11.7	11.4	11.3	11.8
tradebeans	62.9	64.0	62.7	64.0
tradesoap	63.9	65.5	65.1	65.6
xalan	4.9	5.1	5.1	4.9

Table 4.10: Peak memory usage (in MB) for Tracematches(TM), Previous JavaMOP(Old), and optimized JavaMOP(New) against UnsafeSyncMap (during 5 iterations, OOM = Out of Memory)

shows more than 1300%, and both of them crashed for UnsafeMapIter. With avrora, on average, the new version of JavaMOP shows 62% runtime overhead, while the previous version of JavaMOP shows 140% runtime overhead and Tracematches shows 203% and both of them hang for UnsafeMapIter. With pmd, on average, the new version of JavaMOP shows 94% runtime overhead, while the previous version of JavaMOP shows 125% runtime overhead and hangs for three specifications, and Tracematches shows 1139% and hangs for two specifications.

Tables 4.8, 4.9, and 4.10 show the peak memory usage of the three systems. the new version of JavaMOP has lower peak memory usage than the previous version of JavaMOP in most cases. The cases where the new version of JavaMOP does not show lower peak memory usage are within the limits of expected memory jitter. However, memory usage of the new version of JavaMOP is still higher than the memory usage of Tracematches in some cases. Tracematches has several finite automata specific memory optimizations [19], which cannot be implemented in a formalism-independent system like the new version of JavaMOP. Although Tracematches is sometimes more memory efficient, it shows prohibitive runtime overhead monitoring bloat and pmd. There is a trade-off between memory usage and runtime overhead. If the new version of JavaMOP more actively removes terminated monitors, memory usage will be lower, at the cost of runtime performance. Overall, our monitor termination optimization achieves the most efficient parametric monitoring system with reasonable memory performance.

From this experiment, considering the fact that these cases are the worst combinations of benchmark programs and properties, we can see that our research on efficiency of runtime monitoring were successful in realizing efficient runtime monitoring of parametric properties.

4.4.3 Characteristics of Specifications and Optimization Techniques

Each specification has different number of parameters, different number of events, and event patterns so that optimization techniques improve them differently. We look into the characteristics of specifications and the nature of optimization techniques. Figure 4.11 summarizes the evaluation with partially enabled optimization techniques, on bloat which shows the most runtime overhead in the main evaluation in this section. As more optimization techniques are applied, the runtime performances improve accordingly. Indexing Cache improves runtime performance over all specifications; therefore we focus on how the other two techniques affect each specification.

HasNext

HasNext has only one parameter, `Iterator` and there is no non-parameterized event. Thus, there is no partial initialization of parameters; the enable set optimization does not make any change on this specification. Also, it is clear

	Old			New
Enable Set		✓	✓	✓
Indexing Cache			✓	✓
Monitor Collection				✓
HasNext	448	448	135	116
UnsafeIter	769	769	400	251
UnsafeMapIter	OOM	1091	920	178
UnsafeSyncColl	735	712	487	212
UnsafeSyncMap	858	660	407	130

Table 4.11: Average *Percent* Runtime Overhead for JavaMOP, gradually enabling the optimization techniques from Previous JavaMOP(Old) to optimized JavaMOP(New), against bloat (convergence within 3%, OOM = Out of Memory)

that we can garbage collect a monitor when its parameter is garbage collected. Therefore, the JVM garbage collection effectively collects unnecessary monitors without help from our monitor garbage collection. As we can see in Figure 4.11, the enable set optimization makes no change and the monitor collection makes only a small change, which comes from more lighter data structures that the new version of JavaMOP supports.

UnsafeIter

UnsafeIter has two parameters, `Collection` and `Iterator`. Since the creation event initiates all two parameters, the enable set optimization does not affect this specification as well. However, unlike HasNext, it is unclear when to garbage collect monitors in this specification. When all parameters are garbage collected, it is obvious that the corresponding monitor can be collected; the JVM garbage collection can handle this case. When an `Iterator` is collected, there is no way for the related monitor to reach the final state. Since the JVM garbage collection cannot handle this case, our monitor garbage collection can handle it, improving the performance.

UnsafeMapIter

UnsafeMapIter has three parameters, `Map`, `Collection`, and `Iterator`. It has a creation event which initiates only the first two parameters. Thus, enable set optimization effectively removes unnecessary creation of monitors. In Figure 4.11, it was not possible to monitor this specification without the enable set optimization, but it becomes possible after applying this optimization. When an `Iterator` is garbage collected, the related monitor cannot reach to the final state, therefore the monitor can be garbage collected although its map and collection are still alive. While the JVM garbage collector cannot handle this case since the monitor is still accessible, our monitor garbage collection can effectively handle this. In Figure 4.11, we can see a great improvement since `Map` and `Collection` have much longer lifetime than `Iterator`.

UnsafeSyncColl

UnsafeSyncColl has two parameters, `Collection` and `Iterator`. Its creation event initiates only one parameter, `Collection`, therefore the enable set optimization improves the performance. However, the improvement is small since it has only two parameters while UnsafeMapIter and UnsafeSyncMap has three parameters. This is because there are more possible partial parameter instances to skip monitoring in the specifications with three parameters. Like UnsafeIter, the garbage collection improves the performance of monitoring this specification since it can handle the case where only the `Iterator` is collected.

UnsafeSyncMap

UnsafeSyncMap has three parameters, `Map`, `Set`, and `Iterator`. The creation event initiates only the first parameter, `Map` and it has more number of parameters than UnsafeSyncColl, therefore the enable set optimization improves the performance of monitoring this specification more than it does on UnsafeSyncColl (Figure 4.11). Also, the garbage collection greatly improves the performance for UnsafeMapiter due to the same reason.

4.5 Summary

In this chapter, we present a series of formalism-independent optimization techniques for parametric monitoring. Our thorough evaluation shows our optimization techniques effectively improves not only average performance but also worst case performance. Since all optimization techniques introduced in this chapter are formalism-independent, there are some advantages and disadvantages. While these techniques can be applied to other parametric monitoring system for its generality, they cannot fully utilize characteristics of each logical formalism for better optimization. Although we achieved the best performance among parametric monitoring systems even with these formalism-independent optimization techniques, further improvements can be achieved using formalism-dependent optimizations. For example, in the enable set optimization, state-based analysis can give finer grained results than the current event-based analysis. However, state-based analysis might not be applied to other formalisms (e.g. context free grammar).

Chapter 5

Scalable Parametric Monitoring

In real usages of parametric runtime monitoring, it is natural to monitor multiple specifications simultaneously (e.g., security policies). However, to the best of the author's knowledge, all earlier efforts on parametric monitoring have been focusing on better performance when monitoring a single specification. Many of the existing parametric monitoring systems are not capable of monitoring multiple specifications simultaneously, or their runtime and memory overheads increase linearly (or worse) as they monitor more specifications. Those parametric monitoring systems easily become prohibitive with the existence of a large number of specifications. A practical parametric monitoring system must be *scalable* to the number of specifications that it monitors simultaneously.

Theoretically, if all specifications are independent from each other without any overlap in declared events or parameter types, there is no way to monitor them more efficiently. However, in practice, there are likely multiple specifications on the same class, often sharing some events and parameter types. Among 137 specifications from the Java API documentation of three main packages, in [59], only 42 specifications are totally independent from all the other specifications. Therefore, it is a reasonable assumption that some specifications describe behaviors of the same parameter, sharing events and parameters.

In this chapter, we present scalable parametric monitoring techniques for monitoring multiple simultaneous specifications more efficiently in the presence of some overlaps between specifications. The main idea of the scalable techniques is to share resources for monitoring between specifications, reducing the memory usage and utilizing the caches more often. Since our scalable techniques are formalism-independent and address general issues in the indexing tree technique, they can be applied to other parametric monitoring systems that use similar indexing tree structures. Also, they are orthogonal to other optimization techniques like static optimization [62, 29, 28, 41], which reduce runtime and memory overhead significantly. However, we deliberately disabled static optimizations in this chapter to measure the effectiveness of our scalable techniques properly.

This chapter is structured as follows: Section 5.1 presents a thorough profiling of current runtime overheads from monitoring, and discusses the main current bottlenecks in monitoring; Section 5.2 discusses our scalable parametric monitoring techniques in detail; Section 5.3 presents our evaluation results for the 137 specifications; Section 5.4 discusses some ineffectual approaches that we have tried; and Section 5.5 concludes.

5.1 Overhead Analysis

In this section, we analyze the overhead of monitoring to find the main bottlenecks in monitoring. For this analysis, we have selected 9 specifications[1] that have caused the most overhead in previous evaluations. We run the specifications on the bloat and pmd benchmarks because they have shown the largest overheads among the benchmarks in our evaluation (Section 5.3). We use the same system settings from the evaluation, and HPROF, the Heap/CPU profiling tool included in the Sun JDK [6] is used to obtain performance statistics. There are two modes for CPU usage analysis in HPROF: the CPU Usage Times Profile and the CPU Usage Sampling Profile. The CPU Usage Times Profile adds a considerable amount of overhead, obstructing the analysis of the actual bottlenecks. Moreover, we do not need to know the exact time distribution to figure out where bottlenecks occur. The CPU Usage Sampling Profile, which causes less performance degradation, is good enough for this analysis. Since the CPU Usage Sampling Profile does not combine the results for the same method of different object instances, we manually combine them and categorize.

Tables 5.1 and 5.2 summarize the profiling results for monitoring bloat and pmd. The results for bloat show total overhead of 1330%; that is 1430% total execution time compared to the original non-monitored bloat. In the same way, monitoring pmd shows a total overhead of 831%. Because profiling can change the program behavior, numbers may contain errors, so they should be considered as rough estimations.

The MOPSet.event entry in Table 5.1 shows the overhead spent updating monitor states when events occur. This component is formulated from the property of the specification, and is already optimized well. MOPMap.cleanup and MOPMap.full_cleanup remove mappings of garbage collected parameter objects and monitors. The difference is whether it partially or fully scans the map. These cleanup methods are well tuned so that they are unlikely to be improved significantly. The methods MOPMap.endObject and MOPSet.endObject propagate information about garbage collected parameters. They consist of simple statements and have already been thoroughly optimized [51].

System.identityHashCode is the system default hashing function provided in the Java API, which is based on reference identity instead of the equals method provided by classes. It returns the same hash code for objects a and b if a == b, and tries to return different codes otherwise, but uniqueness is not guaranteed. Although this is just one of several statements in the MOPMap.get method that

[1] Map_UnsafeIterator, Collection_UnsafeIterator, Iterator_HasNext, Collections_SynchronizedCollection, NavigableMap_Modification, Collections_SynchronizedMap, Iterator_RemoveOnce, List_UnsynchronizedSubList, Collections_SortBeforeBinarySearch

Overhead Fraction	Method Name
355%	Original Program
281%	MOPSet.event
205%	MOPMap.cleanup
130%	System.identityHashCode
69%	MOPMap.get
67%	MOPSet.size
51%	MOPMap.endObject
28%	Aspect Code
27%	MOPMap.full_cleanup
22%	MOPSet.endObject

Table 5.1: Overhead distribution when monitoring bloat (total overhead: 1330%)

Overhead Fraction	Method Name
479%	Original Program
90%	MOPSet.event
56%	MOPMap.cleanup
28%	System.identityHashCode
25%	MOPSet.size
13%	MOPMap.get
8%	MOPMap.full_cleanup
7%	MOPMap.endObject
6%	WeakReference ⟨init⟩
5%	MOPSet.endObject

Table 5.2: Overhead distribution when monitoring pmd (total overhead: 831%)

Description	Peak Memory Usage	Young Garbage Collection Time	Full Garbage Collection Time
Original bloat	5MB	6%	2%
Original pmd	21MB	7%	8%
Monitoring bloat (out of 1330% overhead)	970MB	278%	258%
Monitoring pmd (out of 831% overhead)	603MB	172%	181%

Table 5.3: Memory usage analysis

retrieves monitor(s) for a parameter instance, it produces more overhead than all other methods combined. Calling this method is unavoidable since it is used to retrieve keys in the MOPMap implementation. However, we need to call this method as little as possible.

While many monitoring components show significant overhead, it is notable that the original program components are also slower when monitoring is present (i.e. 100%). To understand this situation, we analyze the memory usage when monitoring, using Java Management Extensions (JMX) [1]. Table 5.3 summarizes the memory usage analysis. Monitoring triggers huge memory overheads, resulting in significantly more garbage collection time. With respect to the original program execution time (100%), in monitoring bloat, young object garbage collection takes 278% and full garbage collection takes 258%. In total, garbage collection takes 536% when monitoring bloat and 353% when monitoring pmd. This explains why the original components of the code run far slower when monitoring is present.

We must conclude that the main remaining bottleneck to runtime performance in monitoring is excessive memory usage. Huge memory overhead causes more frequent and longer garbage collections, resulting in larger runtime overhead. We should reduce memory overhead to optimize runtime performance. For example, in Table 5.2, `WeakReference` object initializations show 6% overhead, while there is no other class ranked in the result. This is because there is a very large number of weak references. We need to reduce the number of objects created for monitoring purposes, especially weak references.

5.2 Optimizations for Scalability

The more specifications that we monitor simultaneously, the more overhead. Our goal is to improve the overhead in the presence of multiple specifications by finding structures and parts of the monitoring algorithm that may be shared between different specifications. If no specifications overlap with others, in terms of declared events or parameters types, there is nothing much we can improve. Theoretically, the overhead in this case will be the sum of overheads from monitoring them individually. When the memory overhead is excessive, it can be worse than the sum because of the garbage collection behavior.

However, in practice, there are generally multiple specifications for each class, often sharing some events. Among 137 specifications from [59], only 42 specifications are totally independent from all other specifications. Another 95 specifications share parameters or events with some of other specifications. By sharing resources between overlapping specifications we can achieve a truly scalable parametric runtime monitoring system.

In this Section, we explain new techniques for increasing runtime and memory performance first, then we focus on the big picture of the new monitoring mechanism in comparison with the previous monitoring mechanism. Our techniques are formalism-independent and general so that they can be applied to other parametric monitoring systems that use similar indexing tree structures.

5.2.1 Global WeakReference Table

As explained in Chapter 2, `WeakReference` is a reference class that refers to an object without blocking it from garbage collection. The indexing tree uses weak references to store parameter objects in its mappings, without blocking garbage collections. In previous versions of JavaMOP, there was no collaboration between specifications, so each specification created a weak reference object for each parameter object. Thus, multiple weak references were potentially created for the same parameter object, if it appeared in different specifications. There is no need to have multiple copies of `WeakReference`; it simply wastes memory.

As a solution to share `WeakReference` objects between specifications, we introduce a global `WeakReference` table, implemented in the class `GlobalWeakRefTable`, for each parameter type, which all specifications share. This table takes a parameter object as an input and outputs a weak reference. If there is no weak reference in the table for the input object, the table will create one. Thus, weak references will be created only by this table and there will be exactly one copy for one parameter object. Also, upon a non-creation event, we can query the existence of the weak reference without creating one. If there is no weak reference for the parameter object in the table, then there is no monitor in any specification for the parameter object. Thus, we can skip the rest of the steps for checking the existence of monitors for the non-creation event.

The functionality of the `GlobalWeakRefTable` is similar to `HashMap` from the Java API, but its implementation is totally different. If the `GlobalWeakRefTable` stores keys (parameter objects) and values (weak references) in its internal table like `HashMap`, it will cause memory leaks. Instead, the `GlobalWeakRefTable` stores only weak references. Since weak references can refer to the original objects, we can retrieve the weak reference for an object by checking if the weak reference points to the object.

Although the `GlobalWeakRefTable` introduces one more step in the monitoring mechanism, it reduces not only memory overhead by reducing the number of weak references, but also runtime overhead. From the analysis in Section 5.1, we know that System.identityHashCode() causes the most runtime overhead in the indexing trees. Instead of calling this method in each indexing tree, each `GlobalWeakRefTable` calls this method and stores the result in weak references so that indexing trees can reuse it. To allow this, we implement `MOPWeakReference`, a subclass of `WeakReference` which has a hashcode field, and change the indexing tree to take `MOPWeakReference` as input rather than parameter objects. With this change, indexing trees no longer call the System.identityHashCode() method, removing the main overhead in accessing them. The `GlobalWeakRefTable` calls this method at most once for each parameter object in an event, minimizing the number of the method calls to System.identityHashCode().

The GlobalWeakRefTable is essentially the same as the indexing tree except that it does not return monitors. It cleans up references to garbage collected objects and expands the internal data structure just like the indexing tree does [51]. We can reduce overhead even more by combining GlobalWeakRefTables with relevant indexing trees, reducing the number of tables and maps. If there is an indexing tree that has the same parameter type at the first level as the GlobalWeakRefTable, they can be combined into one data structure. In the majority of cases, GlobalWeakRefTables can be combined with indexing trees. Among the many GlobalWeakRefTables for the 137 specifications from [59], there are only two GlobalWeakRefTables that cannot be combined into indexing trees when monitoring individually, and all GlobalWeakRefTables can be combined into indexing trees when monitoring them simultaneously.

5.2.2 Caches for Global WeakReference Table

Under our new technique, the GlobalWeakRefTable is the most frequently accessed data structure in monitoring since all events should query this table before accessing any indexing tree. Therefore, it is important to optimize this table. One natural and common method of optimization is caching. In the previous approach, there was already an indexing cache (Section 4.2). After adding GlobalWeakRefTables, it caches not only a monitor but also weak references for the monitor so that it can reduce the number of the method calls to System.identityHashCode(). Thus, it acts as a cache for both the indexing tree and the GlobalWeakRefTable.

Although the indexing cache provides a good cache hit ratio within a specification, it is not good enough when monitoring multiple specifications. First, since there are multiple events from different specifications for the same object, it is likely that multiple specifications consecutively access GlobalWeakRefTables for the same object, when their indexing caches miss. Second, the indexing cache is a one-entry cache which is fragile if more than two objects are frequently used together in an interleaved way.

To improve the performance upon this observation, we now use a one-entry level-1 cache to handle the first case and a multi-entry level-2 cache to handle the second case. On a query to the table, we first check the one-entry cache and when it misses, we check the multi-entry cache. However, if we linearly search in the multi-entry cache, the overhead will increase linearly with the number of entries in the cache. Thus, we use a mapping so that we can check only one entry at a time. Because each instrumentation point tends to access the same object consecutively, we index the multi-entry cache by a few least significant bits of the unique id number for instrumentation points, provided by AspectJ.

In this way, the multi-entry cache is implemented efficiently. The benefit of the caches surpasses the overhead from maintaining the caches in most cases.

5.2.3 Combining Indexing Trees

The indexing tree is one of the major bottlenecks in terms of both runtime and memory performance. It contains all of the mappings from parameter objects to monitors. The size of the indexing tree grows as the specification creates more monitors. Additionally, the indexing tree cleans up mappings of garbage collected parameter objects and monitors by itself. Therefore, we can reduce runtime and memory overhead by combining indexing trees. We can combine indexing trees if their defined parameter types share the same prefix. For example, indexing trees for ⟨`Collection`, `Iterator`⟩ and ⟨`Collection`⟩ can be combined but indexing trees for ⟨`Map`, `Collection`, `Iterator`⟩ and ⟨`Collection`, `Iterator`⟩ cannot be combined since the first parameter type, `Map`, appears only in the first.

Combining indexing trees between different specifications is also possible as long as they satisfy the condition for combining. However, it is usually inefficient because there is insufficient mapping overlap between specifications (Section 5.4). Thus, we combine indexing trees only within each specification. Combining indexing trees in each specification improves not only the performance of monitoring multiple specifications but also the performance of monitoring each specification.

For example, Figure 5.1 shows all indexing trees for Map_UnsafeIterator before combining them. There are six indexing trees for:

1. ⟨`Map`, `Collection`, `Iterator`⟩
2. ⟨`Map`, `Collection`⟩
3. ⟨`Map`⟩
4. ⟨`Collection`, `Iterator`⟩
5. ⟨`Collection`⟩
6. ⟨`Iterator`⟩

Among six indexing trees, the first three indexing trees can be combined into one, and the fourth and fifth indexing trees can be combined as well. As a result, three indexing trees will remain (Figure 5.2).

5.2.4 Eliminating HashEntry

`HashEntry` is an internal data structure of indexing trees for storing a mapping from a parameter to a monitor, a set, or a next-level map. `HashMap` from the Java API and `ReferenceIdentityMap` from the Apache Commons Collections Library [42] also use similar data structures for the same purpose. It contains a `WeakReference` object (specifically, `MOPWeakReference`), a monitor or

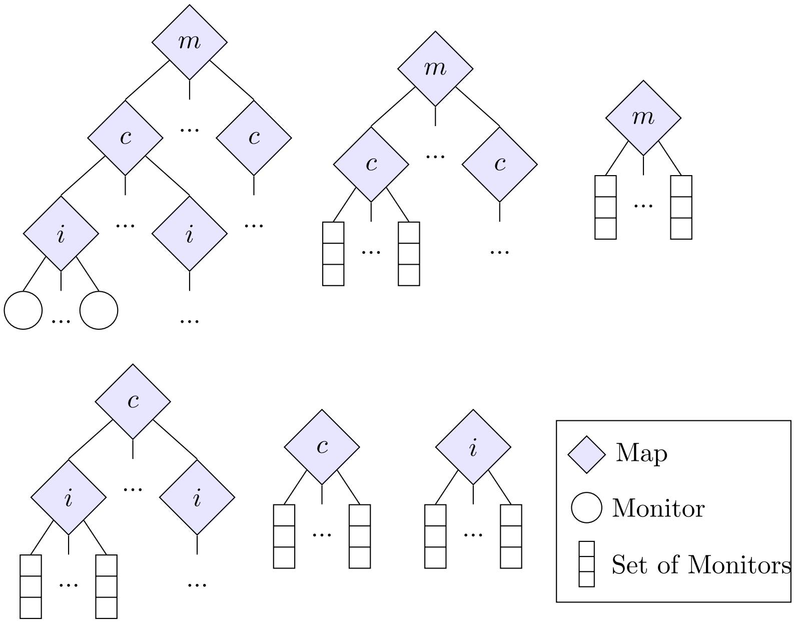


Figure 5.1: Indexing trees for Map_UnsafeIterator before combining

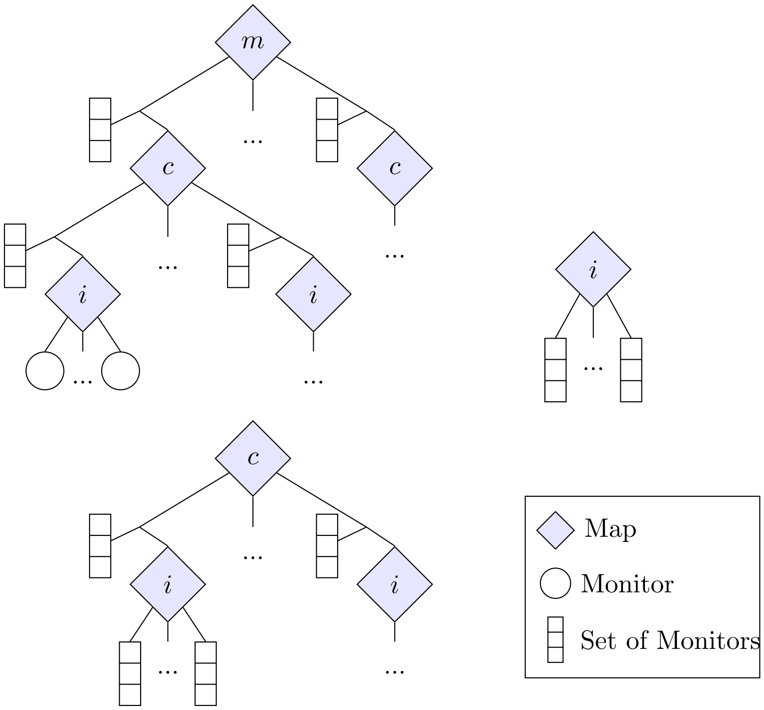


Figure 5.2: Indexing trees for Map_UnsafeIterator after combining

a set to contain monitors, and a reference to another HashEntry for chaining them to solve hash conflicts.

Although this data structure is small (it takes only 12 bytes plus some auxiliary data that the JVM adds, on 32bits x86 machine), it is the most created data structure in monitoring. For bloat, there are about twice as many HashEntry objects as monitors, in monitoring 137 specifications simultaneously in Section 5.3. This is because a monitor can belong to many indexing trees. Then, there are about 56 million HashEntry objects created. This costs at least 670 Megabytes in the total memory usage (note that it is not the peak memory usage).

We observe that a monitor or a set belongs to only one HashEntry object. Although a monitor can belong to many indexing trees, only the indexing tree for the fully instantiated parameter instances can directly retrieve a monitor for the given parameter instance. All other indexing trees return a set of monitors, and a set can belong to only one indexing tree. Therefore, a monitor or a set can replace HashEntry by piggybacking the information about the mapping. Since itself is a value, it only needs to piggyback a MOPWeakReference and a reference to another entry, which is a monitor or a set in this case. When indexing trees are combined, a HashEntry can contain multiple values. In this case, we do not apply this technique.

The memory usage of HashEntry is just transferred to monitors and sets except references to values, so improving the total memory usage is not the key point in this technique (nevertheless, it reduces about 200 Megabytes of total memory usage in monitoring bloat against 137 specifications). This technique reduces the number of objects to garbage collect when monitors are collected. In effect, the garbage collector of the JVM can reuse more space in the same amount of time; it improves the runtime and memory performance of monitoring noticeably.

5.2.5 Specification Activator

In monitoring multiple specifications, such as the 137 specifications from [59], it is common that only some of them are actively monitored when applied to a given program. This is because one program generally does not cover every specification in such a large set of standardized specifications. When a specification does not have any creation event during the execution of a program, it does not need to monitor the program at all. We keep a boolean value as an *activator* for each specification and activate it when there is at least one creation event. When the specification is not activated, we ignore all non-creation events, suppressing the unnecessary overhead. If there is no creation event at all during the execution, all non-creation events will be ignored.

This simple technique successfully deactivates unnecessary specifications during the execution of a program, reducing unnecessary runtime overhead. Even in monitoring a single specification, it can effectively remove unnecessary overhead. In our evaluation (Section 5.3), some specifications are effectively

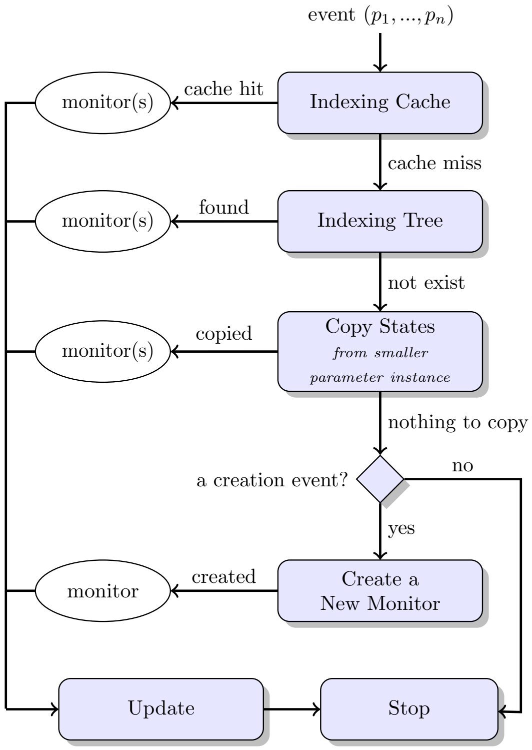


Figure 5.3: Overview of the previous monitoring mechanism

deactivated and show no overhead at all. The overhead of maintaining specification activators is essentially unnoticeable, far less than the error range of our evaluation (up to 3%).

5.2.6 Summary of New Monitoring Techniques

Figure 5.4 summarizes the scalable parametric monitoring mechanism using techniques introduced in this section. Compared to the previous monitoring mechanism summarized in Figure 5.3, there is an activator at the beginning and the `GlobalWeakRefTable` before the indexing tree. Also, instead of parameter objects, it uses weak references in accessing indexing trees.

The main idea of our scalable parametric monitoring is that the global `WeakReference` table, called `GlobalWeakRefTable`, allows sharing of weak references, reducing memory overhead. Also, caching on this table reduces runtime overhead over all specifications using it. Moreover, there are fewer indexing trees and there is no hash method call from the indexing tree. Thus, the overhead from indexing trees has been dramatically decreased. Since the “Copy State” component and the “Create a New Monitor” component also access the indexing tree to add new monitors, overheads from both components decrease as well.

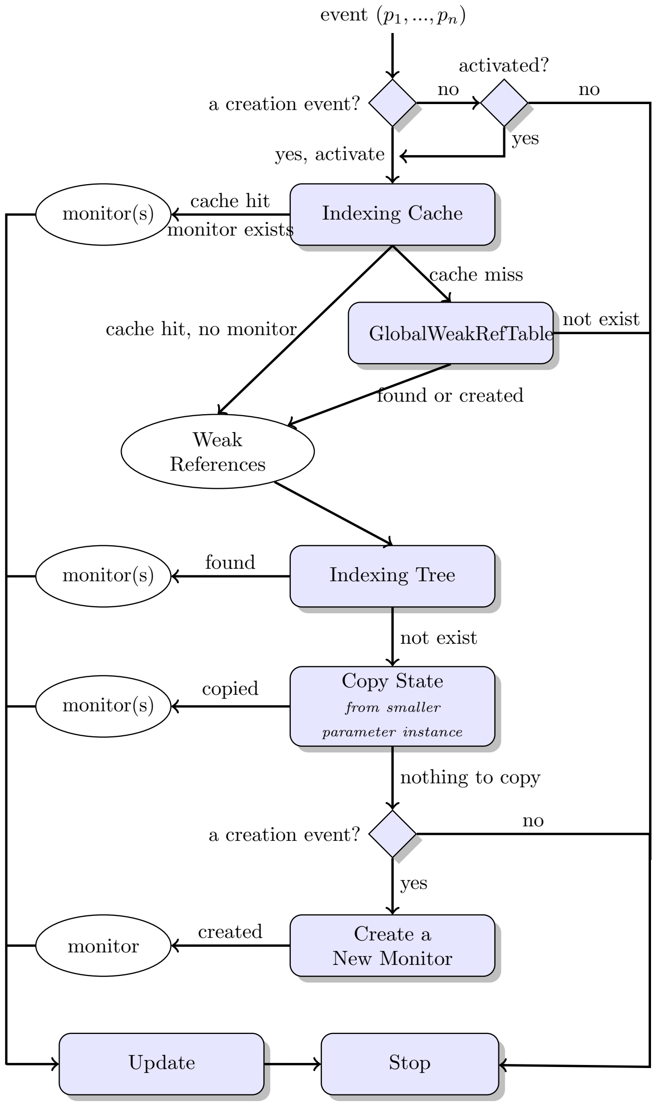


Figure 5.4: Overview of the scalable parametric monitoring mechanism

5.3 Evaluation

In this section, we evaluate JavaMOP with the presented scalability improvements on 137 specifications from [59]. We compare our work on scalability to the optimized version of JavaMOP (JavaMOP 2011) which implements all optimization techniques presented in Chapter 4. Before the work in this chapter, when monitoring a single property, JavaMOP had the best runtime performance of any monitoring system, while maintaining competitive memory performance (Section 4.4). Also, JavaMOP 2011 can monitor 137 specifications but not efficiently enough. To the best of the author's knowledge, there is no other parametric monitoring tool which is capable of practically monitoring 137 specifications simultaneously.

5.3.1 Experimental Settings

For our evaluation, we used a Pentium 4 2.66GHz / 2GB RAM / Ubuntu 9.10 machine and Sun JVM 1.6.0_10. For instrumenting benchmark programs with JavaMOP monitoring code, we used version 1.6.11 of the AspectJ compiler (ajc). We monitor 137 specifications for version 9.12 of the DaCapo (DaCapo 9.12) benchmark suite. We also present the result from the bloat benchmark in the old version of the DaCapo (DaCapo 2006-10) benchmark suite, because it generates large overheads and it is missing in the new version. We used the default data input size, and the -converge option so that the execution time result converges within 3%. AspectJ instrumentation can cause the code to run differently, sometimes resulting in negative overheads even without monitoring. Also, monitoring affects the garbage collection behavior with more memory pressure, often improving garbage collection time; this also accounts for the negative overheads.

All 137 specifications from [59] are based on the Java 6 API documentation concerning three main packages: 30 specifications for `java.io`, 49 specifications for `java.lang`, and 58 specifications for `java.util`. Some specifications are related to the end of the program execution. However, two versions of DaCapo iterate a benchmark program in one execution until the execution time converges. Therefore, we modified those specifications slightly so that they catch the end of iteration of a benchmark program.

5.3.2 Results and Discussions

Tables 5.4, 5.5, 5.6, 5.7, 5.8, 5.9, 5.10, and 5.11 summarize the results of the evaluation on the two versions of JavaMOP. Monitoring 137 specifications simultaneously is a considerably challenging task. While monitoring 137 specifications with bloat, there are 839,575,093 events and 27,826,935 monitors created. With pmd, there are 68,438,904 events and 9,510,880 monitors created. Also, in JavaMOP 2011, 129 indexing trees are required, but the indexing tree combination technique (Section 5.2.3) reduces the number of indexing trees to 105. Therefore, it is not surprising to see a huge overhead. Although JavaMOP 2011 was the most efficient parametric monitoring system until the work in this chapter, it shows more than 100% overhead on five benchmarks out of 15, including fop.

For fop, the instrumentation crashes because the added instrumentation results in a method larger than the 64KB limit for Java methods. The method size was already too big before the instrumentation, and our instrumentation makes it exceed the limit. In regular programming, the limit of 64KB seems reasonable; any method over 64KB should be re-designed and divided into several methods. However, for procedurally generated code, this limit imposed by Java seems too harsh. While we were unable to obtain overhead for fop with 137 simultaneous specifications in either version of JavaMOP, we do have numbers for monitoring the specification of each package separately.

		java.io			
# of specs		30			
		Previous		Scalable	
	Orig (sec)	Sum	Together	Sum	Together
bloat	14.4	6	-2	-3	0
avrora	13.9	7	8	0	0
batik	3.5	0	4	0	2
eclipse	79.5	0	-1	0	3
fop	2.0	8	7	18	0
h2	18.7	0	0	13	3
jython	13.6	10	-1	0	3
luindex	2.9	9	5	5	5
lusearch	25.3	14	13	17	13
pmd	8.4	0	-1	0	-2
sunflow	32.5	0	1	0	3
tomcat	14.1	0	-1	0	1
tradebeans	45.7	40	12	11	2
tradesoap	95.0	0	2	11	0
xalan	20.9	6	12	-7	24

Table 5.4: Average *percent* runtime overhead for Previous JavaMOP (JavaMOP 2011) and Scalable JavaMOP (JavaMOP 2012) against 30 specifications from java.io (convergence within 3%, N/A: instrumentation crashes)

		java.lang			
# of specs		49			
		Previous		Scalable	
	Orig (sec)	Sum	Together	Sum	Together
bloat	14.4	289	327	300	339
avrora	13.9	19	10	12	10
batik	3.5	0	3	0	1
eclipse	79.5	0	1	42	0
fop	2.0	96	56	52	54
h2	18.7	17	24	34	19
jython	13.6	27	21	18	23
luindex	2.9	5	5	12	2
lusearch	25.3	28	34	21	28
pmd	8.4	-3	8	0	6
sunflow	32.5	0	1	0	1
tomcat	14.1	0	0	0	1
tradebeans	45.7	33	3	26	-1
tradesoap	95.0	16	2	9	0
xalan	20.9	-17	-12	-32	-15

Table 5.5: Average *percent* runtime overhead for Previous JavaMOP (JavaMOP 2011) and Scalable JavaMOP (JavaMOP 2012) against 49 specifications from java.lang (convergence within 3%, N/A: instrumentation crashes)

		java.util			
# of specs		58			
		Previous		Scalable	
	Orig (sec)	Sum	Together	Sum	Together
bloat	14.4	1203	1493	762	556
avrora	13.9	468	336	279	177
batik	3.5	50	37	41	24
eclipse	79.5	0	1	7	0
fop	2.0	584	450	380	325
h2	18.7	71	54	70	38
jython	13.6	112	90	85	73
luindex	2.9	3	5	11	6
lusearch	25.3	29	26	25	28
pmd	8.4	858	898	584	371
sunflow	32.5	4	8	4	4
tomcat	14.1	0	0	0	1
tradebeans	45.7	51	1	126	0
tradesoap	95.0	12	0	16	0
xalan	20.9	38	52	27	53

Table 5.6: Average *percent* runtime overhead for Previous JavaMOP (JavaMOP 2011) and Scalable JavaMOP (JavaMOP 2012) against 58 specifications from java.util (convergence within 3%, N/A: instrumentation crashes)

		All			
# of specs		137			
		Previous		Scalable	
	Orig (sec)	Sum	Together	Sum	Together
bloat	14.4	1498	1950	1059	886
avrora	13.9	494	364	291	182
batik	3.5	50	40	41	24
eclipse	79.5	0	-2	49	-1
fop	2.0	688	N/A	450	N/A
h2	18.7	88	73	117	55
jython	13.6	149	121	103	83
luindex	2.9	17	9	28	7
lusearch	25.3	71	75	63	59
pmd	8.4	855	988	584	394
sunflow	32.5	4	10	4	5
tomcat	14.1	0	0	0	1
tradebeans	45.7	124	-1	163	0
tradesoap	95.0	28	0	36	-1
xalan	20.9	27	34	-12	23

Table 5.7: Average *percent* runtime overhead for Previous JavaMOP (JavaMOP 2011) and Scalable JavaMOP (JavaMOP 2012) against 137 specifications from java.io, java.lang, and java.util (convergence within 3%, N/A: instrumentation crashes)

		java.io			
# of specs		30			
		Previous		Scalable	
	Orig	Sum	Together	Sum	Together
bloat	4.9	5.0	5.7	5.0	5.5
avrora	4.7	4.7	4.5	4.6	4.4
batik	77.3	77.3	76.3	77.3	79.2
eclipse	101.0	101.0	100.0	101.0	99.4
fop	23.9	22.9	25.8	25.4	25.9
h2	267.1	267.1	265.3	267.1	260.9
jython	21.9	22.1	23.0	21.9	22.9
luindex	6.8	5.7	7.9	5.7	7.0
lusearch	4.6	4.4	4.7	4.0	4.6
pmd	22.3	22.3	25.1	22.3	26.3
sunflow	4.5	4.5	4.5	4.5	5.0
tomcat	11.7	11.7	11.7	11.7	12.3
tradebeans	62.9	64.3	63.3	67.1	63.2
tradesoap	63.9	63.9	64.2	63.9	64.1
xalan	4.9	4.9	5.0	4.9	4.9

Table 5.8: Peak memory usage (in MB) for Previous JavaMOP (JavaMOP 2011) and Scalable JavaMOP (JavaMOP 2012) against 30 specifications from java.io (during 5 iterations, N/A: instrumentation crashes)

		java.lang			
# of specs		49			
		Previous		Scalable	
	Orig	Sum	Together	Sum	Together
bloat	4.9	559.2	626.5	628.0	627.8
avrora	4.7	10.9	12.3	7.9	12.5
batik	77.3	77.3	75.1	77.3	72.5
eclipse	101.0	101.0	103.2	101.0	109.1
fop	23.9	79.0	73.2	49.6	58.0
h2	267.1	303.5	327.1	317.4	357.7
jython	21.9	57.0	76.1	78.6	86.5
luindex	6.8	8.1	18.8	8.0	19.7
lusearch	4.6	4.4	7.5	5.1	7.0
pmd	22.3	87.1	38.5	46.8	38.0
sunflow	4.5	4.5	6.6	4.5	6.3
tomcat	11.7	11.7	11.8	11.7	12.3
tradebeans	62.9	66.6	63.1	66.0	63.1
tradesoap	63.9	69.6	62.1	67.3	64.4
xalan	4.9	20.4	21.4	22.2	21.9

Table 5.9: Peak memory usage (in MB) for Previous JavaMOP (JavaMOP 2011) and Scalable JavaMOP (JavaMOP 2012) against 49 specifications from java.lang (during 5 iterations, N/A: instrumentation crashes)

		java.util			
# of specs		58			
		Previous		Scalable	
	Orig	Sum	Together	Sum	Together
bloat	4.9	330.2	1011.2	112.1	171.8
avrora	4.7	44.8	73.1	33.8	56.2
batik	77.3	99.2	166.2	89.1	89.5
eclipse	101.0	101.0	113.8	101.0	99.2
fop	23.9	341.4	402.6	232.6	117.7
h2	267.1	2307.9	1176.0	1363.2	475.5
jython	21.9	91.8	191.8	85.4	61.5
luindex	6.8	6.4	8.8	6.7	12.8
lusearch	4.6	4.8	4.5	5.0	4.9
pmd	22.3	430.5	1474.9	371.3	175.8
sunflow	4.5	4.7	5.5	4.3	4.5
tomcat	11.7	11.7	11.4	11.7	11.7
tradebeans	62.9	66.3	63.0	63.6	62.8
tradesoap	63.9	68.3	65.4	66.0	62.7
xalan	4.9	4.9	5.0	5.0	4.9

Table 5.10: Peak memory usage (in MB) for Previous JavaMOP (JavaMOP 2011) and Scalable JavaMOP (JavaMOP 2012) against 58 specifications from java.util (during 5 iterations, N/A: instrumentation crashes)

		All			
# of specs		137			
		Previous		Scalable	
	Orig	Sum	Together	Sum	Together
bloat	4.9	884.6	≥1500	735.3	1295.9
avrora	4.7	51.0	737.2	36.9	65.0
batik	77.3	99.2	166.7	89.1	92.1
eclipse	101.0	101.0	108.0	101.0	102.6
fop	23.9	395.5	N/A	259.8	N/A
h2	267.1	2344.3	1343.5	1413.5	845.6
jython	21.9	127.1	240.2	142.1	91.7
luindex	6.8	6.6	20.8	6.8	20.8
lusearch	4.6	4.4	7.4	4.9	8.2
pmd	22.3	495.3	1457.4	395.8	254.9
sunflow	4.5	4.7	7.5	4.3	7.1
tomcat	11.7	11.7	11.9	11.7	11.7
tradebeans	62.9	71.4	63.1	70.9	63.1
tradesoap	63.9	74.0	64.7	69.4	63.6
xalan	4.9	20.4	22.9	22.3	25.5

Table 5.11: Peak memory usage (in MB) for Previous JavaMOP (JavaMOP 2011) and Scalable JavaMOP (JavaMOP 2012) against 137 specifications from java.io, java.lang, and java.util (during 5 iterations, N/A: instrumentation crashes)

Tables 5.4, 5.5, 5.6, and 5.7 show the average percent runtime overhead of the two versions of JavaMOP. They shows the sum of overheads for monitoring each specification individually, and the overhead of monitoring them simultaneously, for each benchmark. To avoid the error accumulation, we exclude overheads under 3% for the summation. Overall, Scalable JavaMOP (JavaMOP 2012) shows significantly less runtime overhead than JavaMOP 2011. In monitoring multiple specifications, JavaMOP 2011 shows higher overheads than the sum of overheads in many places. This is because heavy memory pressure from multiple specifications triggers garbage collection more often. However, Scalable JavaMOP shows much less overhead than the sum of overheads in most cases.

JavaMOP 2011 shows 1950% overhead when monitoring all 137 specifications for bloat, while the sum of overheads is 1498%. For pmd, it shows 988% overhead when all specifications are monitored, while the sum of overheads is 855%. However, Scalable JavaMOP shows 886% and 394% overheads for bloat and pmd, respectively, when all specifications are monitored. These overheads are less than half of what the previous version showed. Also, they are less than the sums of overheads in the Scalable JavaMOP, which are 1059% and 584%, respectively. Note that Scalable JavaMOP also improves the runtime performance of monitoring a single specification, resulting in smaller sums.

Tables 5.8, 5.9, 5.10, and 5.11 summarize the peak memory usage during 5 iterations. In a similar way to the runtime result, they show the sum of memory overheads from monitoring each specification individually. In the sum of the peak memory usage, the original peak memory usage is counted only once. For example, on bloat, which shows 4.9MB peak memory usage, if two specifications show 5.5MB and 6.2MB peak memory usage, respectively, the sum of peak memory usage is 6.8MB. Overall, JavaMOP 2012 shows significantly less memory overhead than JavaMOP 2011. Similar to runtime performance, JavaMOP 2012 uses less memory, not only when monitoring multiple specifications simultaneously, but also when monitoring them individually. In monitoring specifications individually, in total, JavaMOP 2012 uses about 28% less memory than JavaMOP 2011. In monitoring multiple specifications simultaneously, for avrora and pmd, JavaMOP 2012 shows 11.3 times and 5.7 times less peak memory usage than the JavaMOP 2011, respectively. Also, in total, JavaMOP 2012 uses about 49% less memory than JavaMOP 2011, in monitoring multiple specifications simultaneously.

Monitoring a large number of specifications shows different memory usage from monitoring a single specification. During monitoring process, a large number of objects is generated for the purposes of monitoring. Many of these monitoring objects must be garbage collected. Since the JVM controls the garbage collection throughput so that it does not overwhelm the entire execution time, the garbage collection might not be able to clean up all garbage objects on time. This can cause parameter objects to live longer than usual, delaying accompanied monitoring resources from being garbage collected. In this case, the JVM simply consumes more memory as long as there is more space left. After reaching the memory limit, it starts spending more time for garbage collection. This explains for bloat and others, why monitoring multiple specifications simultaneously shows more peak memory usage than the sum of peak memory usages of individual monitoring and the sum of peak memory usages of monitoring specifications in each package. For example, for bloat and the Scalable JavaMOP, monitoring all specifications in `java.io`, `java.lang`, and `java.util` shows 5.5MB, 627.8MB, and 171.8MB, but monitoring all of the specifications simultaneously shows 1295.9MB memory usage at peak.

It is also interesting to see how much our Scalable JavaMOP improves the runtime and memory performance of monitoring a single specification, compared to the previous version of JavaMOP (JavaMOP 2011) which already implements all optimization techniques presented in Chapter 4. Tables 5.12, 5.13, 5.14, 5.15, 5.16, and 5.17 are the tables from Chapter 4, with additional columns named "Scale" to show the performance changes by scalable techniques presented in this chapter. While JavaMOP 2012 adds a bit more overhead in a few cases, it shows a great improvement on many other cases. Especially, for UnsafeSyncColl and UnsafeSyncMap, it shows less than 25% runtime overhead on bloat, which is about 30 times and 45 times faster than JavaMOP 2011 before the dissertation, respectively, and about 9 times and 7 times faster than the optimized version from Chapter 4, respectively. On average, the Scalable JavaMOP shows 16.6% runtime overhead, while JavaMOP 2011 shows 54.4% and the optimized version from Chapter 4 shows 20.9%. Also, as for peak memory usage, the JavaMOP 2012 shows 45.5% memory overhead, while JavaMOP 2011 shows 140.9% and the optimized version from Chapter 4 shows 65.0%. Tracematches shows, on average, 309.1% runtime overhead and 17.2% memory overhead.

Overall, the Scalable JavaMOP (JavaMOP 2012) shows, on average, about 3 times faster runtime performance with about 3 times less peak memory usage then JavaMOP 2011 and about 19 times faster runtime performance and compatible (3 times more) memory overhead compared to Tracematches, ignoring the fact that both Tracematches and JavaMOP 2011 crashed on several benchmarks. Also, in monitoring multiple specification simultaneously, JavaMOP 2012 shows less than half of runtime overhead and 49% less memory overhead than the already optimized version of JavaMOP (JavaMOP 2011) from Chapter 4.

5.4 Ineffectual Approaches

In this section, we discuss some ineffectual approaches that we have tried while improving the scalability of parametric monitoring. Although they turn out to be ineffectual in parametric monitoring, some of them might be useful in different settings or they might inspire new effectual ideas.

Combining Indexing Trees between Specifications As mentioned in Section 5.2.3, we combine indexing trees only within each specification. If we

	ORIG	HasNext				UnsafeIter			
	(sec)	TM	Old	New	Scale	TM	Old	New	Scale
antlr	3.6	-1	4	-2	1	0	0	-2	-1
bloat	14.4	2119	448	116	146	11258	769	251	269
chart	12.2	0	0	-2	-4	11	5	-1	-1
hsqldb	8.4	15	0	-3	-4	17	-1	-3	-4
jython	9.0	13	0	0	3	11	-4	1	7
avrora	13.9	45	48	55	62	637	298	118	110
batik	3.5	3	4	3	2	355	11	8	11
eclipse	79.5	-2	1	-1	1	0	2	-1	1
fop	2.0	200	57	48	64	350	23	13	17
h2	18.7	89	17	13	17	128	7	4	3
luindex	2.9	0	1	1	1	0	1	1	2
lusearch	25.3	-1	7	0	1	1	0	2	1
pmd	8.4	176	89	59	77	1423	162	123	121
sunflow	32.5	47	5	3	4	7	0	0	1
tomcat	14.1	8	-1	1	-1	37	-1	1	-1
tradebeans	45.7	0	1	1	0	1	0	2	1
tradesoap	95.0	1	0	0	-1	2	-2	1	0
xalan	20.9	4	-2	2	-1	27	2	2	-2

Table 5.12: Average *Percent* Runtime Overhead for Tracematches(TM), Previous JavaMOP(Old), and optimized JavaMOP(New) against HasNext and UnsafeIter (convergence within 3%, OOM = Out of Memory)

	ORIG	UnsafeMapIter				UnsafeSyncColl			
	(sec)	TM	Old	New	Scale	TM	Old	New	Scale
antlr	3.6	-2	5	1	2	-1	2	-1	1
bloat	14.4	OOM	OOM	178	150	1359	735	212	24
chart	12.2	-1	4	-2	-2	-2	1	-1	-1
hsqldb	8.4	29	0	-3	-4	9	0	-2	-4
jython	9.0	150	11	3	5	11	-4	1	2
avrora	13.9	OOM	OOM	42	44	75	140	80	6
batik	3.5	OOM	65	5	3	208	444	9	7
eclipse	79.5	5	-1	0	1	-4	-1	1	1
fop	2.0	OOM	OOM	14	38	OOM	OOM	25	47
h2	18.7	1350	OOM	6	10	868	69	4	11
luindex	2.9	1	0	1	0	1	1	1	0
lusearch	25.3	2	2	0	-1	4	0	1	0
pmd	8.4	OOM	OOM	188	85	1818	OOM	76	58
sunflow	32.5	9	6	1	0	13	6	5	1
tomcat	14.1	3	-1	1	-1	2	-1	1	-1
tradebeans	45.7	5	-1	-1	-1	-1	-1	2	-1
tradesoap	95.0	2	0	1	2	0	0	1	-1
xalan	20.9	10	1	2	2	3	1	3	-2

Table 5.13: Average *Percent* Runtime Overhead for Tracematches(TM), Previous JavaMOP(Old), and optimized JavaMOP(New) against UnsafeMapIter and UnsafeSyncColl (convergence within 3%, OOM = Out of Memory)

	ORIG	UnsafeSyncMap			
	(sec)	TM	Old	New	Scale
antlr	3.6	0	2	0	2
bloat	14.4	1942	858	130	19
chart	12.2	-2	3	-2	-2
hsqldb	8.4	7	-1	-3	-5
jython	9.0	10	-4	0	2
avrora	13.9	54	73	16	0
batik	3.5	5	7	0	1
eclipse	79.5	OOM	2	-1	1
fop	2.0	OOM	OOM	19	42
h2	18.7	83	25	5	0
luindex	2.9	2	2	0	1
lusearch	25.3	3	1	1	-1
pmd	8.4	OOM	OOM	26	48
sunflow	32.5	17	8	6	3
tomcat	14.1	2	-1	3	-1
tradebeans	45.7	3	2	5	0
tradesoap	95.0	2	0	5	-1
xalan	20.9	4	-2	3	-2

Table 5.14: Average *Percent* Runtime Overhead for Tracematches(TM), Previous JavaMOP(Old), and optimized JavaMOP(New) against UnsafeSyncMap (convergence within 3%, OOM = Out of Memory)

	ORIG	HasNext				UnsafeIter			
	(MB)	TM	Old	New	Scale	TM	Old	New	Scale
antlr	4.3	4.4	4.1	3.8	4.1	4.8	4.0	4.5	4.1
bloat	4.9	40.3	19.3	13.9	15.7	7.8	175.4	79.0	86.0
chart	17.0	17.4	17.3	17.0	16.3	16.9	16.5	17.2	17.5
hsqldb	136.5	136.1	136.7	137.6	142.2	139.1	136.8	137.6	141.9
jython	4.9	5.1	4.7	4.8	4.7	5.5	5.1	5.0	4.3
avrora	4.7	4.6	12.1	9.1	10.0	4.4	114.0	15.8	18.6
batik	77.3	79.2	81.9	79.3	80.0	75.2	93.4	86.6	86.3
eclipse	101.0	100.8	104.0	97.1	98.1	98.3	100.3	110.3	107.4
fop	23.9	97.4	47.1	52.5	64.7	24.3	25.6	29.4	27.3
h2	267.1	267.8	588.8	565.2	702.6	267.2	267.5	262.4	268.6
luindex	6.8	5.6	6.7	5.6	7.3	6.3	7.4	6.8	7.2
lusearch	4.6	4.7	4.6	4.8	4.6	4.6	4.3	4.2	4.7
pmd	22.3	56.9	65.5	48.5	59.0	17.2	147.2	86.4	114.0
sunflow	4.5	4.5	4.8	4.9	4.4	4.8	4.6	4.7	4.8
tomcat	11.7	11.4	11.6	11.4	11.9	12.5	11.8	11.5	12.1
tradebeans	62.9	62.9	62.4	62.1	62.6	63.7	63.9	64.1	62.8
tradesoap	63.9	61.8	64.8	63.3	63.0	63.4	64.7	64.4	65.7
xalan	4.9	4.9	5.0	5.1	4.9	4.9	5.0	4.9	4.9

Table 5.15: Peak memory usage (in MB) for Tracematches(TM), Previous JavaMOP(Old), and optimized JavaMOP(New) (during 5 iterations, OOM = Out of Memory)

	ORIG	UnsafeMapIter				UnsafeSyncColl			
	(MB)	TM	Old	New	Scale	TM	Old	New	Scale
antlr	4.3	4.1	4.0	4.6	3.9	4.1	4.2	4.2	4.7
bloat	4.9	OOM	OOM	56.7	14.7	6.7	100.0	48.3	5.0
chart	17.0	16.6	15.9	19.2	17.2	17.0	16.4	17.2	17.1
hsqldb	136.5	136.0	140.0	136.8	140.4	136.1	146.2	146.3	136.3
jython	4.9	6.1	20.9	5.1	4.6	5.3	4.9	5.4	4.8
avrora	4.7	OOM	OOM	8.5	7.5	4.3	18.4	12.6	4.7
batik	77.3	OOM	173.8	79.6	78.7	78.2	180.7	85.1	75.8
eclipse	101.0	106.9	198.9	101.1	89.3	100.4	115.1	90.1	94.5
fop	23.9	OOM	OOM	28.1	33.9	OOM	OOM	24.8	35.3
h2	267.1	312.4	OOM	268.2	382.9	271.4	1456.7	265.5	382.3
luindex	6.8	7.4	6.8	6.9	6.8	7.4	7.5	7.5	8.0
lusearch	4.6	4.0	4.2	4.8	4.7	4.5	4.3	4.6	4.7
pmd	22.3	OOM	OOM	93.6	60.4	20.3	OOM	84.6	43.8
sunflow	4.5	4.7	4.6	4.4	4.9	5.1	4.4	4.9	4.8
tomcat	11.7	11.9	12.0	11.0	11.7	11.3	11.9	11.3	11.8
tradebeans	62.9	63.3	62.4	62.7	63.5	63.2	62.8	62.0	64.0
tradesoap	63.9	64.1	65.4	62.0	64.1	60.7	64.1	65.9	63.7
xalan	4.9	4.9	4.9	4.9	4.8	5.0	4.7	5.0	4.9

Table 5.16: Peak memory usage (in MB) for Tracematches(TM), Previous JavaMOP(Old), and optimized JavaMOP(New) (during 5 iterations, OOM = Out of Memory)

	ORIG	UnsafeSyncMap			
	(MB)	TM	Old	New	Scale
antlr	4.3	4.6	4.4	4.9	4.8
bloat	4.9	6.9	25.8	12.3	6.5
chart	17.0	17.4	16.4	17.1	17.1
hsqldb	136.5	142.1	136.4	137.0	145.5
jython	4.9	5.8	5.0	5.1	4.7
avrora	4.7	4.4	12.4	4.9	4.6
batik	77.3	79.9	84.8	76.7	79.2
eclipse	101.0	OOM	102.3	98.7	98.8
fop	23.9	OOM	OOM	25.2	31.5
h2	267.1	271.0	688.2	270.0	261.3
luindex	6.8	7.1	7.3	11.0	6.7
lusearch	4.6	4.6	4.8	4.7	4.8
pmd	22.3	OOM	OOM	32.9	34.5
sunflow	4.5	4.5	4.8	4.5	4.7
tomcat	11.7	11.4	11.3	11.8	11.2
tradebeans	62.9	64.0	62.7	64.0	63.7
tradesoap	63.9	65.5	65.1	65.6	63.8
xalan	4.9	5.1	5.1	4.9	5.0

Table 5.17: Peak memory usage (in MB) for Tracematches(TM), Previous JavaMOP(Old), and optimized JavaMOP(New) (during 5 iterations, OOM = Out of Memory)

combine indexing trees for different specifications, as well, we can reduce the number of indexing trees even more. However, there is a lot of wasted space in the combined indexing tree. For example, an indexing tree A maps p_1 to m_1 and p_2 to m_2, and another indexing tree B maps p_2 to m_3 and p_3 to m_4. The combined indexing tree of A and B will map p_1 to $(m_1, \varnothing)$, p_2 to (m_2, m_3), and p_3 to $(\varnothing, m_4)$. All empty spaces indicated by $\varnothing$ will be wasted while the indexing trees A and B do not have empty space. More memory overhead from wasted space triggers more garbage collection, slowing down the monitoring.

Enhanced Indexing Cache The indexing cache provides faster retrieval of monitors from the indexing tree. There are several ideas to improve its hit ratio. We can apply a multi-entry cache from Section 5.2.2. Also, we can cache not only monitors but also lack thereof to save searching the indexing tree for nothing. However, since the indexing cache provides already a high hit ratio and the cost to access the indexing tree is already decreased by the `GlobalWeakRefTable`, these enhancements to the indexing cache do not improve the performance. Certainly those ideas increase cache hit ratio, but their benefits are cancelled out by the overheads necessary to support them.

Indexing Tree Cleaning by GlobalWeakRefTables Since we can manage all weak references for each parameter type in one place, the `GlobalWeakRefTable`, we can let the `GlobalWeakRefTable` clean up the indexing trees. In this way, we can remove garbage collected parameter objects from all indexing trees at once, eliminating the need for partial cleanups. Note that partial cleanups could occur even when there is no garbage collected parameter object. We can also have a bit map in the weak reference to indicate to which indexing trees the referent belongs so that we need check only the indexing trees that actually contain it. However, this approach only moves cleanup costs from indexing trees to the `GlobalWeakRefTable`, showing no improvement. The cleanup by the `GlobalWeakRefTable` is more effective because it knows which weak references should be removed. However, cleaning up from outside of the indexing tree costs more because we must locate the entry before we can remove it.

Statistics-Based Indexing Tree Cleaning As mentioned previously, partial cleanups at indexing trees can occur even when there are no garbage collected parameter objects. Since we have the `GlobalWeakRefTable`, we can keep statistics about garbage collected parameter objects and use it for deciding whether to trigger a partial cleanup. However, in most cases, there are garbage collected parameter objects. Saving a relatively small number of partial cleanups does not compensate the overhead necessary.

Event Activator Similar to the specification activator (Section 5.2.5), non-creation events can be skipped if there is no monitor created for the parameter of the event. However, this approach does not improve the performance because the specification activator already works effectively and the `GlobalWeakRefTable` already returns no weak reference if there was no creation event for the parameter object. Thus, this approach only introduces an overhead of maintaining activators (boolean variables), although the overhead is too small to be notable.

5.5 Discussion

Parametric monitoring is a technique for improving the reliability of software that has received an ever increasing amount of attention. Previous work on parametric monitoring has focused on the performance of monitoring single properties in isolation. Realistic uses of monitoring, however, involve monitoring many properties simultaneously, as the large number of properties from [59] can attest. In this chapter we have improved the efficiency of JavaMOP with respect to monitoring multiple simultaneous properties; as an added bonus, we also improved performance in the case of a single property. We preformed a thorough analysis of the remaining bottlenecks in the JavaMOP system, and we addressed those that could be addressed without adding more runtime overhead than they save. The cases that were ineffectual, presented in this chapter, show that sometimes it is more expensive to address an inefficiency than to let it be. The remaining cases produced real, tangible performance enhancements, in some cases halving overhead in a system that was already heavily optimized.

Chapter 6

Multi-Threaded Unit Testing

As runtime monitoring becomes more practical in the dissertation, we expect more people to use it in a wider spectrum of application domains. Here we present our work on searching for real world applications for runtime monitoring. We apply runtime monitoring in our new unit testing framework for multi-threading environment, to monitor and enforce thread scheduling as specified by the user.

6.1 Improved Multi-threaded Unit Testing

Multi-threaded code is notoriously hard to develop and test. A multi-threaded unit test exercises the code with two or more threads. Each test execution follows some schedule/interleaving of the multiple threads, and different schedules can give different results. Developers often want to enforce a particular schedule for test execution, and to do so, they use time delays (Thread.sleep in Java). Unfortunately, this approach can produce false positives or negatives, and can result in unnecessarily long testing time. There have been many researches tackling some problems in specifying and enforcing schedules in multi-threaded unit testing. However, despite these researches, multi-threaded unit testing still has many issues including readability, modularity, reliability, schedule language, and so on.

To solve these issues, we develop the improved multi-threaded unit testing (IMUnit) framework. We first introduce a new language that allows explicit specification of schedules as orderings on events encountered during test execution. By describing schedules explicitly, developers can focus more on functionality testing while writing unit tests, and pay less attention to reasoning about the execution of threads. Also, this approach has good modularity since the intended schedule is not intermixed with the test code, and it is much easier to specify multiple schedules for a particular unit test. Then, the specified schedules are checked and enforced by the runner in IMUnit, on help of JavaMOP.

When we execute unit tests, we monitor thread schedules and enforce the intended schedule using JavaMOP. JavaMOP monitors thread schedules by observing the order of events in each thread and checks if the current thread can proceed without violating the intended schedule. If a proceeding is expected to violate the intended schedule, JavaMOP blocks the thread until it is okay for the thread to proceed. Sleep-based multi-threaded unit tests are unreliable mainly because they rely on real time – this often leads to false positives/negatives and/or slow testing time. Our approach, on the other hand, is more reliable and achieves faster testing time.

We first give an example in Section 6.1.1, then introduces our schedule language that enables natural and explicit specification of schedules, in Section 6.1.2. Section 6.1.3 discuss how IMUnit enforces/checks schedules by using JavaMOP, and Section 6.1.4 evaluates our approach.

6.1.1 Example

We illustrate improved multi-threaded unit testing (IMUnit) with the help of an example multi-threaded unit test for the ArrayBlockingQueue class in `java.util.concurrent` (JSR-166) [49]. ArrayBlockingQueue is an array-backed implementation of a bounded blocking queue. One operation provided by ArrayBlockingQueue is `add`, which performs a non-blocking insertion of the given element at the tail of the queue. If `add` is performed on a full queue, it throws an exception. Another operation provided by ArrayBlockingQueue is `take`, which removes and returns the object at the head of the queue. If `take` is performed on an empty queue, it blocks until an element is inserted into the queue. These operations could have bugs that get triggered when the `add` and `take` operations execute on different threads. Consider testing some scenarios for these operations (in fact, the JSR-166 TCK provides over 100 tests for various scenarios for similar classes).

Figure 6.1 shows a multi-threaded unit test that ArrayBlockingQueue exercises `add` and `take` in two scenarios. In particular, Figure 6.1(a) shows the test written as a regular JUnit test method, with sleeps used to specify the required schedule. We invite the reader to consider what scenarios are specified with that test (without looking at the other figures). It is likely to be difficult to understand which schedule is being exercised by reading the code of this unit test. While the sleeps provide hints as to which thread is waiting for another thread to perform operations, it is unclear which operations are intended to be performed by the other thread before the sleep finishes.

The test actually checks that `take` performs correctly both with and without blocking, when used with `add` from another thread. To check both scenarios, the test exercises a particular schedule where the first `add` operation finishes before the first `take` operation starts, and the second `take` operation blocks before the second `add` operation starts. Line 13 shows the first sleep that is intended to pause the `main` thread[1] while the `addThread` finishes the first `add` operation. Line 9 shows the second sleep which is intended to pause the `addThread` while the `main` thread finishes the first `take` operation and then proceeds to block while

[1] JVM names the thread that starts the execution `main` by default, although the name can be changed later.

```
@Test
public void testTakeWithAdd() {
  ArrayBlockingQueue<Integer> q;
  q = new ArrayBlockingQueue<Integer>(1);
  new Thread(
    new CheckedRunnable() {
      public void realRun() {
        q.add(1);
        Thread.sleep(100);
        q.add(2);
      }
    }, ”addThread”).start();
  Thread.sleep(50);
  Integer taken = q.take();
  assertTrue(taken == 1 && q.isEmpty());
  taken = q.take();
  assertTrue(taken == 2 && q.isEmpty());
  addThread.join();
}
```

(a) JUnit

```
public class TestTakeWithAdd
    extends MultithreadedTest {
  ArrayBlockingQueue<Integer> q;
  @Override
  public void initialize() {
    q = new ArrayBlockingQueue<Integer>(1);

  }
  public void addThread() {
    q.add(1);
    waitForTick(2);
    q.add(2);
  }
  public void takeThread() {
    waitForTick(1);
    Integer taken = q.take();
    assertTrue(taken == 1 && q.isEmpty());
    taken = q.take();
    assertTick(2);
    assertTrue(taken == 2 && q.isEmpty());
  }
}
```

(b) MultithreadedTC

```
@Test
@Schedule(”finishedAdd1 -> startingTake1,
                [startingTake2}] -> startingAdd2”)
public void testTakeWithAdd() {
  ArrayBlockingQueue<Integer> q;
  q = new ArrayBlockingQueue<Integer>(1);
  new Thread(
    new CheckedRunnable() {
      public void realRun() {
        q.add(1);
        @Event(”finishedAdd1”)
        @Event(”startingAdd2”)
        q.add(2);
      }
    }, ”addThread”).start();
  @Event(”startingTake1”)
  Integer taken = q.take();
  assertTrue(taken == 1 && q.isEmpty());
  @Event(”startingTake2”)
  taken = q.take();
  assertTrue(taken == 2 && q.isEmpty());
  addThread.join();
}
```

(c) IMUnit

Figure 6.1: Example multi-threaded unit test for ArrayBlockingQueue

performing the second `take` operation. If the specified schedule is not enforced during the execution, there may be a false positive/negative. For example, if both `add` operations execute before a `take` is performed, the test will throw an exception and fail even if the code has no bug, and if both `take` operations finish without blocking, the test will not fail, even if the blocking `take` code had a bug.

Figure 6.1(b) shows the same test written using MultithreadedTC [68]. Note that it departs greatly from traditional JUnit where each test is a method. In MultithreadedTC, each test has to be written as a class, and each method in the test class contains the code executed by a thread in the test. The intended schedule is specified with respect to a global, logical clock. Since this clock measures time in *ticks*, we call the approach tick-based. When a thread executes a `waitForTick` operation, it is blocked until the global clock reaches the required tick. The clock advances implicitly by one tick when all threads are blocked (and at least one thread is blocked in a `waitForTick` operation). While a MultithreadedTC test does not rely on real time, and is thus more reliable than a sleep-based test, the intended schedule is still not immediately clear upon reading the test code. It is especially not clear when `waitForTick` operations are blocked/unblocked, because ticks are advanced implicitly when all the threads are blocked.

Figure 6.1(c) shows the same test written using IMUnit. The interesting events encountered during test execution are marked with `@Event` annotations, and the intended schedule is specified with a `@Schedule` annotation that contains a comma-separated set of *orderings* among events. Note that `@Event` annotations appear on statements. The current version of Java (version 6) does not support annotations on statements, but the upcoming version of Java (version 7) will add such support. For now the time being, `@Event` annotations can be written as comments, e.g., `/* @Event("finishedAdd1") */`, which IMUnit translates into code for test execution. Since `@Schedule` annotations appear on methods, they are already fully supported in the current version of Java. An ordering is specified using the binary operator `->`, where intuitively the left is intended to execute before the right . An specified within square brackets denotes that the thread executing that event is intended to block after that event. It should be clear from reading the schedule that the `addThread` should finish the first `add` operation before the `main` thread starts the first `take` operation, and that the `main` thread should block while performing the second `take` operation before the `addThread` starts the second `add` operation.

6.1.2 Schedule Language

We now describe the syntax and semantics of the language for describing desired schedules in IMUnit.

```
<Schedule> ::= { <Ordering> [","] } <Ordering>
<Ordering> ::= <Condition> "->" <Basic Event>
<Condition> ::= <Basic Event> | <Block Event>
              | <Condition> "||" <Condition>
              | <Condition> "&&" <Condition>
              | "(" <Condition> ")"
<Basic Event> ::= <Event Name> ["@" <Thread Name>]
              | "start" "@" <Thread Name>
              | "end" "@" <Thread Name>
<Block Event> ::= "[" <Basic Event> "]"
<Event Name> ::= { <Id> "." } <Id>
<Thread Name> ::= <Id>
```

Figure 6.2: Syntax of the IMUnit schedule language

Concrete Syntax

Figure 6.2 shows the concrete syntax of the implemented IMUnit schedule language. An IMUnit schedule is a comma-separated set of *orderings*. Each ordering defines a condition that must hold before a basic event can take place. A *basic event* is an event name possibly tagged with its issuing thread name when that is not understood from the context. An *event name* is any identifier, possibly prefixed with a qualified class name. There are two implicit event names for each thread, `start` and `end`, indicating when the thread starts and when it terminates. Any other event must be explicitly introduced by the user with the `@Event` annotation (see Figure 6.1(c)). A *condition* is a conjunctive/disjunctive combination of basic and block events, where block events are written as basic events in square brackets. A *block event* $[e']$ in the condition c of an ordering $c \to e$ states that e' must precede e and, additionally, the thread of e' is blocked when e takes place.

Schedule Logic

It is more convenient to define a richer logic than what is currently supported by our IMUnit implementation; the additional features are natural and thus can also be implemented in the future. The semantics of our logic is given in Section 6.1.2; here is its syntax:

$$\begin{array}{rcl}
a & ::= & \mathit{start} \mid \mathit{end} \mid \mathit{block} \mid \mathit{unblock} \mid \text{event names} \\
t & ::= & \text{thread names} \\
e & ::= & a@t \\
\varphi & ::= & [t] \quad | \quad \varphi \to \varphi \quad | \quad \text{usual propositional connectives}
\end{array}$$

The intuition for $[t]$ is "thread t is blocked" and for $\varphi \to \psi$ "if ψ held in the past, then φ must have held at some moment before ψ". We call these two temporal operators the *blockness* and the *ordering* operators, respectively. For uniformity, all events are tagged with their thread. There are four implicit events: $\mathit{start}@t$ and $\mathit{end}@t$ were discussed above, and $\mathit{block}@t$ and $\mathit{unblock}@t$ correspond to when t gets blocked and unblocked[2].

[2] It is expensive to explicitly generate *block*/*unblock* events in Java precisely when they occur, because it requires polling the status of each thread; our currently implemented fragment only

For example, the following formula in our logic

$$\begin{aligned} & (a_1@t_1 \wedge ([t_2] \vee (\neg(\mathit{start}(t_2) \rightarrow a_1@t_1)))) \rightarrow a_2@t_2 \\ \wedge \quad & (a_2@t_2 \wedge ([t_1] \vee (\mathit{end}(t_1) \rightarrow a_2@t_2))) \rightarrow a_2@t_2 \end{aligned}$$

says that if event a_2 is generated by thread t_2 then: (1) event a_1 must have been generated before that and, when a_1 was generated, t_2 was either blocked or not started yet; and (2) when a_2 is generated by t_2, t_1 is either blocked or terminated. As explained shortly, every event except for *block* and *unblock* is restricted to appear at most once in any execution trace. Above we assumed that $a_1, a_2 \notin \{\mathit{block}, \mathit{unblock}\}$.

Before we present the precise semantics, we explain how our current IMUnit language shown in Figure 6.2 (whose design was driven exclusively by practical needs) is a smaller fragment of the richer logic. An IMUnit schedule is a conjunction (we use comma instead of $\wedge$) of orderings, and schedules cannot be nested. Since generating *block* and *unblock* events is expensive, IMUnit currently disallows their explicit use in schedules. Moreover, to reduce their implicit use to a fast check of whether a thread is blocked or not, IMUnit also disallows the explicit use of $[t]$ formulas. Instead, it allows *block events* of the form $[a@t]$ (note the square brackets) in conditions. Since negations are not allowed in IMUnit, and since we can show (after we discuss the semantics) that $(\varphi_1 \vee \varphi_2) \rightarrow \psi$ equals $(\varphi_1 \rightarrow \psi) \vee (\varphi_2 \rightarrow \psi)$, we can reduce any IMUnit schedule to a Boolean combination of orderings $\varphi \rightarrow e$, where φ is a conjunction of basic events or block events. All that is left to show is how block events are desugared. Consider an IMUnit schedule $(\varphi \wedge [a_1@t_1]) \rightarrow a_2@t_2$, saying that $a_1@t_1$ and φ must precede $a_2@t_2$ *and* t_1 is blocked when $a_2@t_2$ occurs. This can be expressed as $((\varphi \wedge a_1@t_1) \rightarrow a_2@t_2) \wedge ((a_2@t_2 \wedge [t_1]) \rightarrow a_2@t_2)$, relying on $a_2@t_2$ happening at most once.

Semantics

Our schedule logic is a carefully chosen fragment of *past-time linear temporal logic (PTLTL)* over special well-formed multi-threaded system execution traces.

Program executions are abstracted as finite traces of events $\tau = e_1 e_2 \ldots e_n$. Unlike in conventional LTL, our traces are finite because unit tests always terminate. Traces must satisfy the obvious condition that events corresponding to thread t can only appear while the thread is alive, that is, between *start*@t and *end*@t. Using PTLTL, this requirement states that for any trace τ and any event $a@t$ with $a \notin \{\mathit{start}, \mathit{end}\}$, the following holds:

$$\tau \vDash \neg \diamondsuit (a@t \wedge (\diamondsuit \mathit{end}@t \vee \neg \diamondsuit \mathit{start}@t))$$

needs, through its restricted syntax, to check if a given thread is currently blocked or not, which is fast.

where $\diamonddot$ stands for "eventually in the past". Moreover, except for *block*@*t* and *unblock*@*t* events, we assume that each event appears at most once in a trace. With PTLTL, this says that the following must hold ($\odot$ is "previously"):

$$\tau \models \neg \diamonddot\, (a@t \wedge \odot \diamonddot\, a@t)$$

for any trace τ and any $a@t$ with $a \notin \{block, unblock\}$.

The semantics of our logic is defined as follows:

$$\begin{array}{lll}
e_1 e_2 \ldots e_n \models e & \text{iff} & e = e_n \\
\tau \models \varphi \;\; \wedge/\vee \;\; \psi & \text{iff} & \tau \models \varphi \text{ and/or } \tau \models \psi \\
e_1 e_2 \ldots e_n \models [t] & \text{iff} & (\exists 1 \leq i \leq n)\ (e_i = block@t \text{ and} \\
 & & \qquad (\forall i < j \leq n)\ e_j \neq unblock@t) \\
e_1 e_2 \ldots e_n \models \varphi \rightarrow \psi & \text{iff} & (\forall 1 \leq i \leq n)\ e_1 e_2 \ldots e_i \not\models \psi \text{ or} \\
 & & (\exists 1 \leq i \leq n)\ (e_1 e_2 \ldots e_i \models \psi \text{ and} \\
 & & \qquad\qquad (\exists 1 \leq j \leq i)\ e_1 e_2 \ldots e_j \models \varphi)
\end{array}$$

It is not hard to see that the two new operators $[t]$ and $\varphi \rightarrow \psi$ can be expressed in terms of PTLTL as

$$\begin{array}{rcl}
[t] & \equiv & \neg unblock@t\ \mathcal{S}\ block@t \\
\varphi \rightarrow \psi & \equiv & \boxdot\neg\psi \ \vee\ \diamonddot(\psi \wedge \diamonddot\varphi)
\end{array}$$

where $\mathcal{S}$ stands for "since" and $\boxdot$ for "always in the past".

6.1.3 Enforcing and Checking

We now describe the IMUnit Runner, our tool for enforcing/checking schedules for IMUnit multithreaded unit tests. It is implemented as a custom test runner for the JUnit testing framework. It executes each test for each IMUnit schedule (a test can have multiple schedules) and has two operation modes. In the *active mode*, it controls the thread scheduler to enforce an execution of the test to satisfy the given schedule. Note that this mode avoids the main problem of sleep-based tests, that of false positives and negatives due to the execution of unintended schedules. In the *passive mode*, our tool observes and checks the execution provided by the JVM against the given schedule, without interfering. The passive mode is particularly useful for checking whether executions enforced by the tool for some schedules satisfy other schedules.

The runner is implemented using JavaMOP. As explained in Chapter 2, JavaMOP is generic in the property specification formalism and provides several such formalisms as *logic plugins*, including past-time linear temporal logic (PTLTL). Although our schedule language in IMUnit is a semantic fragment of PTLTL (Section 6.1.2), enforcing PTLTL specifications in their full generality on multithreaded programs is a rather expensive problem.

Instead, we have developed a custom JavaMOP logic plugin for our current IMUnit schedule language from Figure 6.2. This plugin synthesizes a correspond-

```
switch (event) {
case finishedAdd1:
  occurred_finishedAdd1 = true; notifyAll();
case startingTake2:
  thread_startingTake2 = currentThread();
  occurred_startingTake2 = true; notifyAll();
case startingTake1:
  while (!occurred_finishedAdd1)
    wait();
  occurred_startingTake1 = true; notifyAll();
case startingAdd2:
  while (!(occurred_startingTake2 &&
            isBlocked(thread_startingTake2)))
    wait();
  occurred_startingAdd2 = true; notifyAll(); }
```

Figure 6.3: Monitor for the schedule in Figure 6.1(c)

ing monitor that either enforces or checks a given IMUnit schedule, depending on the running mode. The monitor is infused within the test program by means of appropriate instrumentation in such a way that the schedule is enforced or checked at runtime, depending on the mode. Since JavaMOP takes care of all the low-level instrumentation and monitor integration details for us (after a straightforward mapping of IMUnit events into JavaMOP events), we here only briefly discuss our new JavaMOP logic plugin. It takes as input an IMUnit schedule and generates as output a monitor written in pseudo-code; a *Java shell* for this language then turns the monitor into AspectJ code [54], which is further woven into the test program. In the active mode, the resulting monitor enforces the schedule by blocking the violating thread until all the conditions from the schedule are satisfied. In the passive mode, it simply prints an error when its corresponding schedule is violated.

A generated monitor for an IMUnit schedule observes the defined events. When an event e occurs, the monitor checks all the conditions that the event should satisfy according to the schedule, i.e., a Boolean combination of basic events and block events (Figure 6.2). The status of each basic event is maintained by a Boolean variable which is true iff the event occurred in the past. The status of a block event is checked as a conjunction of this variable and its thread's blocked state when e occurs. In the active mode, the thread of e will be blocked until this Boolean expression becomes true. If the condition contains any block event, periodic polling is used for checking thread states. Thus, IMUnit pauses threads only if their events are getting out of order for the schedule. Note that the user may have specified an infeasible schedule, which can cause a deadlock where all threads are paused, waiting for infeasible events. Our runner includes a low-overhead runtime deadlock detection mechanism that detects and reports such deadlocks (and other general deadlocks). This way, IMUnit allows both parallel execution and serialization, depending on the schedule. In the passive mode, the monitor will simply print an error message when any Boolean expression is false.

As an example, Figure 6.3 shows the active-mode monitor generated for the schedule in Figure 6.1(c). When events `finishedAdd1` and `startingTake2` occur, the monitor just sets the corresponding Boolean variables, as there is no condition for those events. For event `startingTake1`, it checks if there was an event `finishedAdd1` in the past by checking the variable `occurred_finishedAdd1`; if not, the thread will be blocked until `finishedAdd1` occurs. For event `startingAdd2`, in addition to checking the Boolean variable for `startingTake2`, it also checks whether the thread of the event `startingTake2` is blocked; if not, the thread of the event `startingAdd2` will be blocked until both conditions are satisfied.

6.1.4 Evaluation

To evaluate the IMUnit contributions—schedule language, automated migration, and schedule execution—we analyzed over 200 sleep-based tests from several open-source projects. Table 6.1 lists the projects and the number of sleep-based tests that we manually migrated to IMUnit. We first describe our experience with the IMUnit language. We then present quantitative results of our inference techniques for migration. We finally discuss the test running time with IMUnit execution.

Schedule Language

It is hard to quantitatively evaluate and compare languages, be it implementation or specification languages, including languages for specifying schedules. One metric we use is how *expressive* the language is, i.e., how many schedules from sleep-based tests can be expressed in IMUnit such that *sleeps can be removed altogether*. Note that IMUnit conceptually subsumes sleeps: sleeps and IMUnit events/schedules can co-exist in the same test, and developers just need to make sleeps long enough to account for the IMUnit schedule execution/enforcement. While every sleep-based test is trivially an IMUnit test, we are interested only in those tests where IMUnit allows removing sleeps altogether.

We were able to remove sleeps from 198 tests, in fact all sleeps from all but 4 tests. While the current version of IMUnit is highly expressive, we have to point out that we refined the IMUnit language based on the experience with migrating the sleep-based tests. When we encountered a case that could not be expressed in IMUnit, we considered how frequent the case is, and how much IMUnit would need to change to support it. For example, blocking events are very frequent, and supporting them required a minimal syntactic extension (adding events with square brackets) to the initial version of our language. However, some cases would require bigger changes but are not frequent enough to justify them. The primary example is events in a loop. IMUnit currently does not support the occurrence of an event more than once in a trace. We did find 4 tests that would require multiple event occurrences, but changing the language to support them (e.g., adding event counters or loop indices to events) would add a layer of complexity that is not justified by the small number of cases. However, as we apply IMUnit to more projects, and gain more experience, we expect that the language could grow in the future.

Subject	Tests	Events	Orderings
Collections [42]	18	51	32
JBoss-Cache [50]	27	105	47
Lucene [12]	2	3	4
Mina [13]	1	2	1
Pool [14]	2	8	3
Sysunit [37]	9	33	34
JSR-166 TCK [49]	139	577	277
$\sum$	198	779	398

Table 6.1: Subject Programs Statistics

Subject	Original [s]	IMUnit [s]		Speedup	
		DDD	DDE	DDD	DDE
Collections	4.96	1.06	1.67	4.68	2.97
JBoss-Cache	65.58	31.25	31.76	2.10	2.06
Lucene	11.02	3.57	6.12	3.09	1.80
Mina	0.26	0.17	0.20	1.53	1.30
Pool	1.43	1.04	1.04	1.38	1.38
Sysunit	17.67	0.35	0.45	50.49	39.27
JSR-166 TCK	15.20	9.56	9.56	1.59	1.59
GeometricMean				3.39	2.76

Table 6.2: Test execution time. DDD - deadlock detection disabled; DDE - deadlock detection enabled

Performance

Table 6.2 shows the execution times of the 198 original, sleep-based tests and the corresponding IMUnit tests (for IMUnit, with deadlock detection both disabled and enabled). We ran the experiments on an Intel i7 2.67GHz laptop with 4GB memory, using Sun JVM 1.6.0_06 and AspectJ 1.6.9. Our goal for IMUnit is to improve readability, modularity, and reliability of multithreaded unit tests, and we did not expect IMUnit execution to be faster than sleep-based execution. In fact, one could even expect IMUnit to be slower because of the additional code introduced by the instrumentation and the cost of controlling schedules. It came as a surprise that IMUnit is faster than sleep-based tests, on average 3.39x. Even with deadlock detection enabled, IMUnit was on average 2.76x faster. This result is with the sleep durations that the original tests had in the code.

We also compared the running time of IMUnit with MultithreadedTC on a common subset of JSR-166 TCK tests that the MultithreadedTC authors translated from sleep-based to tick-based [67]. For these 129 tests, MultithreadedTC was 1.36x faster than IMUnit. Although MultithreadedTC is somewhat faster, it has a much higher migration cost, and in our view, produces test code that is harder to understand and modify than the IMUnit test code. Moreover, we were surprised to notice that running MultithreadedTC on these tests, translated by the MultithreadedTC authors, can result in some failures (albeit with a low probability), which means that these MultithreadedTC tests can be unreliable and lead to false positives in test runs.

6.2 Discussion

The monitoring ability of JavaMOP can be applied to many areas including testing, debugging, security, and verification, with its practicality now. In this chapter, we present the improved multi-threaded unit testing (IMUnit) framework as an application of parametric monitoring. It can explicitly express desired thread schedules for multi-threaded unit tests using the proposed novel schedule language. Then, the IMUnit runner executes the unit tests with enforcing/checking the thread schedules. In this way, we have achieved the reliable and fast multi-threaded unit testing. In this application, JavaMOP provided ease of implementation of monitoring/enforcing thread scheduling, thanks to its modular architecture. As JavaMOP become more expressive, efficient, and scalable, we expect more of real applications in the near future; applying JavaMOP to more real areas is an interesting future research.

Chapter 7

Related Work

In this chapter, we review other parametric monitoring systems and discuss their practicality. A practical monitoring system must be capable of expressing various kinds of specifications (*expressiveness*) and monitoring them efficiently (*efficiency*). Also, it should be able to monitor multiple specifications simultaneously (scalability), however, to the best knowledge of the author, there was no parametric monitoring system which is capable of monitoring a large number of specifications. Therefore, we focus only on *expressiveness* and *efficiency* in this chapter.

As we can see in Table 7.1, all the runtime monitoring systems in the figure work with hardwired formalism. Also, none of them share the exact same logical formalism for expressing properties. This observation strengthens our belief that there is probably *no silver bullet logic* (or *super logic*) that serves all purposes. Besides logical formalism they support, there are three more orthogonal attributes of a runtime monitoring system: scope, running mode, and handlers. The scope determines where to check the property; it can be class invariant, global, interface, etc. The running mode denotes where the monitoring code runs; it can be inline (weaved into the code), online (operating at the same time as the program), outline (receiving events from the program remotely, e.g., over a socket), or offline (checking logged event traces)[1]. The handlers specify what actions to perform under exceptional conditions; there can be violation and validation handlers. It is worth noting that for many logics, violation and validation are not complementary to each other, i.e., the violation of a formula does not always imply the validation of the negation of the formula.

Tracematches [9, 19] enables the programmer to trigger the execution of certain code by specifying a parametric regular pattern of events in a computation trace, where the events are defined over entry/exit of AspectJ pointcuts. When the pattern is matched during the execution, the associated code will be executed. Tracematches are one of the most optimized runtime parametric monitoring systems in terms of memory performance, however, it hardwires its property specification formalism (regular expression only). Also, as shown in Chapters 4 and 5, the average runtime overhead of Tracematches is orders of magnitute higher than that of JavaMOP.

[1] Offline implies outline, and inline implies online.

Approach	Language	Logic	Scope	Mode	Handler
Hawk [38]	Java	Eagle	global	inline	violation
J-Lo [25]	Java	ParamLTL	global	inline	violation
Jass [23]	Java	assertions	global	inline	violation
JavaMaC [56]	Java	PastLTL	class	outline	violation
jContractor [8]	Java	contracts	global	inline	violation
JML [58]	Java	contracts	global	inline	violation
JPaX [44]	Java	LTL	class	offline	violation
P2V [61]	C, C++	PSL	global	inline	validation/ violation
PQL [62]	Java	PQL	global	inline	validation
PTQL [43]	Java	SQL	global	outline	validation
Spec# [20]	C#	contracts	global	inline/ offline	violation
RuleR [22]	Java	RuleR	global	inline	violation
Temporal Rover [40]	*several*	MiTL	class	inline	violation
Tracematches [19]	Java	Reg. Exp.	global	inline	validation

Table 7.1: Runtime Monitoring Breakdown

J-LO [25] is a tool for runtime-checking temporal assertions. These temporal assertions are specified using LTL, and the syntax adopted in J-LO is similar to Tracematches' except that the formulae are written in a different logic. J-LO mainly focuses on checking properties at runtime rather than providing programming support. In J-LO, the temporal assertions are inserted into Java files as annotations that are then compiled into runtime checks. There is no thorough performance evaluation on J-LO available, but in [25], J-LO shows "a slowdown of several orders of magnitude" when monitoring `Iterator`-based properties on a program which is not mentioned. Also, it shows "the relatively low additional overhead" when monitoring the same properties on jHotDraw which does not use `Iterator` frequently. However, JavaMOP does not show any noticeable overhead on jHotDraw against `Iterator`-based properties. As mentioned in [25], "it suggests that there is room for optimization."

Both Tracematches and J-LO support parametric events, i.e., free variables can be used in the event patterns and will be bound to specific values at runtime for matching events. Conceptually, J-LO can be captured by the JavaMOP tool, because LTL is supported by the MOP framework, JavaMOP supports the Java langauge, and J-LO's temporal assertions can be easily translated into JavaMOP specifications that contain only action events and validation handlers.

MaC [56, 55, 57], PathExplorer (PaX) [44], Eagle [21], and RuleR [22] are runtime verification frameworks for logic based monitoring, within which specific tools for Java – Java-MaC, Java PathExplorer, and Hawk [38], respectively – are implemented. All these runtime verification systems work in outline monitoring mode and have hardwired specification languages: MaC uses a specialized language based on interval temporal logic, JPaX supports just LTL, and Eagle adopts a fixed-point logic. Java-MaC and Java PathExplorer integrate monitors via Java bytecode instrumentation, making them difficult to port to other languages.

Temporal Rover [40] is a commercial runtime verification tool based on future time metric temporal logic. It allows programmers to insert formal specifications in programs via annotations, from which monitors are generated. An Automatic Test Generation (ATG) component is also provided to generate test sequences from logic specifications. Temporal Rover and its successor, DB Rover, support both inline and offline monitoring. However, they also have their specification formalisms hardwired and are tightly bound to Java.

Jass [23] is a precompiler which turns the assertion comments into Java code. Besides pre-/post- conditions and class invariants, it also provides refinement checks. The design of trace assertions in Jass is mainly influenced by CSP [45], and the syntax is more like a programming language. jContractor is implemented as a Java library which allows programmers to associate contracts with any Java class or interface. Contract methods can be included directly within the Java class or written as a separate contract class. Before loading each class, jContractor detects the presence of contract code patterns in the Java class bytecode and performs on-the-fly bytecode instrumentation to enable checking of contracts during the program's execution. jContractor also provides a support library for writing expressions using predicate logic quantifiers and operators such as *Forall*, *Exists*, *suchThat*, and *implies*. Using jContractor, the contracts can be directly inserted into the Java bytecode even without the source code.

This 'contracts' approach can be simulated in JavaMOP using a raw specification. While a property written in a logical formalism monitors event patterns, it is also possible to monitor programs without any property. A raw specification is a specification which does not have any property. In a raw specification, there are user-defined events and event actions. With these event actions, one can check pre-/post- conditions.

Java modeling language (JML) [58] is a behavioral interface specification language for Java. It provides a more comprehensive modeling language than DBC extensions. Not all features of JML can be checked at runtime; its runtime checker supports a DBC-like subset of JML. Spec# [20] is a DBC- like extension of the object-oriented language C#. It extends the type system to include non-null types and checked exceptions and also provides method contracts in the form of pre- and post-conditions as well as object invariants. Using the Spec# compiler, one can statically enforce non-null types, emit run-time checks for method contracts and invariants, and record the contracts as metadata for consumption by downstream tools.

Program Query Language (PQL) allows programmers to express design rules that deal with sequences of events associated with a set of related objects [62]. Both static and dynamic tools have been implemented to find solutions to PQL queries. The static analysis conservatively looks for potential matches for queries and is useful to reduce the number of dynamic checks. The dynamic analyzer checks the runtime behavior and can perform user-defined actions when matches are found. PQL has a “hardwired” specification language based on context-free grammars (CFG) and supports only inline monitoring. CFGs can potentially express more complex languages than regular expressions, so in principle PQL can express more complex safety policies than Tracematches. However, in [63], PQL has shown prohibitive runtime overhead. Although the result was without static analysis, the evaluation suggests more optimization in its runtime monitoring.

Program Trace Query Language (PTQL) [43] is a language based on SQL-like relational queries over program traces. The current PTQL compiler, Partiqle, instruments Java programs to execute the relational queries on the fly. PTQL events are timestamped and the timestamps can be explicitly used in queries. PTQL queries can be arbitrarily complex and, as shown in [43], PTQL’s runtime overhead seems acceptable in many cases but we were unable to obtain a working package of PTQL and compare it in our experiments with JavaMOP because of license issues. PTQL properties are globally scoped and their running mode is inline. PTQL provides no support for recovery, its main use being to detect errors.

Chapter 8

Conclusions and Future Work

The goal of the dissertation is to make runtime parametric monitoring practical. The author suggests three perspectives: *expressiveness*, *efficiency*, and *scalability*. The dissertation presents a number of techniques for all three perspective and they are integrated into the JavaMOP framework, resulting in the most expressive (with various formalisms) parametric monitoring system that shows the best runtime performance and competitive memory performance. Also, it is the first parametric monitoring system that is capable of monitoring a large number of specifications *efficiently*.

Chapter 2 provided background on parametric monitoring and the indexing tree technique. The indexing tree technique is a means to locate relevant monitors upon each event. In parametric monitoring, there can be multiple monitors related to an event, since an event can be partially parameterized. In this case, all monitors for the compatible parameter instances to the partial parameter instance of the event, should be updated. However, there are many challenges in implementing efficient indexing trees. First, upon garbage collections of parameters, indexing trees must clean up broken mappings for them to avoid memory leaks. Second, indexing trees should provide efficient mappings to monitors. Third, indexing trees should be capable of storing unlimited number of mappings since there is no limit in the number of parameter instances. Thus, indexing trees should adjust the size of its internal data structures.

Chapters 4 and 5 presented a number of optimization techniques closely related to the indexing tree technique. Instead of general purpose data structures from the Apache Commons Collections Library [42], custom data structures for indexing trees are proposed and implemented, which satisfy the above conditions. On top of these custom data structures, the monitor garbage collection technique and the resource sharing technique for scalability are implemented. Thorough evaluations show that these techniques effectively have reduced runtime and memory overheads.

Chapter 3 introduced parametric monitoring without any limitation that the first events must initiates all parameters. In this way, many specifications including Map_UnsafeIterator from Chapter 2 were able to be expressed in JavaMOP. Also, a new logical formalism for monitoring stack-based properties, called PTCaRet, is efficiently implemented with an optimization to reduce overhead from tracking every method begin and end. Furthermore, the prototype of specification inheritance is suggested. It allows reuse of specifications for more sophisticated structure between specification.

Chapter 6 suggested that JavaMOP can be also used in monitoring/enforcing thread scheduling. JavaMOP was used in the improved multi-threaded unit testing framework (IMUnit) as a part of the prototype implementation for monitoring/enforcing/checking thread scheduling against the desired schedules which are explicitly given by developers. As a result, IMUnit has shown reliable and fast multi-threaded unit testing.

Future Work There are still challenges left for even more practical parametric monitoring. Since JavaMOP is separated from the AspectJ compiler, it is not capable of analyzing target programs for static analysis, code optimization, and it cannot support extended pointcuts. For the broader scope of events and target code specific optimization, we leave integrating JavaMOP into an AspectJ compiler as a future work. Also, formalism-independent static analysis and model-checking based on JavaMOP specifications are also interesting ideas that should be investigated.

Since parametric monitoring became more practical in the dissertation, we expect more applications in the near future. Parametric monitoring can be used for enforcing security policies, checking library/module API policies, and runtime verification framework. With our scalable parametric monitoring techniques, multiple simultaneous parametric specifications can be monitored efficiently.

As more specifications are described in JavaMOP, more logical formalisms are required. Although JavaMOP supports the flexible architecture that a new logical formalism can be easily implemented as a plugin, some logical formalisms might have challenges in supporting them efficiently, like PTCaRet. Also, structuring specifications is also becoming more important. The dissertation suggests a prototype of specification inheritance, but it needs to be investigated further.

Bibliography

[1] Java Management Extensions. http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html.

[2] *ISO/IEC 14977:1996, Information technology – Syntactic metalanguage – Extended BNF*. ISO, Geneva, Switzerland.

[3] JavaMOP. http://javamop.com.

[4] Mars Climate Orbiter Fact Sheet, NASA. http://mars.jpl.nasa.gov/msp98/orbiter/fact.html.

[5] SPECjvm 2008. http://www.spec.org/jvm2008/.

[6] Hprof: A heap/cpu profiling tool in j2se 5.0. http://java.sun.com/developer/technicalArticles/Programming/HPROF.html.

[7] Tracematches Benchmarks. http://abc.comlab.ox.ac.uk/tmahead.

[8] P. Abercrombie and M. Karaorman. jContractor: Bytecode instrumentation techniques for implementing DBC in Java. In *Runtime Verification*, volume 70.4 of *ENTCS*. Elsevier, 2002.

[9] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie J. Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'05)*, pages 345–364. ACM, 2005.

[10] Rajeev Alur, Kousha Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *LNCS*, pages 467–481. Springer, 2004.

[11] Tomoyuki Aotani and Hidehiko Masuhara. Scope: an aspectj compiler for supporting user-defined analysis-based pointcuts. In *Proceedings of the 6th international conference on Aspect-oriented software development (AOSD '07)*, pages 161–172. ACM, 2007.

[12] Apache Software Foundation. Apache Lucene, . http://lucene.apache.org/.

[13] Apache Software Foundation. Apache MINA, . http://mina.apache.org/.

[14] Apache Software Foundation. Apache Commons Pool, . http://commons.apache.org/pool/.

[15] aspectj. AspectJ. http://eclipse.org/aspectj/.

[16] P. Avgustinov and C. Church. *Trace Monitoring with Free Variables*. PhD thesis, Oxford University, 2009.

[17] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhotak, Ondrej Lhotak, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. ABC: an extensible AspectJ compiler. In *Aspect-Oriented Software Development (AOSD'05)*, pages 87–98. ACM, 2005.

[18] Pavel Avgustinov, Julian Tibble, Eric Bodden, Ondrej Lhotak, Laurie Hendren, Oege de Moor, Neil Ongkingco, and Ganesh Sittampalam. Efficient trace monitoring. Technical Report abc-2006-1, Oxford University, 2006.

[19] Pavel Avgustinov, Julian Tibble, and Oege de Moor. Making trace monitors feasible. In *Object Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*, pages 589–608. ACM, 2007.

[20] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS'04*, volume 3362 of *LNCS*, pages 49–69. Springer, 2004.

[21] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *VMCAI'04*, volume 2937 of *LNCS*, pages 44–57. Springer, 2004.

[22] Howard Barringer, David Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: from EAGLE to RULER. *J. Logic Computation*, November 2008.

[23] Detlef Bartetzko, Clemens Fischer, Michael Moller, and Heike Wehrheim. Jass-Java with Assertions. In *Runtime Verification*, volume 55.2 of *ENTCS*. Elsevier, 2001.

[24] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'06)*, pages 169–190. ACM, 2006.

[25] Eric Bodden. J-LO, a tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen University, 2005.

[26] Eric Bodden, Florian Forster, and Friedrich Steimann. Avoiding infinite recursion with stratified aspects. In *GI-Edition Lecture Notes in Informatics "NODe 2006 GSEM 2006"*, volume P-88, pages 49 – 64. Bonner Köllen Verlag, September 2006.

[27] Eric Bodden, Laurie Hendren, and Ondřej Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *European Conference on Object Oriented Programming (ECOOP'07)*, volume 4609 of *LNCS*, pages 525–549. Springer, 2007.

[28] Eric Bodden, Feng Chen, and Grigore Roşu. Dependent advice: A general approach to optimizing history-based aspects. In *Aspect-Oriented Software Development (AOSD'09)*, pages 3–14. ACM, 2009.

[29] Eric Bodden, Patrick Lam, and Laurie Hendren. Clara: A framework for partially evaluating finite-state runtime monitors ahead of time. In *Runtime Verification (RV'10)*, volume 6418 of *LNCS*, pages 183–197. Springer, 2010.

[30] Feng Chen and Grigore Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *Runtime Verification (RV'03)*, volume 89 of *ENTCS*, pages 108–127. Elsevier, 2003.

[31] Feng Chen and Grigore Roşu. Java-MOP: A monitoring oriented programming environment for Java. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *LNCS*, pages 546–550. Springer, 2005.

[32] Feng Chen and Grigore Roşu. MOP: An efficient and generic runtime verification framework. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*, pages 569–588. ACM, 2007.

[33] Feng Chen and Grigore Roşu. Parametric trace slicing and monitoring. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, volume 5505 of *LNCS*, pages 246–261. Springer, 2009.

[34] Feng Chen, Marcelo D'Amorim, and Grigore Roşu. Checking and correcting behaviors of Java programs at runtime with JavaMOP. In *Runtime Verification (RV'06)*, volume 144 of *ENTCS*, pages 3–20. Elsevier, 2006.

[35] Feng Chen, Dongyun Jin, Patrick Meredith, and Grigore Roşu. Monitoring oriented programming - a project overview. In *Proceedings of the Fourth International Conference on Intelligent Computing and Information Systems (ICICIS'09)*, pages 72–77. ACM, 2009.

[36] Feng Chen, Patrick Meredith, Dongyun Jin, and Grigore Roşu. Efficient formalism-independent monitoring of parametric properties. In *Automated Software Engineering (ASE'09)*, pages 383–394. IEEE, 2009.

[37] Codehaus. Sysunit. `http://docs.codehaus.org/display/SYSUNIT/Home`.

[38] Marcelo d'Amorim and Klaus Havelund. Event-based runtime verification of Java programs. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

[39] Bruno De Fraine, Mario Südholt, and Viviane Jonckers. Strongaspectj: flexible and safe pointcut/advice bindings. In *Proceedings of the 7th international conference on Aspect-Oriented Software Development (AOSD '08)*, pages 60–71. ACM, 2008.

[40] Doron Drusinsky. The Temporal Rover and the ATG Rover. In *Model Checking and Software Verification (SPIN'00)*, volume 1885 of *LNCS*, pages 323–330. Springer, 2000.

[41] Matthew Dwyer, Rahul Purandare, and Suzette Person. Runtime verification in context: Can optimizing error detection improve fault diagnosis. In *Runtime Verification (RV'10)*, volume 6418 of *LNCS*, pages 36–50. Springer, 2010.

[42] The Apache Software Foundation. The Apache Commons Collections. http://commons.apache.org/collections/.

[43] Simon Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'05)*, pages 385–402. ACM, 2005.

[44] Klaus Havelund and Grigore Roşu. Monitoring Java programs with Java PathExplorer. In *Runtime Verification (RV'01)*, volume 55 of *ENTCS*. Elsevier, 2001.

[45] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall Intl., New York, 1985.

[46] Kevin Hoffman and Patrick Eugster. Bridging java and aspectj through explicit join points. In *Proceedings of the 5th international symposium on Principles and practice of programming in Java (PPPJ '07)*, pages 63–72. ACM, 2007.

[47] Vilas Jagannath, Milos Gligoric, Dongyun Jin, Grigore Roşu, and Darko Marinov. IMUnit: Improved multithreaded unit testing. In *Proceedings of the Third International Workshop on Multicore Software Engineering (IWMSE'10)*, IEEE, pages 48–49, 2010.

[48] Vilas Jagannath, Milos Gligoric, Dongyun Jin, Qingzhou Luo, Grigore Roşu, and Darko Marinov. Improved multithreaded unit testing. In *Foundations of Software Engineering (FSE'11)*, pages 223–233. ACM, 2011.

[49] Java Community Process. JSR 166: Concurrency utilities. `http://g.oswego.edu/dl/concurrency-interest/`.

[50] JBoss Community. JBoss Cache. `http://www.jboss.org/jbosscache`.

[51] Dongyun Jin, Patrick O'Neil Meredith, Dennis Griffith, and Grigore Roşu. Garbage collection for monitoring parametric properties. In *Programming Language Design and Implementation (PLDI'11)*, pages 415–424. ACM, 2011.

[52] Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Roşu. Javamop: Efficient parametric runtime monitoring framework. In *Proceeding of the 34th International Conference on Software Engineering (ICSE'12)*. IEEE, 2012. to appear.

[53] Dongyun Jin, Patrick O'Neil Meredith, and Grigore Roşu. Scalable parametric runtime monitoring. Technical Report http://hdl.handle.net/2142/30757, Department of Computer Science, University of Illinois at Urbana-Champaign, April 2012.

[54] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *European Conference on Object Oriented Programming (ECOOP'01)*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.

[55] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a runtime assurance tool for Java. In *Runtime Verification*, 2001.

[56] Moonjoo Kim, Mahesh Viswanathan, Hanêne Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Formally specified monitoring of temporal properties. In *Europoean Conference on Real-Time Systems (ECRTS'99)*, 1999.

[57] MoonZoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-MaC: A run-time assurance approach for Java programs. *J. Formal Methods in System Design*, 24(2):129–155, 2004.

[58] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *Object Oriented Programming, Systems, Languages and Applications (OOPSLA'00)*, pages 105–106. ACM, 2000.

[59] Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, and Grigore Roşu. Towards categorizing and formalizing the JDK API. Technical Report http://hdl.handle.net/2142/30006, Department of Computer Science, University of Illinois at Urbana-Champaign, March 2012.

[60] Nancy G. Leveson. An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26:18–41, 1993.

[61] H. Lu and A. Forin. The design and implementation of P2V, an architecture for zero-overhead online verification of software programs. Technical Report MSR-TR-2007–99, Microsoft Research, 2007.

[62] Michael Martin, V. Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Object Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*, pages 365–383. ACM, 2005.

[63] Patrick Meredith, Dongyun Jin, Feng Chen, and Grigore Roşu. Efficient monitoring of parametric context-free patterns. In *Automated Software Engineering (ASE'08)*, pages 148–157. IEEE, 2008.

[64] Patrick Meredith, Dongyun Jin, Feng Chen, and Grigore Roşu. Efficient monitoring of parametric context-free patterns. *J. Automated Software Engineering*, 17(2):149–180, June 2010.

[65] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. An overview of the MOP runtime verification framework. *International Journal on Software Techniques for Technology Transfer*, 2011.

[66] Rodolfo Pellizzoni, Patrick Meredith, Marco Caccamo, and Grigore Roşu. Hardware runtime monitoring for dependable cots-based real-time embedded systems. In *Real-Time System Symposium (RTSS'08)*, pages 481–491. IEEE, 2008.

[67] William Pugh and Nathaniel Ayewah. MultithreadedTC - A framework for testing concurrent Java applications. `http://code.google.com/p/multithreadedtc/`.

[68] William Pugh and Nathaniel Ayewah. Unit testing concurrent software. In *ASE*, 2007.

[69] Grigore Roşu, Feng Chen, and Thomas Ball. Synthesizing monitors for safety properties – this time with calls and returns –. In *Runtime Verification (RV'08)*, volume 5289 of *LNCS*, pages 51–68. Springer, 2008.

[70] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel java grande benchmark suite. In *Supercomputing (SC'01)*, pages 8–8. ACM, 2001.

[71] Robert E. Strom and Shaula Yemeni. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12:157–171, January 1986.