PREPARATION-FREE AND COMPREHENSIVE RUNTIME VERIFICATION TOOL FOR TESTING JAVA PROGRAMS

BY

CHOONGHWAN LEE

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, 2013

Urbana, Illinois

Doctoral Committee:

Associate Professor Grigore Roşu, Chair and Director of Research
Associate Professor Darko Marinov
Associate Professor Mahesh Viswanathan
Associate Professor Tao Xie
Abstract

Runtime verification is an effective and accurate technique for ensuring that an execution of a program conform to certain specifications at runtime. Although excessive runtime overhead, one of its main drawbacks, has been alleviated by many recent works, its usefulness seems to be limited by rarely available specifications and non-trivial preparation.

This thesis presents research for showing that it is achievable to build a runtime verification system that reveals violations in an execution of a program without requiring any preparation from user’s point of view. This attempt is demonstrated by providing a comprehensive set of specifications for a few commonly used Java class library packages, and devising a system that is capable of instrumenting the program under monitoring at runtime. Additionally, this thesis presents an automated specification mining technique, a few optimization techniques for monitoring, and a new runtime monitoring system, designed with modularity in mind, that separates instrumentation, which can be domain-specific, from monitoring. Using the new system, these specifications have been thoroughly tested and the results show that runtime verification is indeed a convenient and efficient means of ensuring the correctness of a program execution.
To my family
I would like to thank my parents and brother for their support. In particular, I thank my mother, more than words could say, for giving birth to me, raising me up, answering my random questions without complaint—sometimes by sitting with me and reading the encyclopedia together—and always being there.

I would also like to thank my advisor Grigore Roşu for being supportive and patient, even though I have probably disappointed him many times. In particular, I could never forget his advice on writing; I cannot imagine how hard and time-consuming it would be to give me detailed comments for each revision of the draft. Also, I would like to thank the rest of my committee: Darko Marinov, Mahesh Viswanathan, and Tao Xie. Their valuable comments helped me strengthen the research and this thesis.

I also learned a lot from the fellow researchers of the Formal Systems Laboratory (FSL) and would like to thank Michael Adams, Feng Chen, Chucky Ellison, Cansu Erdogan, Dwight Guth, Mark Hills, Jeff Huang, Soha Hussein, Michael Ilseman, Dongyun Jin, David Lazar, Qingzhou Luo, Patrick Meredith, Brandon Moore, Daejun Park, Andrei Popescu, Traian Şerbânuta, Andrei Ştefănescu, and Yi Zhang.

I would like to thank Geneva Belford for her help and advice, especially during my first years; without her help, I would have not been able to start smoothly. Besides research, I was fortunate enough to work as a TA under the supervision of Tom Gambill, Sam Kamin, and Elsa Günter. I would like to thank them for advising me and giving me the opportunity to have teaching experience. Though I cannot enumerate all the names, I would also like to thank many students who were always nice and friendly. Outside of school, I was able to work with several kind colleagues; in particular, I would like to thank Kent Yang and Ruth Aydt for being kind and supportive.

My friends from Seoul National University—Hyung-Chan An, Jun Ki Lee, and Hoeseok Yang—thank you for being almost always online for chatting, and cheering me up with your legendary pieces of gaedlib. Thanks to Wordsobe Mun, my longest friend, for friendship and being unchanged—you always remind me of my precious memory of my childhood, which makes me smile. Thanks to Paul Simonson—my
first, best, and only roommate ever—for affecting me positively and sharing the unit with me, who may not be an ideal person to live with, without complaint.

I believe that nothing would have been possible if I did not grow up in good societies. I would like to thank those who make efforts to make a better society where ordinary people like me can get opportunities to learn almost for free—I feel that I owe more than two decades of my life to my fellow Koreans, and about seven years of it to Americans, especially people in Illinois.

The research has been supported in part by NSF grant CCF-1218605, NSA grant H98230-10-C-0294, DARPA HACMS program as SRI subcontract 19-000222, and the Korea Foundation for Advanced Studies (KFAS).
# Table of Contents

**Chapter 1** Introduction ................................................................. 1  
  1.1 Problem Description ................................................................. 1  
  1.2 Overall Guide ............................................................................. 3  

**Chapter 2** Background ................................................................. 4  
  2.1 Parametric Specification .............................................................. 4  
  2.2 Observing Program Executions ..................................................... 6  
    2.2.1 Observing Program Executions For Monitoring ....................... 7  
    2.2.2 Observing Program Executions For Mining ............................. 8  
    2.2.3 Available Techniques ......................................................... 9  
  2.3 Trace Slicing ............................................................................... 11  
    2.3.1 Trace Slicing For Monitoring .............................................. 13  
    2.3.2 Trace Slicing For Mining .................................................... 14  
  2.4 Monitoring Specifications ......................................................... 14  
  2.5 Mining Specifications .................................................................. 15  

**Chapter 3** Related Work .............................................................. 17  
  3.1 Providing Specifications .............................................................. 17  
    3.1.1 Mining Specifications ............................................................ 17  
    3.1.2 Learning Properties ............................................................. 20  
    3.1.3 Formalizing Desirable Behaviors ......................................... 21  
    3.1.4 Providing Augmented Documentation ................................. 21  
  3.2 Monitoring Specifications ............................................................ 22  

**Chapter 4** Mining Parametric Specifications ............................... 25  
  4.1 Approach Overview ................................................................. 25  
  4.2 Mining Event Specifications ....................................................... 27  
    4.2.1 Learning Related Methods and Parameters ........................... 28  
    4.2.2 Filtering out Generics ......................................................... 30  
    4.2.3 Miscellaneous Filters ........................................................ 31  
  4.3 Slicing Traces ........................................................................... 32  
    4.3.1 Complete and Connected Parameter Bindings ....................... 32  
    4.3.2 Complexity of Trace Slicing ................................................. 34  
    4.3.3 Trace Slicing Algorithm ..................................................... 35  
  4.4 Learning Parametric Specifications .......................................... 39  
    4.4.1 Probabilistic Finite State Automata (PFSA) Learner ............... 39  
    4.4.2 Finite State Automata (FSA) Refiner ................................... 40  

---

vi
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5</td>
<td>4.5 Evaluation of jMiner</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>4.5.1 Performance of SLICE</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>4.5.2 Automated Specification Mining</td>
<td>45</td>
</tr>
<tr>
<td>Chapter 5</td>
<td>Writing Parametric Specifications From Documentation</td>
<td>50</td>
</tr>
<tr>
<td>5.1</td>
<td>5.1 Approach Overview</td>
<td>50</td>
</tr>
<tr>
<td>5.2</td>
<td>5.2 Formalizing the Java API</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>5.2.1 Separating Specification-Implying Text</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>5.2.2 Writing Formal Specifications</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td>5.2.3 Classifying Formal Specifications</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>5.2.4 Examples</td>
<td>56</td>
</tr>
<tr>
<td>5.3</td>
<td>5.3 Evaluation</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>5.3.1 Correctness of Specifications</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>5.3.2 Bug Finding</td>
<td>64</td>
</tr>
<tr>
<td>Chapter 6</td>
<td>Monitoring Parametric Specifications</td>
<td>66</td>
</tr>
<tr>
<td>6.1</td>
<td>6.1 A New Monitoring System</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td>6.1.1 Limitations of Monolithic Design</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td>6.1.2 RV-MONITOR: A Runtime Verification Library Generator</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td>6.1.3 JAVAMOP: An Integrated Runtime Monitoring System</td>
<td>69</td>
</tr>
<tr>
<td>6.2</td>
<td>6.2 Monitoring Multiple Specifications Simultaneously</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>6.2.1 Overhead Analysis</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>6.2.2 Fine-Grained Locks</td>
<td>74</td>
</tr>
<tr>
<td></td>
<td>6.2.3 Optimization for Kleene Star</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td>6.2.4 Weaving for Multiple Specifications</td>
<td>77</td>
</tr>
<tr>
<td></td>
<td>6.2.5 Evaluation</td>
<td>78</td>
</tr>
<tr>
<td>6.3</td>
<td>6.3 Preparation-Free Monitoring</td>
<td>83</td>
</tr>
<tr>
<td></td>
<td>6.3.1 Difficulties in Preparation for Monitoring</td>
<td>83</td>
</tr>
<tr>
<td></td>
<td>6.3.2 Architecture</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>6.3.3 Generating JAVAMOP-AGENT</td>
<td>87</td>
</tr>
<tr>
<td></td>
<td>6.3.4 Runtime Instrumentation</td>
<td>89</td>
</tr>
<tr>
<td></td>
<td>6.3.5 Configuration</td>
<td>91</td>
</tr>
<tr>
<td></td>
<td>6.3.6 Evaluation</td>
<td>91</td>
</tr>
<tr>
<td>Chapter 7</td>
<td>Conclusion</td>
<td>94</td>
</tr>
<tr>
<td>7.1</td>
<td>7.1 Limitations</td>
<td>94</td>
</tr>
<tr>
<td>7.2</td>
<td>7.2 Conclusion</td>
<td>95</td>
</tr>
<tr>
<td>Appendix A</td>
<td>Weaving for Monitoring Multiple Specifications</td>
<td>97</td>
</tr>
<tr>
<td></td>
<td>A.1 Method Call Pointcut</td>
<td>98</td>
</tr>
<tr>
<td></td>
<td>A.2 Field Reference and Field Set Pointcuts</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>A.3 Constructor Call Pointcut</td>
<td>99</td>
</tr>
<tr>
<td>References</td>
<td></td>
<td>102</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Problem Description

Runtime verification systems \cite{23, 64, 17, 55, 55, 18, 31, 52, 33, 8, 10, 54, 29, 2, 9} analyze an execution of software and check if the given specifications are satisfied, increasing the reliability of the analyzed software. Significant runtime overhead was one of main drawbacks, but recently proposed static and dynamic optimization techniques \cite{14, 22, 33, 44} enable those systems to monitor a real world program with reasonable overhead.

Despite the usefulness of runtime monitoring of requirements, it seems there are a few hurdles that make developers or users reluctant to use such systems:

1. since specifications, which runtime monitoring systems check an execution of a program against, are rarely available, it may be doubtful whether these systems can detect any violations;

2. even if one is willing to expend time and effort on writing specifications, it may be doubtful whether runtime overhead is tolerable;

3. it may be also doubtful whether it is convenient to use such systems for real world programs.

Specifications seem rarely available because they are not easy to produce—they often require a deep understanding of the implementation—and it is hard to know what formalism is expressive and readable to describe requirements. Also, previous works on the optimization of runtime monitoring systems have focused on the case of monitoring a single specification at a time; thus, it is still unknown whether monitoring hundreds of specifications simultaneously does not impose excessive overhead or is even possible.

In addition to simultaneous monitoring of multiple specifications, the problem of usability has not been addressed, and one may wonder if it is not prohibitively hard to apply monitoring to his/her project. In particular, instrumentation, a required preparation step for many monitoring systems, can be an obstacle because it is indeed non-trivial for a real world program.
Another concern that developers might have is whether a runtime monitoring system can be extended, when they encounter a case that this system cannot handle. Although the core monitoring functionality can be universal, the means of capturing certain actions in an execution can be domain-specific and, consequently, one may find the off-the-shelf system unsuitable for cases that the system is not designed for.

This thesis presents techniques for providing parametric specifications (Section 2.1), through both an automated system that uses program executions and a manual work that uses documentation. First, this thesis presents a mining system that can be completely automated and infer many meaningful parametric specifications. Then, it also proposes a methodology for writing specifications from documentation, called the API specification (Section 5.1), and shows that parametric specifications are able to find bugs in mature real world programs.

This thesis also presents techniques for monitoring hundreds of specifications, an unprecedented challenge in runtime monitoring systems. A preliminary experiment indeed showed that, when many specifications are simultaneously used, runtime overhead can be high and instrumentation, which is a required step for runtime monitoring, can even fail. A few techniques for improving performance and a technique for avoiding the instrumentation failure are addressed in this thesis.

To improve usability, this thesis presents a system, named JAVA-MOP-IJW (It Just Works), that requires no preparation from the user’s perspective. Taking specifications as input, JAVA-MOP-IJW creates a self-contained JAR file, called a JAVA-MOP-AGENT, that includes the monitoring system, the compiled specifications and all of their dependencies. A JAVA-MOP-AGENT can be enabled by a command line argument when one starts a Java Virtual Machine (JVM), or one can encapsulate it into a one-line-long shell script that can be used as a drop-in replacement for the java executable—this replacement will result in execution of a program and, at the same time, detection of all the violations of the given specifications.

This thesis also discusses a new design of a runtime monitoring system for enabling one to extend the system without the need to understand and modify it, and presents an actual implementation. This new system is implemented in such a way that its core, called RV-MONITOR, can be used as a universal platform for building various runtime monitoring systems. RV-MONITOR, the core module, is designed to implement only indispensable features of a runtime monitoring system—such as listening to events and triggering handlers when a pattern matches—and expose a set of Java methods that can be invoked by the module for firing events and others, which can be domain-specific. This design enables one to build a new monitoring system, which will be still powered by all the optimization techniques from this the-

---

1The system in Jin et al. [44] was also called RV-MONITOR; the name was reused because this system evolved from it. That system is now referred to as JAVA-MOP.
sis and others [20, 22, 53, 44], by simply assembling any means of firing events, such as AspectJ [46], into it.

**Contributions** The contributions of this thesis include:

- **JMiner**, a completely automated system for mining parametric specifications from unit test cases and program execution traces;

- a comprehensive set of formal specifications for four widely used packages (java.io, java.lang, java.net, and java.util) of the Java API, which is ready to be used by an existing runtime monitoring system, JAVAMOP;

- optimization techniques for monitoring multiple specifications simultaneously that result in overhead less than the previous state-of-the-art;

- **RV-Monitor**, an efficient and universal module for the core monitoring functionality, and JAVAMOP 4.0, an integrated runtime monitoring system built on top of RV-Monitor;

- **JAVAMOP-IJW**, a system for generating from a set of specifications JAVAMOP-Agent, a runtime monitoring system that can be used as a drop-in replacement for the java executable;

- a large scale evaluation using 179 parametric specifications simultaneously.

**1.2 Overall Guide**

Chapter 2 provides some background material that needs to be clearly defined to discuss what this thesis presents in the remainder of it, and Chapter 3 explains related work on mining and monitoring specifications. Chapters 4 and 5 respectively present an automated system and a manual work for providing parametric specifications. Chapter 6 then discusses runtime monitoring systems that can utilize these specifications: Section 6.1 first discusses a new design for monitoring systems, and presents JAVAMOP 4.0, a new system based on that design; Section 6.2 presents techniques for monitoring multiple specifications efficiently; and Section 6.3 presents JAVAMOP-IJW, another new monitoring system, which requires no preparation from user’s perspective, such as instrumenting the program to be monitored.
Chapter 2

Background

This chapter provides background on terms and techniques that the research presented in this thesis uses. The presented research can be divided into two parts, mining and monitoring, as respectively introduced in Sections 2.4 and 2.5. Before providing introductions to these parts, this chapter first describes notions and techniques that both parts commonly use. Section 2.1 introduces parametric specifications, which are used as the output of a mining system and the input of a monitoring system. Sections 2.2 and 2.3 describe techniques for observing a program execution and analyzing the resulting execution trace according to parameters, which are necessary to mine or monitor specifications dynamically.

2.1 Parametric Specification

A formal specification defines behaviors that systems or parts of systems must or are recommended to obey. An example of a formal specification is a regular expression: "open write close", where open, write, and close represent creating a FileOutputStream object, calling write(), and calling close(), respectively. This specification states that an opened FileOutputStream object can perform an arbitrary number of write operations and then should be closed. In spite of its simplicity, it is effective in finding a common error: forgetting to invoke close() on a FileOutputStream object of local scope in catch blocks or a finally block.1

Of particular significance is a parametric specification [55], which enables one to define any number of parameters, each of which is bound to a concrete object at runtime. The difference between parametric specifications and non-parametric ones is apparent when one wishes to expresses an interaction that involves multiple objects; e.g., consider the following caveat documented in the API specification:

1It is not generally permissible for one thread to modify a Collection while another thread is iterating over it. In general, the results of the finalize(), invoked by the garbage collector, eventually calls close() to release the resources, but such delayed action can cause file corruption—it occurs because the modification is not visible to other file-handling objects or processes until the buffer is flushed by close() or flush()—and file operation failure—some file systems disallow moving or deleting a file when the file is opened.
One can attempt to describe illegal uses by writing a non-parametric specification: “createIterator useIterator* modifyCollection+ useIterator”, where createIterator, useIterator, and modifyCollection, respectively, represent creating an iterator from a collection (calling `iterator()`), using the iterator (such as calling `hasNext()` or `next()`), and modifying the collection (such as calling `add()`). Although this specification may be useful for toy programs, it can cause false alarms if a program uses multiple collections and iterators. For example, if two distinct iterators appear, one before a modification and the other after the modification, then the above pattern will be matched. The main cause of this false alarm is that there is no distinction between different iterators, and, as a result, the specification is forced to be globally obeyed.

In contrast, a parametric specification permits parameters, which act as the means of making distinction among different objects, and, consequently, interactions from the distinct iterators are not mixed. As a concrete example, Figure 2.1 shows an RV-MONITOR specification. At the beginning (line 1), parameters of this specification are defined: `c` and `i`. These parameters define what types of objects are used to split interactions; in this example, there will be a single interaction for each pair of collection and iterator. A non-parametric specification can be thought of as a specification with no parameters; nothing would split interactions and, as a result, a single interaction would correspond to an entire execution.

The body of a parametric specification typically consists of three parts: event definitions, a property, and a handler. An event definition defines an event and its parameters; e.g., a createIterator event (line 2) carries both `c` and `i`, and a modifyCollection event (line 3) carries only `c`. A property defines a desired/undesired pattern for each interaction; here, it expresses the undesired pattern in an extended regu-
lar expression (ERE) (line 6). A *handler* specifies the behavior when an interaction
matches or fails to match the property; this example simply prints a warning (lines
8–10), but it can contain any code, from logging to recovery.

The value that is associated with a parameter in an event definition can be any-
thing that can be captured when an event occurs. For example, if an event definition
corresponds to a method invocation, any of the target object, the arguments, the re-
turn value, or the calling thread can be the associated value. For this reason, a type-
state \[55\] can be thought of as a special case of a parametric specification. Although
a parameter of an event may be of a primitive type, that of a specification should be
of a reference type. This is because it is hard to conceptualize the notion of life span
and identity for primitive values, which seems essential to define an interaction.

A property does not have to be written in an ERE; one can write it in other
formalisms, such as linear temporal logic (LTL) and context-free grammar (CFG),
or even devise a new one. For the purpose of this thesis, however, the main focus
will be on EREs.

An event definition in an RV-Monitor specification does not specify when the
corresponding event should be fired. Such conditions are assumed to be provided
by another specification. This separation has been made based on the observation
that such conditions vary, depending on the purpose, and there is no silver bullet
language for specifying them in a uniform and elegant way, as further explained in
Sections 2.2.3 and 6.1.1.

### 2.2 Observing Program Executions

Since the presented mining and monitoring techniques are dynamic, they need to
observe program executions. Observing a program execution can be considered
as obtaining a *parametric trace*, which will be defined below, while running a pro-
gram. This section formally defines the notions of *event*, *trace* and *parameter bind-
ing*, which are derived from Chen and Roşu [21], and explains available techniques
for obtaining a parametric trace.

An *event* is a certain action during program execution; this is usually a method
invocation, but can be a field access, object reclamation, static initialization, or pro-
gram termination. In the simplest form, an occurrence of an event can be denoted
by an identifier associated with the event. For example, any invocation of *Iterator.hasNext()* or *Iterator.next()* can be denoted by an identifier *useIterator*. Such an identifier is called a *base event*, and a *non-parametric trace* is formally de-
defined as follows:

\[ \text{Definition 1. (Base events and non-parametric traces)} \]

Let \( E \) be a set of *base events*. 
An E-trace, or non-parametric trace, is a finite sequence of events in E, i.e., an element in $E^*$. We write $e \in w$ when event $e \in E$ appears in trace $w \in E^*$.

While non-parametric traces might be sufficient for some mining approaches, the mining and monitoring approaches in this research will utilize parameter information to split interactions (Section 2.1) and, therefore, use parametric traces, defined as follows when $[A \rightarrow B]$ denotes the sets of partial functions from $A$ to $B$:

**Definition 2. (Parametric events and traces)** Let $X$ be a set of parameters and let $V_X$ be a set of corresponding parameter values. If $E$ is a set of base events (Definition 1), then let $E(X)$ denote the set of corresponding parametric events $e(\theta)$, where $e$ is a base event in $E$ and $\theta$ is a partial function in $[X \rightarrow V_X]$. A parametric trace is a trace with events in $E(X)$, that is, a word in $E(X)^*$. Let $\text{Dom}(\theta)$ be $\{x \in X \mid \theta(x) \text{ defined}\}$ and $\bot \in [X \rightarrow V_X]$ be the map undefined everywhere; i.e., $\text{Dom}(\bot) = \emptyset$. A partial map in $[X \rightarrow V_X]$ is called a parameter binding.

A parametric specification provides both $E$ and $X$. For example, in the specification shown in Figure 2.1, $E$ is \{createIterator, modifyCollection, useIterator\}, and $X$ is (Collection, Iterator). Suppose that there is a createIterator event—it is defined to have two parameters—and this particular event occurrence carries a Collection object $c_1$ and an Iterator object $i_1$. In this parametric event, the base event is createIterator and the parameter binding is (Collection$\mapsto c_1$, Iterator$\mapsto i_1$).

### 2.2.1 Observing Program Executions For Monitoring

To construct a parametric trace, it is necessary to know when a parametric event in $E$ should be fired during execution. For this reason, a monitoring system requires one to provide the condition for firing an event for each event definition. In JAVAMOP [55], for example, such a condition is described as an AspectJ [46] pointcut with JAVAMOP’s extension. As an example, Figure 2.2 shows a parametric specification with AspectJ pointcuts. In this specification, the condition for firing a uselterator event is defined on lines 18–21, which can be interpreted as “when Iterator.hasNext() or Iterator.next(), regardless of its parameter types or return type, is about to be invoked, a parametric event, useltator with the parameter binding $\langle$Iterator$\mapsto$ the target object $\rangle$, is fired.”

In this thesis, a parametric specification that specifies conditions for firing events will be referred to as a JAVAMOP specification, in order to avoid confusion with an RV-MONITOR specification.

A parametric event excludes all irrelevant parameters, according to the conditions for firing events. The above condition, for example, describes that the return values of both methods are not of interest. As a result, while observing a program execution, a monitoring system ignores them. Although it might be obvious, it should
be also noted that the above condition filters out all the irrelevant events; e.g., invoking `Collection.size()` does not fire any event. That is, a parametric trace is focused regarding the given parametric specification, which provides $E$ and $X$.

### 2.2.2 Observing Program Executions For Mining

Unlike a monitoring system, which requires a parametric specification and therefore can assume parametric traces to be focused, a mining system cannot assume such traces. Moreover, a fully automated mining system cannot assume that even the set of events, denoted by $E$, and the set of parameters, denoted by $X$, are known.

A reasonable way to deal with such lack of information would be to generate a comprehensive trace from an execution, in such a way that a parametric trace

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

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

    ere : createIterator useIterator* modifyCollection+ useIterator

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

Figure 2.2: A JAVAMOP specification `Collection_UnsafeIterator`. 
according to any $E$ and $X$ can be derived from it. This way, a mining system can attempt to infer a specification with different $E$ and $X$, without rerunning the program, which may not only consume much time but also yield different trace. For example, the mining approach presented in this thesis generates a trace that considers any method invocation as an event; i.e., $E$ is the set of methods that are ever called during execution, and $X$ is the set of all reference types that appear in $E$.

2.2.3 Available Techniques

This section describes a few commonly used techniques for observing program executions. All of these approaches have been implemented and used in the mining system or monitoring system in this research.

Instrumentation

The most widely used technique is to instrument the program under monitoring or mining. This technique injects a routine for generating a parametric or non-parametric event into any code point that matches the condition for firing that event. For example, for each call site of `Iterator.hasNext()` or `Iterator.next()`, one can insert code for capturing the target object and generating a `useIterator` parametric event with this object.

The only notable part of this technique is how to pick up places that match with the provided conditions for firing events. For the purpose of picking up such places, one can utilize AspectJ [46], an Aspect-Oriented Programming (AOP) tool. As one can see from the example in Section 2.2.1, an AspectJ pointcut enables one to specify method invocation using patterns with some wild cards, and capture related objects, such as the target object and arguments. In addition to method invocation, AspectJ has pointcuts for field access, static initialization, and so forth, which makes AspectJ almost sufficient in specifying conditions for firing parametric events. For this reason, AspectJ has been adopted by some monitoring systems, such as JAVAMOP [55] and tracematches [2]. For example, JAVAMOP generates an AspectJ `aspect` from the user-specified parametric specifications. Then, an AspectJ compiler can be employed to instrument a program to be monitored, which is referred to as `weaving`; by reading the program, it picks out places, called `join points`, where an event should be fired, and inserts into each `join point` the corresponding `advice`, part of the generated aspect for firing a parametric event and handling it.

Being pattern-based, AspectJ may not be ideal if events need to be fired in arbitrary places. Also, inserting a piece of advice into matched join points increases the size of a method. In particular, if a method contains many matched join points and many pieces of advice are inserted into such join points, the increment can be
excessive, causing the size to exceed 64KB, Java's limit \[57, \S 4.9.1\]. In such cases, one can instrument at a lower level using a general instrumentation tool, such as Javassist \[12\]. Such tools typically enable one to visit each statement or expression, and insert user-specified statements and expressions before or after it.

Although most monitoring systems assume that instrumentation is performed at compile time, it is possible to instrument at runtime; more specifically, at the time each class is loaded onto the JVM, using a Java agent \[39\]. A Java agent is a user-defined JAR-packaged component, enabled by a command line option; e.g., to enable javamop_agent.jar while running a program named Foo:

```bash
$ java -javaagent:javamop_agent.jar Foo
```

Then, the JVM invokes the enabled Java agents whenever it is about to load a class, which gives the agents an opportunity to modify the class in such a way that events can be fired during execution. AspectJ similarly supports such runtime instrumentation, which is called load-time weaving (LTW), by providing a Java agent that implements the weaving functionality.

### A Java Virtual Machine Tool Interface (JVMTI) Agent

As explained in Section 2.2.2, a mining system needs to generate a comprehensive trace, such as recording every method invocation, due to the lack of specifications. For the purpose of generating such trace, writing a JVMTI \[41\] agent for listening to JVM's events can be more convenient and thorough than instrumentation. Designed for writing profilers and debuggers \[41\], JVMTI provides a way to listen to various events that occur at runtime, such as entering a method, returning from a method, and accessing a field. For example, one can let the JVM call a callback function defined in his/her own JVMTI agent whenever a thread enters a method. The callback function then can inspect the target object and arguments, and fire a parametric event. Since the proposed mining approach focuses on only method invocations, the agent is configured to record only two types of events: entering a method (JVMTI_EVENT_METHOD_ENTRY), and returning from a method with or without an exception (JVMTI_EVENT_METHOD_EXIT).

A JVMTI agent is typically written in C/C++ and packed into a dynamically linked library (a .dll file) under Microsoft Windows or a shared object (an .so file) under UNIX systems. It can be enabled by a command line option, similarly to a Java agent; e.g., to enable jminer_agent.dll while running a program named Foo:

```bash
$ java -agentpath:jminer_agent.dll Foo
```

Being managed and invoked by the system (i.e., JVM), a JVMTI agent is notified of all events that originate from not only user-defined classes but also classes in

---

10
the runtime library, which is typically packed in rt.jar; as a result, an agent can generate a thorough execution trace. In contrast, it is not trivial to listen to events that originate from the runtime classes using instrumentation, where firing events is performed solely by the user code. Since most JVMs allow the user to specify an alternative runtime library, it might be possible to instrument it first and let the JVM use the instrumented one. However, this is not only inconvenient but also unsafe; the runtime library is sensitive and, consequently, such modification may cause the JVM to crash during initialization.

Apart from the convenience, the strength of such a JVMTI agent lies in its ability to maintain a unique identifier for each object. One may be tempted to think that System.identityHashCode() returns a unique identifier but there is a chance of hash collision. Maintaining unique identifiers without any chance of collision is crucial because they are the key to separate interactions; assigning the same identifier to two distinct objects would result in merging two different interactions, which would result in false positives/negatives under monitoring or inaccurate specifications under mining. JVMTI allows an agent to associate a tag with an object (SetTag() and GetTag()), and embed a unique identifier in the tag.

2.3 Trace Slicing

Once a parametric trace is obtained, trace slicing is then performed in order to extract interactions. This section explains a few non-trivial issues that arise when the trace slicer identifies interactions, and formally defines trace slicing.

Within a parametric trace, multiple interactions coexist and they may overlap. For example, consider the following simplified fragment of a parametric trace:

1. createIterator(Collection->c0, Iterator->i1)
2. useIterator(Iterator->i1)
3. modifyCollection(Collection->c0)
4. createIterator(Collection->c0, Iterator->i2)
5. useIterator(Iterator->i2)

Here, one Iterator object i1 is used before the underlying Collection object c0 is modified. After the modification, another Iterator object i2 is then created for the same Collection object.

As explained in Section 2.1, distinct interactions are identified based on parameters that are bound to parameter values. For example, when the set of parameters X is (Collection, Iterator), this parametric trace is considered to have two interactions: one for (Collection->c0, Iterator->i1) and the other for (Collection->c0, Iterator->i2).
Although it is obvious that events 1 and 4, respectively, should belong to the two interactions, it might be questionable whether the other three events, which do not carry all the parameters, should belong to any interaction. To consider such cases, the following relation is first defined [21]:

**Definition 3.** \( \theta' \) **is less informative** than \( \theta \), written \( \theta' \subseteq \theta \), if \( \theta'(x) \) is defined then \( \theta(x) \) is also defined and \( \theta'(x) = \theta(x) \), for any \( x \in X \).

For example, both \( \langle \text{Iterator} \mapsto i_1 \rangle \) from event 2 and \( \langle \text{Collection} \mapsto c_0 \rangle \) from event 3 are less informative than \( \langle \text{Collection} \mapsto c_0, \text{Iterator} \mapsto i_1 \rangle \).

Under many monitoring systems, those three incompletely bound events are considered to be part of interactions for more informative parameter bindings; e.g., both events 2 and 3 are considered to belong to the interaction for \( \langle \text{Collection} \mapsto c_0, \text{Iterator} \mapsto i_1 \rangle \). It seems that this decision is made because those events are also important steps to reach legal or illegal states. In the above example, modifying a collection (event 3) is indeed significant because the modification invalidates all the previously created iterators for that collection and, as a result, those iterators should not be used anymore.

A natural consequence from this decision is that an event can belong to multiple interactions; e.g., in the above trace, \( \langle \text{Collection} \mapsto c_0 \rangle \) from event 3 is less informative than not only \( \langle \text{Collection} \mapsto c_0, \text{Iterator} \mapsto i_1 \rangle \) but also \( \langle \text{Collection} \mapsto c_0, \text{Iterator} \mapsto i_2 \rangle \); therefore, this event is part of both interactions.

Trace slicing is formally defined, based on the above decision, as follows:

**Definition 4.** **(Trace slicing)** Given a parametric trace \( \tau \in E(X)^* \) and a partial function \( \theta \) in \( [X \rightarrow V_X] \), let the \( \theta \)-**trace slice** \( \tau \rhd \theta \) of \( \tau \) be the non-parametric trace in \( E^* \) defined as:

- \( \epsilon \rhd \theta = \epsilon \), where \( \epsilon \) is the empty trace/word, and
- \( (\tau_\theta e(\theta')) \rhd \theta = \begin{cases} (\tau \rhd \theta)e & \text{when } \theta' \subseteq \theta \\ \tau \rhd \theta & \text{otherwise} \end{cases} \)

A **trace slice**, the output of trace slicing, represents an interaction because, by definition, it filters out all the events that are irrelevant to the given parameter binding \( \theta \). Also, a trace slice drops parameters because it is already specific to the given parameter binding and they are no longer needed. For example, the trace slice for \( \langle \text{Collection} \mapsto c_0, \text{Iterator} \mapsto i_1 \rangle \) is \( \text{[createIterator, useIterator, modifyCollection]} \) from events 1, 2 and 3.

Although trace slicing is defined with respect to one particular parameter binding, a typical trace slicing algorithm (or **trace slicer**) detects every parameter binding that appears in the given parametric trace on-the-fly. That is, a trace slicer takes a
parametric trace $\tau$ and a set of parameters $X$ as input, and outputs multiple trace slices, one for each parameter binding observed in the parametric trace. For example, when the above parametric trace and $X = \langle \text{Collection, Iterator} \rangle$ are provided as input, a trace slicer yields two trace slices: one for $\langle \text{Collection} \mapsto c_0, \text{Iterator} \mapsto i_1 \rangle$, and the other for $\langle \text{Collection} \mapsto c_0, \text{Iterator} \mapsto i_2 \rangle$.

Monitoring systems and mining systems need different types of trace slicers, although both have the same goal: identifying interactions. Below the difference between their assumptions and approaches are explained.

### 2.3.1 Trace Slicing For Monitoring

For the purpose of monitoring, a trace slicer is required to run along with the program under monitoring because a violation needs to be reported immediately. Such online trace slicers were introduced in Chen and Roșu [21] and Chen et al. [22]. For efficiency concerns, they do not actually keep trace slices, which can be arbitrarily long and consume much memory. Instead, for each parameter binding, they maintain a monitor instance (Section 2.4), which keeps only the minimal information. When an event occurs, these trace slicers dispatch the event to all the corresponding monitors, so that each monitor transitions accordingly, and then forget about it. It is acceptable for these trace slicers to forget events because a typical monitoring system does not include the event history in a violation report.

Since an event is not kept, it is impossible to look up past events and, therefore, these trace slicers may be required to create monitor instances even for parameter bindings that have not been observed but may occur in the future. As an example, consider the following parametric trace:

```plaintext
1 createIterator(\langle \text{Collection} \mapsto c_3, \text{Iterator} \mapsto i_3 \rangle)
2 useIterator(\langle \text{Iterator} \mapsto i_3 \rangle)
3 modifyCollection(\langle \text{Collection} \mapsto c_4 \rangle)
```

The first two events are handled without a surprise: when the first event occurs, a monitor instance for $\langle \text{Collection} \mapsto c_3, \text{Iterator} \mapsto i_3 \rangle$ is created; and the second event is dispatched to this monitor, because the parameter binding of the monitor instance is more informative than that of the second event. When the third event occurs, however, a monitor instance for $\langle \text{Collection} \mapsto c_4, \text{Iterator} \mapsto i_3 \rangle$ can be surprisingly created, although this parameter binding has never been observed. This proactive monitor creation is performed because, in the future, an event may bring that parameter binding—without this preparation, this monitor instance would not be able to make necessary transitions according to events that occur between the third event and that future event.
Based on the semantics of Collection and Iterator, \((\text{Collection} \mapsto c_4, \text{Iterator} \mapsto i_3)\) in this example, is spurious because an iterator is specific to a collection and, therefore, \(i_3\) will never interact with \(c_4\). One can prevent a trace slicer from creating a monitor instance for such a spurious parameter binding, by not using the creation keyword at the event definition; e.g., in Figure 2.1, only createIterator can create a monitor instance because it is marked with this keyword on line 2. In contrast, modifyCollection on line 3 is not marked and, consequently, the third event in the above trace does not cause the trace slicer to create a monitor instance.

2.3.2 Trace Slicing For Mining

Unlike a monitoring system, a mining system, especially a property learner (Sections 2.5 and 4.4) in it, needs non-sequential access to events in trace slices. Therefore, it is unavoidable to produce physical trace slices and, consequently, it is necessary to remember events during trace slicing. This may impose significant memory overhead during trace slicing.

Also, user-defined information for suppressing spurious parameter bindings, such as \((\text{Collection} \mapsto c_4, \text{Iterator} \mapsto i_3)\) in the parametric trace in Section 2.3.1, is unavailable to a trace slicer for mining. Instead, a spurious parameter binding can be detected by checking whether it is ever observed as it is or as a combination of multiple existing parameter bindings that are “connected” by common parameter values, which will be discussed further in Section 4.3.1. For example, a trace slicer for mining can find \((\text{Collection} \mapsto c_4, \text{Iterator} \mapsto i_3)\) spurious by considering that it is never observed as it is, and no combination of parameter bindings yields it.

In Section 4.3, this thesis presents a trace slicing algorithm that tackles these two challenges: overhead and spurious trace slices.

2.4 Monitoring Specifications

Since each interaction is identified by a trace slicer, as a form of a trace slice, the remaining part of a monitoring system can separately check whether each trace slice matches or fails to match the property in the given parametric specification. In JAVA-MOP, such check is performed in a monitor instance, an instance of a monitor template. A monitor instance can be thought of as a finite state machine (FSM), where the input alphabet being the event definitions in the specification. The transition table and other information for running such an FSM is derived from the property, at the time the specification is compiled, and stored in the monitor template. When an event occurs, it is dispatched to all the corresponding monitor instances by the

---

2 Although some simple property learners require only sequential access to trace slices, a general mining system, discussed in this thesis, does not assume that only such learners are chosen.
trace slicer, and it triggers a transition in each monitor instance. As one may expect, certain states in a monitor instance indicate a match (or a failure) of the given property. If such states are reached, the monitoring system invokes the corresponding handler defined in the specification.

Common challenges among monitoring systems include expressiveness and efficiency. In particular, runtime monitoring can significantly degrade performance and even result in a crash, due to memory exhaustion, because it is not uncommon that millions of events and parameter bindings appear during an execution of a real world program.

### 2.5 Mining Specifications

If trace slices are obtained from executions of mature programs, one can assume that they are likely to represent correct behavioral patterns. Based on this assumption, an easy approach would be to write a property that accepts only any of these trace slices. This naive method, however, is undesirable for at least two reasons. First, the inferred property is likely to be so picky that any trace slice that is slightly different but still legal would be considered a violation. For example, suppose that we are attempting to infer a resource usage pattern on the Reader class, like the one shown in Figure 4.16. This naive method would remember the number of occurrences of read and, as a result, it would yield a property that falsely warns any other program, unless that program happens to invoke read() the same number of times. Second, the inferred property is likely to be complicated to read if diverse trace slices were obtained. A complicated property hinders users from reviewing it, which is an important step for miners that do not guarantee the correctness. It may also result in worse performance when one uses it for monitoring a program afterwards.

To avoid these problems, a mining system that is capable of inferring an arbitrary property generalizes the observed trace slices. There have been many algorithms that achieve generalization in the context of machine learning [11, 53, 68, 13, 58, 4], and some of them [11, 53] have been adopted to mining systems. In this thesis, such algorithms will be referred to as property learners.

Although generalization is necessary, overly generalized property would be a problem as well, because that property would miss violations. A natural question arises from this: “how general is general enough?”

In addition to the level of generalization, it is also vague to define what specifications are useful. For example, the Iterator\_HasNext specification, which states that Iterator\_next() can be called only if Iterator\_hasNext() is called and it returns true, is deemed useful, given that it is mentioned in both TRACEMATCHES [17] and

---

3The specification in it is slightly different: it ignores the return value of Iterator\_hasNext().
However, one might rightfully argue that the specification is not useful because, in most cases, programmers use `for-each` loops and, consequently, there is no need to check the pattern explicitly. One can also argue that the specification is incomplete because it is legal to call `Iterator.next()` consecutively if the number of elements is known.
Chapter 3

Related Work

There have been many approaches to both runtime monitoring and specification mining. This chapter presents a brief history of the topics that the thesis discusses, and an overview of approaches that are closely related to these topics.

3.1 Providing Specifications

There have been numerous approaches to mining specifications, and they are surprisingly very different from each other. The reason for such diversity might be that this topic is hard and none of existing approaches are mature enough to suggest a general solution. This section explains several approaches to specification mining and some other related techniques. Section 3.1.1 first summarizes mining approaches that can be considered complete systems; i.e., these systems can infer a specification from ordinary source code or execution traces. Then, Section 3.1.2 discusses property learners (Section 2.5). Here property learners, which can be considered part of complete mining systems, are further explained in a dedicated section because they can be potentially adopted by jMiner. In Sections 3.1.3 and 3.1.4, a few manual approaches to providing specifications or more informative documentation are discussed.

3.1.1 Mining Specifications

Ammons et al. [3] propose a technique for mining specifications from execution traces and user-provided input: functions of interest, attributes for those functions, and a scenario seed. It extracts a set of API usage scenarios from execution traces and then passes it to a probabilistic finite state automata (PFSA) learner. Providing attributes requires in-depth knowledge, such as side-effect of each function; one should imagine a hypothetical object corresponding to a scenario, and should mark a parameter as define or, respectively, as use if the parameter changes or depends upon the state of the object. Scenarios are identified by starting from the seed event, searching the execution trace along define-use chain. Having explicit seed events and using the chain reduce the search space, but it may result in failing to recognize
complete interactions. For example, consider the following trace:

1. `ArrayList.add(Collection↦𝑐₁)`
2. `AbstractList.iterator(Collection↦𝑐₁, Iterator↦𝑖₂)`
3. `AbstractList.Itr.hasNext(Iterator↦𝑖₂)`
4. `AbstractList.iterator(Collection↦𝑐₁, Iterator↦𝑖₃)`
5. `AbstractList.Itr.hasNext(Iterator↦𝑖₃)`
6. `ArrayList.add(Collection↦𝑐₁)`

Also, suppose that the seed event is `iterator()`—it is the only event that connects `Collection` and `Iterator`—and that `add()` *defines* `Collection`, `iterator()` *defines* `Iterator` and `uses` `Collection`, and `hasNext()` *uses* `Iterator`. For example, event 2 depends on event 1 because event 1 *defines* `⟨Collection↦𝑐₁⟩` and event 2 *uses* `⟨Collection↦𝑐₁⟩`. From these inputs, two scenarios can be extracted: `[add, iterator, hasNext]` from events 1, 2 and 3; and `[add, iterator, hasNext]` from events 1, 4 and 5. However, none of them are complete with regards to the interaction between `Collection` and `Iterator`: none of them include event 6 because `add()` does not *use* `Collection` and, consequently, event 6 cannot be reached along *define-use* chain in either of the scenarios. As a result, the inferred finite state automata (FSA) will lack the transition from `hasNext()` to `add()`, which wrongly prevents any update after using an iterator. Marking `add()` as both *define* and *use* is not a proper solution either, because this will wrongly add both events 4 and 6 to the scenario of events 1, 2 and 3. No matter how attributes are adjusted, this approach cannot infer the comprehensive FSA shown in Figure 4.13 in situations where jMiner can.

Pradel and Gross [61] propose a dynamic mining technique that collects from execution traces a list of related receiver-method pairs up to a user-specified level of nested method calls, and then infers an FSM. Unlike jMiner, their technique does not consider individual interactions separately. Therefore, it may merge individual interactions and thus infer inaccurate specifications. For example, if the execution trace in Figure 4.13 is observed within a method, their technique will not consider the two interactions separately and, consequently, infer a faulty specification that allows consecutive calls to `next()`. Moreover, it cannot infer a specification that spans over multiple threads, since it creates a separate trace for each thread; e.g., the specification on a `ServerSocket` object and an accepted `Socket` object, which are typically used in different threads, cannot be mined. Furthermore, it may fail to mine specifications from distantly related events if the value of the level of nested method calls is too small. If, on the other hand, the value is too large, it may produce specifications that include too many methods and would likely be application-specific.

Yang et al. [71] propose a technique to find all pairs of methods that satisfy the predefined particular pattern \((ab)^*\) from execution traces. Although their chaining
heuristic composes somewhat more complex patterns, such as \((abc)^*\) (by connecting related specifications into a chain), it cannot infer complicated patterns like in Figure 4.13. Gabel and Su \([32]\) extend Yang et al. \([71]\); their work considers an additional predefined pattern \((ab^i c)^*\), called the resource usage pattern. It then combines instances of these basic patterns, generating complex patterns. Unlike JMINER, it neglects parameters; thus, it may infer meaningless specifications from sequences of irrelevant events that happen to match the predefined patterns.

Dallmeier et al. \([23]\) also present a technique for mining FSAs from execution traces. A state in the FSA inferred by their work represents the results of inspector methods, which observe the internal state of an object, such as \(\text{isEmpty}()\) and \(\text{hasNext}()\), whereas a state in JMINER is abstract, such as “before using an iterator”. Associating each state with inspectors can help users to easily understand the specification, but it is incapable of capturing implicit states such as “an iterator for a collection is being used” because no methods in Collection can observe it. In JMINER, the sequence of method calls can capture those states. Moreover, their work considers only one object and is essentially non-parametric.

Henkel et al. \([36]\) present a dynamic mining technique which is specific to container classes. Their technique actively constructs various operations (by invoking methods of a container), observes the state of the container, and then infers relations among distinct operations, such as state equivalence. In their technique, parameters are predefined and interactions on them are not considered; parameters are inserted into a container and solely used to determine the state of the enclosing container. In contrast, JMINER passively observes interactions on parameters occurring in existing programs and then infers the FSA by generalizing all the observed interactions.

Acharya et al. \([1]\) propose a static technique that generates a set of traces along possible execution paths directly from the source code, and then produces an API usage pattern from it. Since it mines partial orders, the resulting specifications cannot describe loops; thus, it cannot mine many specifications that JMINER can. Zhong et al. \([72]\) also present a static mining technique for sequential patterns from open source repositories. Unlike JMINER, their tool does not consider individual interactions separately. For example, if there are multiple distinct interactions on Collection in a method, their tool can extract a faulty method call sequence. Since their tool inlines multiple methods, the probability that a method call sequence consists of multiple interactions on Collection is high, which makes this approach improper to mine specifications of frequently used classes. Static approaches usually infer from the source code, but some infer from comments using natural language processing (NLP); e.g., Zhong et al. \([73]\) propose an automated technique to infer the resource usage specifications from documentation.
Dallmeier et al. [24] present a dynamic typestate mining approach that incorporates a test case generator for experiencing more behaviors. Their technique first mines the initial typestate model solely from observed behaviors, and then enriches the model by generating mutated test cases and observing whether or not the generated test cases raise runtime exceptions. Their experiment shows that the enriched models are better to find errors and have fewer false positives than the initial models.

3.1.2 Learning Properties

To infer a property, a mining system uses a property learner. A property learner typically takes a set of strings as input, and yields an FSA. There are algorithms for learning other formalism, but this thesis focuses on only FSA-generating ones.

Positive samples, i.e., legal behaviors, are relatively easy to observe—they can be obtained by running mature software—but negative samples are rarely available. Also, it is even hard to make assumption on how comprehensive the observed positive samples are. As a result, many dynamic specification mining approaches employ algorithms that infer solely from an arbitrary number of positive samples. One widely used algorithm is the \( \text{K-TAIL} \) algorithm [11]. This algorithm first constructs an FSA that precisely accepts the input set of strings, and then generalizes the FSA by merging states that are \( k \)-equivalent: two states \( q, q' \) are \( k \)-equivalent if they are not distinguishable by any string \( x \) such that \( |x| \leq k \); i.e., they have the same \( k \)-tail [57]. There is a trade-off between small \( k \) and large \( k \): small \( k \) yields a small FSA but may cause over-generalization; and large \( k \) may prevent generalization, yielding an FSA that accepts only observed strings and rejects unobserved “similar” strings.

The \text{SK-STRINGS} algorithm, which generates a PFSA, is a variant of the \text{K-TAIL} algorithm [53]. This algorithm merges two states if frequently generated strings from each of the two states are matched. Another variant, the \text{GK-TAIL} algorithm, generates an FSA annotated with conditions on data values for each edge, called an extended finite state machine (EFSM) [51].

If negative samples are available as well, the regular positive and negative inference (RPNI) algorithm can be used to obtain an FSA [58]. This algorithm first constructs an FSA that accepts all the positive samples, and then merges states in such a way that all the positive samples are accepted and all the negative ones are rejected. As a result, over-generalization can be avoided as long as negative samples are sufficiently provided. Due to the lack of negative samples and the difficulty of generating them, however, this algorithm and similar ones have not been widely used for specification mining.

Rather than passively receiving all the available samples, some algorithms, such as the \( L^* \) algorithm [4], learn by asking a teacher two types of questions, assuming
that the teacher is capable of answering them correctly. The first type of questions is a *membership query* for checking if a string, generated by the $L^*$ algorithm, belongs to the target language. The other type is a *conjecture* for checking if the learner's current automaton is correct. If the current automaton is incorrect, the teacher is to provide a counterexample string that exclusively belongs to either the learner's automaton or the target language, so that the learner can adjust the automaton. In the context of specification mining, having a teacher that can answer conjectures is impossible. Although Angluin [4] also shows that a random sampling oracle may be substituted for the necessity of answering conjectures, applying this algorithm to mining is still hard because it is non-trivial to answer a membership query for an arbitrary string.

### 3.1.3 Formalizing Desirable Behaviors

Runtime verification tools have defined several formal specifications to gain confidence in their correctness and measure their performance. For example, both JAVAMOP [55] and TRACEMATCHES [3] define several specifications mostly from java.io and java.util. These specifications are subsumed by the work presented in this thesis.

Another formalization approach is presented by Java Modeling Language (JML), which enables one to add contracts and invariants for each method and class [60]. Although behaviors are described differently, many of them can be formalized in both JAVAMOP and JML. For example, consider the following paragraph:

> Once the stream has been closed, further `read()`, `available()`, `reset()` or `skip()` invocations will throw an `IOException`. Closing a closed stream has no effect.

In a JAVAMOP specification, one can specify the undesirable behavior using an ERE: `close+ (read | available | reset | skip)+`. When any of four manipulation methods is invoked after close, this pattern is matched, and JAVAMOP detects a violation. In JML, one can specify the behavior by defining a *model field* of type `boolean` that is set when a stream is created and unset when it is closed, and specifying a precondition on each of the four manipulation methods to ensure that the defined model field is true. Although one can easily understand this simple example, it can be difficult to understand a chain of pre- and post-conditions for an arbitrary sequence of operations [26].

### 3.1.4 Providing Augmented Documentation

There are also a few techniques for providing more informative documentation, rather than providing formal specifications. Although verification is infeasible for
these techniques, such documentation can lead programmers to write safe and reliable code, because it draws their attention and shows caveats.

eMoose [27], an eclipse [30] plugin, highlights directives, keywords that would likely imply desirable patterns, in the documentation when it appears in the editor. In addition to highlighting, this tool also identifies, in the source code editor, all method calls whose targets have directives.

JML [60], explained in Section 3.1.3, enables one to add method contracts and invariants to documentation, and generate from such annotated documentation an API specification augmented with them. Unlike PropDoc’s results (Section 5.1), where specification-implying text is highlighted and formal specifications are linked to that text, this tool simply places the formalized behavioral interface below the existing method or class definition.

3.2 Monitoring Specifications

There are a number of runtime monitoring systems, such as Hawk/Eagle [23], J-Lo [64, 13, 16], JaMaCe [47], JavaMOP [55], JPax [35], Pall [18], POET [31], PQL [52], PTQL [33], QVM [8], RuleR [11], SpoX [34], TemporalRover [29], and Tracematches [2, 9]. Among them, this section focuses on JavaMOP because it is efficient and expressive, in the sense that it supports various formalisms, unlike most other systems. More information about other systems and JavaMOP can be found in Chen [19], Meredith [54] and Jin [43].

One challenge in a monitoring system is efficiency because it is not uncommon that a system needs to maintain millions of monitor instances while monitoring a real world program. To efficiently iterate over all the monitor instances affected by an event, which carries a parameter binding, JavaMOP uses a special data structure, called an indexing tree. An indexing tree is a multi-level map that, at each level, indexes each parameter value of the parameter binding. As an example, Figure 3.1 shows all the indexing trees for the Collection_UnsafeIterator specification, shown in Figure 2.1. When a parametric event createliterator(\text{Collection} \rightarrow {c_0}, \text{Iterator} \rightarrow {i_2}) occurs, for instance, one can efficiently retrieve the affected monitor instance by searching for \(c_0\) and \(i_2\) at each level in the 2-level map, shown in the left of Figure 3.1. However, when only \text{Iterator} is bound, such as uselIterator(\text{Iterator} \rightarrow {i_2}), it would be inefficient to retrieve all the affected instances using the 2-level map. To handle such case efficiently, another map, shown in the right of Figure 3.1, is constructed as well. Unlike the 2-level map, where a leaf holds at most one monitor instance, this 1-level map permits a set of instances at each leaf, because there can be multiple parameter bindings that bind \text{Iterator} to the same
If an indexing tree holds a strong reference (i.e., an ordinary Java reference) to a parameter value, this object becomes ineligible for garbage collection, which leads to a memory leak. To avoid this, indexing trees store weak references, which enables the garbage collector to reclaim the referents. Since a weak reference gives indication that the referent has been reclaimed by returning `null`, a monitoring system can detect broken mappings in the indexing trees and clean them up.

Most existing monitoring systems are not capable of handling multiple specifications simultaneously, or have rudimentary support: considering each specification separately. Since each specification is individually handled, the runtime overhead for running them simultaneously is likely to be at least the summation of the overheads of running each in isolation.

Purandare et al. [62] present a study of overhead arising during the simultaneous monitoring of multiple specifications. Their approach reduces runtime overhead by merging monitors from the same and/or different specifications. Since their implementation was unavailable, it was impossible to investigate their work, but it is deemed orthogonal to the work presented in this thesis. Their work might be also complementary to the optimizations presented in this thesis and Jin [43], but a major optimization introduced in JAVA.MOP 2.3 [44], which keeps track of timestamp for each monitor instance, may make it hard to integrate their work into JAVA.MOP.

Jin [43] presents another work, which is independent of Purandare et al. [62]. At the heart of this technique is sharing resources between specifications, based on the observation that many specifications are concerned with common parameters.

---

1 In this particular specification, even a leaf in the right map will hold at most one monitor instance due to the semantics of `Collection` and `Iterator`—`iterator()` always returns a fresh `Iterator` object. However, JAVA.MOP does not exclude the possibility of multiple monitor instances at the leaf and, therefore, permits a set of them.
and events—an exploratory study shows that only 42 specifications are independent from all others among the 137 specifications from Lee et al. [50]. This work also presents a technique that combines multiple indexing trees when they share the same prefix, in order to reduce memory overhead. For example, for the Collection_UnsafeIterator specification, this technique yields two indexing trees, as shown in Figure 3.2, whereas there are three indexing trees, as shown in Figure 3.1, without this technique. Although this work suggests a reasonable direction, its implementation, JAVAMOP 3.0, turned out to be incorrect due to a wrong assumption it made. JAVAMOP 4.0, presented in this thesis, will incorporate some optimizations from this work, after fixing the fault in the implementation.
Chapter 4

Mining Parametric Specifications

This chapter describes jMiner, a parametric specification mining system that is fully automatic and capable of inferring arbitrarily complex FSMs as properties. Three main components will be explained: a component for inferring the set of events and the set of parameters, a trace slicer, and a property learner. Much of the work in this chapter is from Lee et al. [49].

4.1 Approach Overview

The proposed mining approach consists of two stages, as depicted in Figure 4.1: event specification mining (Section 4.2) and parametric specification mining (Sections 4.3 and 4.4). The former yields a set of event specifications (defined below), and the latter mines a parametric specification for each event specification.

Definition 5. (Event specification) We write a method as \( m(T_t, T_r, T_{p1}, ..., T_{pn}) \), where \( m \) is the method name, \( T_t \) is the target type, \( T_r \) is the return type, and \( T_{p1}, ..., T_{pn} \) are the types of its parameters; for uniformity, we call each of \( T_t, T_r, T_{p1}, ..., T_{pn} \) a method parameter. If \( M \) is a set of methods, let \( X_M \) be all the method parameters of reference type for all methods in \( M \). An event specification is a pair \( \langle M, X \rangle \), where \( M \) is a set of methods and \( X \subset X_M \).

An event specification describes a set of related methods and their parameters that would likely form a meaningful parametric specification. Recall from Section 2.2.2 that an execution trace used for a fully automated mining system is comprehensive and unfocused. An event specification can be a means of turning such an unfocused trace into a focused one, by filtering out irrelevant events (i.e., method invocations) and parameters. For example, consider the following set of methods \( M \):

- \texttt{Collection.iterator(Collection, Iterator)}
- \texttt{Iterator.hasNext(Iterator, boolean)}
- \texttt{Iterator.next(Iterator, Object)}

Then, \( X_M \) is \( \langle \text{Collection, Iterator, Object} \rangle \), because \texttt{boolean} is a primitive type.
In the above definition, $X$ is a subset of $X_M$ because some parameters are insignificant and it is better to be removed. For example, `Iterator.next()` returns an object of `Object` type, but, considering that an element in a container does not play any role in any interaction between a `Collection` object and an `Iterator` object, it would be reasonable to drop that parameter.

In a parametric specification that JMiner eventually produces, an event specification fills the parameters of the specification and the event definitions, which are, for example, respectively written on line 1 and on lines 2–21 in the JavaMOP specification shown in Figure 2.2.

As shown in Figure 4.1, JMiner has a dedicated stage solely for mining event specifications because it is non-trivial and its result can be imprecise. Having this stage gives the user an opportunity to tune the inferred event specifications, if they or the parametric specifications they eventually result in are unsatisfactory. As mentioned in Section 3.1, many approaches implicitly assume that this stage is unnecessary because the provided execution trace is already focused. Since this thesis proposes a fully automated approach, JMiner could not make such assumption.

The second stage takes an event specification and parametric execution traces, which are not necessarily focused, as input, and yields a parametric specification as output. It is assumed that the given trace records method invocations from all
threads in chronological order, altogether. This stage first filters out all events and parameters that are irrelevant to the given event specification. From these filtered traces, the trace slicer extracts trace slices. Then, using these trace slices as positive samples, a property learner infers a property, and putting this property together with the event specification finally yields a parametric specification.

Since JMiner was motivated by the desire to use its output as the input to JA-MOP, the inferred specification is written in a form of the JA-MOP specification like one shown in Figure 4.2. By default, JMiner uses a property learner that infers an FSM and, consequently, every property is written as an FSM. However, one can use any property learners instead, as long as they take as input a set of strings, where a string corresponds to a trace slice in the context of JMiner.

4.2 Mining Event Specifications

To get an accurate parametric specification at the end, it is crucial to have a precise event specification. For example, consider the fragment of a parametric trace shown in Figure 4.2, and an event specification where the set of methods and the set of parameters are respectively all the methods and parameters that appear in the trace. If these inputs are fed into the second stage, the trace slicer would identify not only an actual interaction for \( \langle \text{Collection} \mapsto c_1, \text{Iterator} \mapsto i_1, \text{Object} \mapsto o_0 \rangle \), but also a spurious one for \( \langle \text{Collection} \mapsto c_2, \text{Iterator} \mapsto i_1, \text{Object} \mapsto o_0 \rangle \)—because the semantics is not considered, it is natural for a trace slicer to identify such a parameter binding from the fourth and fifth events. The trace slice for the spurious binding will then be [hasNext, next, add, hasNext], and a property learner would infer from this trace slice a property that permits the use of an iterator after a modification of the underlying collection, which causes a runtime exception. As this example shows, an inaccurate event specification can cause JMiner to produce a wrong parametric specification, even if the given parametric trace has no erroneous behaviors.

This example also implies that some deep knowledge, such as the semantics of classes and their methods, should be considered to infer accurate event specifications. Since such expert knowledge is not revealed through language constructs,
import java.util.*;
public class CheckForComodification {
    private static final int LENGTH = 10;
    public static void main(String[] args) throws Exception {
        List<Integer> list = new ArrayList<Integer>();
        for (int i = 0; i < LENGTH; i++)
            list.add(i);
        try {
            for (int i : list) {
                if (i == LENGTH - 2)
                    list.remove(i);
            }
        } catch (ConcurrentModificationException e) {
            return;
        }
        throw new RuntimeException("No ConcurrentModificationException");
    }
}

Figure 4.3: CheckForComodification.java, a unit test case in OpenJDK 6.

the event specification mining stage of jMiner attempts to extract such knowledge from programs using heuristics, which will be explained in this section.

4.2.1 Learning Related Methods and Parameters

This thesis proposes an automated technique that mines event specifications from unit test cases. The rationale behind this decision is that they are written and maintained by experts who have knowledge on that software, and each test case uses only classes and methods that are closely related. For example, the unit test case shown in Figure 4.3 mainly uses only two types: the ArrayList class, a subclass of Collection, and the Iterator interface, implicitly used in the for-each loop on lines 9–12. This isolation is typical in unit testing because a test case is written for a specific purpose; e.g., this case is written to check if a concurrent modification of a Collection object is detected and a runtime exception is raised. In contrast, a real world program is usually too complicated to identify closely related classes and methods.

The event specification learner takes as input unit test cases, such as one in Figure 4.3, and the package name of interest. For example, in order to mine event specifications in the java.util package of the Java API, the unit test cases for this package and the package name should be provided. The former is used to dynamically observe related methods, and the latter is used to filter out irrelevant classes and methods, as explained below.

28
Figure 4.4: Fragment of an execution trace from the test case shown in Figure 4.3.

While executing each unit test case, the learner records every method invocation. Unlike a typical parametric trace, defined in Section 2.2, a trace used in this stage has two additional pieces of information, in order to enable the learner to recognize the caller-callee relationships: the thread identifier and the depth of the call stack. For example, from the trace obtained from the test case shown in Figure 4.3, the learner can restore the call stack information, as shown in Figure 4.4. This trace contains iterator(), hasNext() and next() because a Java compiler translates a for-each loop, like the one on lines 9–12 in Figure 4.3, into an ordinary for loop that uses the Iterator interface.

To produce more meaningful results, the event specification learner removes irrelevant events because there can be events that just prepare the necessary context and, therefore, are irrelevant to the main purpose of the test case. Among many possible heuristics, two existing techniques, which have been used in Weimer and Necula [70], and Pradel and Gross [61], were chosen for this purpose: the learner discards events unless corresponding methods are defined in the package specified by the user as input; and it also discards events unless they are invoked by methods of the class that declares the main entry of the test case. The rationale behind these heuristics is that a unit test case is rarely for interactions for multiple packages and likely to consist of two parts: one core class for performing the actual test, and other helper classes for supporting the core class. The second heuristics can be implemented by considering the call stack of an observed trace. By applying these heuristics, ensureCapacity() and size() are discarded.

The remaining events may still consist of tangentially related interactions because a test case may have multiple steps that exercise similar classes and methods. To split such interactions, all the remaining events are then partitioned into groups of related events: two events are deemed directly related iff they share at least

---

1Both the thread identifier and the depth of the call stack are available through JVMTI [41].
one common argument, and related iff they are connected through a sequence of directly related events. For example, \(\langle \text{init} \rangle\) and \(\text{add}()\) are directly related due to \(\langle \text{ArrayList} \mapsto 689 \rangle\), and \(\langle \text{init} \rangle\) and \(\text{next}\) are related through \(\text{iterator}()\).

Considering that each partition is likely to be the smallest unit of an interaction intended by the expert, it is reasonable to anticipate a behavioral pattern from a set of methods involved in such an interaction. Therefore, the learner creates an event specification for each partition. Since a desired or undesired behavioral pattern is likely to be determined by interfaces, not by concrete implementations, the learner generalizes types; more specifically, for each object used as a target object, its type is generalized to the least specific type that specifies all methods involving that object. For example, from the trace shown in Figure 4.4, for the partition involving \(\langle \text{ArrayList} \mapsto 689 \rangle\) and \(\langle \text{AbstractList.Itr} \mapsto 950 \rangle\), \(\text{ArrayList}\) is generalized to \(\text{AbstractList}\) because \(\text{AbstractList}\) is the least specific type that specifies all the involved methods: \(\text{add}()\) and \(\text{iterator}()\). This generalization is also applied to methods; e.g., \(\text{ArrayList.add}()\) is generalized to \(\text{AbstractList.add}()\). Similarly, \(\text{AbstractList.Itr}\) is also generalized to \(\text{Iterator}\). As a result, from that trace, an (intermediate) event specification that has the three types \(\langle \text{AbstractList}, \text{Iterator}, \text{Object} \rangle\) and the following five event definitions is created:

- \(\text{AbstractList.}(\text{init})(\text{AbstractList})\)
- \(\text{AbstractList.add}(\text{AbstractList}, \text{Object})\)
- \(\text{AbstractList.iterator}(\text{AbstractList}, \text{Iterator})\)
- \(\text{Iterator.hasNext}(\text{Iterator})\)
- \(\text{Iterator.next}(\text{Iterator}, \text{Object})\)

It should be noted that this intermediate event specification includes \(\text{Object}\), which will be eliminated in the next step, explained in Section 4.2.2.

### 4.2.2 Filtering out Generics

The event specification inferred by the technique explained in Section 4.2.1 seems to work in that the collected event definitions are likely to obey a certain pattern and, consequently, form a parametric specification. However, its set of parameters is imprecise, causing the trace slicer to identify a spurious parameter binding, as explained at the beginning of Section 4.2. As explained there, the spurious interaction in the trace shown in Figure 4.2 is caused by the fourth and fifth events.

The direct cause of that spurious interaction is \(\langle \text{Object} \mapsto o_0 \rangle\), shared by both events, and that could be shared because \(\text{Object}\) is included in the set of parameters in the event specification—without this parameter, \(\langle \text{Object} \mapsto o_0 \rangle\) would be filtered out at jMINER’s second stage, as explained in Section 4.1. However, the more funda-
mental cause, in this particular case, would be the fact that an element of a container was considered as an object that plays a role in the container, although it does not.

Although it seems impossible to take such semantics into account in every case, the above problem can be avoided by recognizing parameters of generic types and excluding them from the event specification. This heuristics can be applied to other generic types, based on the assumption that, at the time the generic class was written, the instantiated types were unknown and, consequently, it is unlikely that a desired or undesired pattern depends on such unknown types.

To detect parameters of generic types, JMiner reads the generic signature, which tells the parameters of generic type and detects that the parameter of AbstractList .add() and the return type of Iterator.next() are generic. It then removes Object from the set of parameters, and completes an event specification.

### 4.2.3 Miscellaneous Filters

Other trivial filters are also applied to discard less important event definitions. The learner first discards event definitions corresponding to methods that are likely to be invoked anytime without affecting the legality of any behavioral pattern. One such example is toString()—it is safe to assume that this method does not have any side-effect and can be invoked anytime. Currently, the learner removes toString(), hashCode() and any getter that returns a primitive type. Here it discriminates between a primitive type and a non-primitive type because a primitive value cannot introduce any object that needs to obey any rule. In contrast, it may be illegal to invoke a certain method even if its target object is from a getter, or to invoke a getter unless a certain action has been done; e.g., Socket.getInputStream() looks like a getter, but it should be invoked only if the socket is connected.

It should be noted that it is tempting to remove pure methods from the event definitions, considering that these methods do not change the internal state of any object. However, this assumption does not always hold because a pure method can act as a guard. For example, Iterator.hasNext() is a pure method, but invoking it can be considered an important step for the caller to proceed to Iterator.next().

After removing unnecessary event definitions, the learner then eliminates an event specification if it contains only a constructor and one method, because it would result in an obvious pattern.

---

2Generic signatures are available through JVMTI [41].
3A getter is detected based on the name of a method; i.e., if the name of a method begins with “get”, it is considered a getter.
4Java does not have a notion of a pure method, although some languages have; e.g., C++ has a constant function, which can be declared by placing the “const” keyword after the parameter list.
5Although this depends on implementations, it would be unusual to write it as a non-pure method; at least, it is pure in both OpenJDK 6 and 7 [59].
4.3 Slicing Traces

As explained in Section 2.3.2, trace slicing for mining has two main challenges: overhead and spurious trace slices. This section explains the concept of complete and connected parameter bindings, introduced to remove spurious trace slices, and introduces a trace slicing algorithm with complexity analysis.

4.3.1 Complete and Connected Parameter Bindings

It is crucial to select only meaningful parameter bindings from all the possible ones because trace slices that correspond to meaningless parameter bindings would cause a property learner to infer an inaccurate property. To select only meaningful parameter bindings, jMiner considers two criteria: completeness and connectedness.

A parameter instance is complete if $\text{Dom}(\theta) = X$, where $X$ is the set of parameters in the given event specification. jMiner’s trace slicer suppresses all the trace slices that correspond to incomplete parameter bindings, because these bindings are partial and, consequently, insufficient to represent typical interactions. For example, consider the parametric trace shown in Figure 4.5 and $X = \langle \text{ArrayList}, \text{AbstractList.Itr} \rangle$. The trace slice for $\langle \text{ArrayList} \mapsto c_1 \rangle$, which is incomplete, is simply [add]; the second event iterator is not included because its parameter binding is not less informative than $\langle \text{ArrayList} \mapsto c_1 \rangle$.

It is possible that no events provide a complete parameter binding. One such example is illustrated in Figure 4.6, when $X = \langle \text{Socket, SocketInputStream, SocketOutputStream} \rangle$. The first four events are part of one interaction $\langle \text{Socket} \mapsto s_1, \text{SocketInputStream} \mapsto i_1, \text{SocketOutputStream} \mapsto o_1 \rangle$, but none of these events provide this complete parameter binding. This example shows that it may be necessary to combine parameter bindings from multiple events.

**Definition 6.** Two parameter bindings $\theta$ and $\theta'$ are compatible iff for any $x \in \text{Dom}(\theta) \cap \text{Dom}(\theta')$, $\theta(x) = \theta'(x)$. We can combine compatible parameter bindings
Socket.<init>(Socket↦s₁)
Socket.getInputStream(Socket↦s₁, SocketInputStream↦i₁)
Socket.getOutputStream(Socket↦s₁, SocketOutputStream↦o₁)
SocketInputStream.read(SocketInputStream↦i₁)
Socket.getOutputStream(Socket↦s₂, SocketOutputStream↦o₂)

Figure 4.6: Fragment of a parametric trace where no events provide a complete parameter binding.

\( \theta \) and \( \theta' \), written \( \theta \sqcup \theta' \):

\[
(\theta \sqcup \theta')(x) = \begin{cases} 
\theta(x) & \text{when } \theta(x) \text{ is defined} \\
\theta'(x) & \text{when } \theta'(x) \text{ is defined} \\
\text{undefined} & \text{otherwise}
\end{cases}
\]

That is, two parameter bindings disagreeing on any parameter are incompatible, and thus cannot be combined. For example, \( \langle \text{Socket} \mapsto s_1, \text{SocketInputStream} \mapsto i_1 \rangle \) is compatible with \( \langle \text{Socket} \mapsto s_1, \text{SocketOutputStream} \mapsto o_1 \rangle \), but is not compatible with \( \langle \text{Socket} \mapsto s_2, \text{SocketOutputStream} \mapsto o_2 \rangle \). \( \langle \text{Socket} \mapsto s_1, \text{SocketInputStream} \mapsto i_1 \rangle \sqcup \langle \text{Socket} \mapsto s_1, \text{SocketOutputStream} \mapsto o_1 \rangle \) yields a complete parameter binding \( \langle \text{Socket} \mapsto s_1, \text{SocketInputStream} \mapsto i_1, \text{SocketOutputStream} \mapsto o_1 \rangle \).

Although combining parameter bindings is necessary, it may introduce spurious ones, if done blindly. For example, \( \langle \text{Socket} \mapsto s_2, \text{SocketInputStream} \mapsto i_1, \text{SocketOutputStream} \mapsto o_2 \rangle \), obtained by combining \( \langle \text{Socket} \mapsto s_2, \text{SocketOutputStream} \mapsto o_2 \rangle \) and \( \langle \text{SocketInputStream} \mapsto i_1 \rangle \) in Figure 4.6, is deemed spurious because the trace shows no evidence that these bindings are related. To prevent combining such unrelated parameter bindings, this thesis introduces the concept of connected parameter bindings.

**Definition 7.** If \( \tau \in E(X)^* \), we define \( \tau \)-connectedness of parameter binding \( \theta \) as follows: 1) if \( e(\theta) \in \tau \) then \( \theta \) is \( \tau \)-connected; and 2) if \( \theta_1, \theta_2 \) are \( \tau \)-connected, compatible, and \( \theta_1 \cap \theta_2 \neq \bot \), then \( \theta_1 \cup \theta_2 \) is also \( \tau \)-connected.

According to the definition, a parameter binding is unconnected if it has any pair of parameter values that have no relation throughout the entire trace. For example, in the trace shown in Figure 4.6, \( \langle \text{Socket} \mapsto s_1, \text{SocketInputStream} \mapsto i_1, \text{SocketOutputStream} \mapsto o_1 \rangle \) is \( \tau \)-connected because of events 2 and 3, but \( \langle \text{Socket} \mapsto s_2, \text{SocketInputStream} \mapsto i_1 \rangle \) is not. By combining parameter bindings only if the combined one is connected, \$\text{MINER}\$’s trace slicer avoids creating spurious parameter bindings. In cases where there is no ambiguity, \( \tau \)-connected will be referred to as connected throughout this thesis.

It should be noted that it is not trivial to compute all possible connected parameter bindings in a parametric trace. One should not mistakenly think that this prob-
lem reduces to computing the ordinary connected components of a graph, where a vertex represents a parameter binding and an edge exists iff the two associated parameter bindings ($\theta_1$ and $\theta_2$) are compatible and $\theta_1 \cap \theta_2 \neq \bot$. Figure 4.7 shows one such graph, where $P$, $Q$, $R$ and $S$ are parameters, and $p_0, q_0, r_1, r_2$ and $s_0$ are parameter values. The graph-connected component in Figure 4.7 correctly suggests that $\langle P \mapsto p_0, Q \mapsto q_0, R \mapsto r_1 \rangle \sqcup \langle Q \mapsto q_0, R \mapsto r_1, S \mapsto s_0 \rangle$ is $\tau$-connected. However, it also suggests that $\langle P \mapsto p_0, Q \mapsto q_0 \rangle \sqcup \langle Q \mapsto q_0, R \mapsto r_2 \rangle \sqcup \langle R \mapsto r_1, S \mapsto s_0 \rangle$ is $\tau$-connected, which is wrong. Indeed, computing the graph-connected components does not take into consideration the compatibility between parameter bindings, while computing the $\tau$-connected parameter bindings must. For example, $\langle Q \mapsto q_0, R \mapsto r_1 \rangle$ and $\langle Q \mapsto q_0, R \mapsto r_2 \rangle$ are incompatible, but the standard graph-connected component fails to recognize it.

4.3.2 Complexity of Trace Slicing

This section explains the worst case complexity of the trace slicing problem in terms of the number of trace slices as a function of the total length $n$ of the given parametric trace and the size of $X$, the set of parameters. More precisely, it shows that there are approximately $\Omega\left(\frac{n}{m}\right)^m$ trace slices in the worst case when $m \geq 1$, where $m + 1$ is the size of $X$. Note that if $|X| = 1$ then there are at most $n$ trace slices and they are easy to compute. However, if $|X| = \frac{n}{2} + 1$ then there are $2^\frac{n}{2}$ trace slices, which shows that the addition of conflicting edges (like in Figure 4.7) makes the graph-connected component problem harder. The maximum of $\left(\frac{n}{m}\right)^m$ is actually reached when $m = \frac{n}{e}$, in which case it becomes $e^{\frac{m^m}{n}}$.

Suppose that $X = \{P_0, P_1, …, P_m\}$ for some $m > 0$ and that $\tau = e_1(\theta_1) e_2(\theta_2) … e_m(\theta_m)$. The worst case is when any two events have at least one common parameter value, so that $\theta \cap \theta' \neq \bot$ for any two parameter bindings $\theta$ and $\theta'$ such that $e(\theta), e'(\theta') \in \tau$; we can achieve that with minimal resources, by designating a pa-

---

6Analysis is approximate, such as making abstraction of the fact that $m$ may not divide $n$. 

---
rameter binding $\langle P_0 \mapsto p_0 \rangle$ and assuming that that is common to all events. Each event may be in conflict with a certain number of other events. For example, suppose that $e_1 \langle \theta_1 \rangle$ is in conflict with $a_1 - 1$ events on parameter $P_1$, where $a_1 > 0$. The other $a_1 - 1$ events are also in conflict with each other, so there is a “cluster” of $a_1$ events which are in conflict with each other on parameter $P_1$. The worst case is when the conflicting $a_1$ events are in conflict with no other event and when, for each trace slice corresponding to the remaining events, each of them yields a new trace slice. Thus, assuming that the remaining events generate $s$ trace slices, there are $a_1 \times s$ trace slices in total. We can iterate over the arguments above and obtain $a_1 \times a_2 \times \cdots \times a_m$ trace slices when we split the $n$ events of $\tau$ into clusters of $a_1, a_2, \ldots, a_m$ events with $a_1 + a_2 + \cdots + a_m = n$, each cluster containing those events conflicting on precisely one of the parameters $P_1, P_2, \ldots, P_m$, respectively. Note that this is not only an over-approximation; it can actually happen, as shown in Figure 4.8. The product is maximized when $a_1 = a_2 = \cdots = a_m = \frac{n}{m}$, in which case it becomes $\left( \frac{n}{m} \right)^m$.

If $X$ is not fixed, then one can actually fabricate an absolute worst-case scenario, which maximizes $\left( \frac{n}{m} \right)^m$. This case occurs when $m = \frac{n}{x}$, in which case the number of trace slices is exponential: $e^n$. However, if $X$ is fixed a priori, which is usually the case, we can only have a polynomial (in the length of the given parametric trace) number of trace slices. Although it is little likely in practice that the size of $X$ is correlated to the length of the trace, it is instructive to have a clear understanding of the worst-case complexity of the problem that this thesis attempts to solve.

### 4.3.3 Trace Slicing Algorithm

As discussed in Section 4.3.2, the number of trace slices is $\left( \frac{n}{m} \right)^m$ in the worst case. Since all trace slices can be distinct, this number gives a lower bound for all trace
slicing algorithms. This lower bound is hard to achieve, though, since computing complete and connected parameter bindings may require several operations of combining. For example, \( \langle p_0 \mapsto p_0, p_1 \mapsto p_{1,1}, p_2 \mapsto p_{2,1}, \ldots, p_m \mapsto p_{m,1} \rangle \) in Figure 4.8 can be obtained only after at least \( m \) combining operations: \((\langle p_0 \mapsto p_0, p_1 \mapsto p_{1,1} \rangle \sqcup \langle p_0 \mapsto p_0, p_2 \mapsto p_{2,1} \rangle \sqcup \langle p_0 \mapsto p_0, p_3 \mapsto p_{3,1} \rangle \sqcup \ldots \sqcup \langle p_0 \mapsto p_0, p_m \mapsto p_{m,1} \rangle )\). Furthermore, a trace slicing algorithm needs to search for compatible parameter bindings, which can be expensive, in order to create combined ones.

Figure 4.9 shows \( \text{jMiner} \)’s trace slicer, called Slicer. This trace slicer traverses the given parametric trace only once and does not output spurious trace slices, such as ones that correspond to incomplete or unconnected parameter bindings. It has two stages: it first processes the entire parametric trace, event by event, constructing intermediate results \( \Delta \); and then it constructs the set of trace slices \( \Psi \), each corresponding to a complete and connected parameter binding.

During the first stage, this algorithm stores in \( \Delta \) intermediate trace slices only for parameter bindings that are carried by events; i.e., it does not combine parameter bindings yet. The second stage, ConstructConnected, constructs \( \Omega \) holding all possible connected parameter bindings by combining compatible ones in the loop on lines 2–3. For each complete and connected parameter binding, its corresponding trace slice is finally constructed on lines 4–6. \( \Gamma \) collects all intermediate trace slices corresponding to \( \theta \)’s sub-bindings. MergeTraces is essentially the merge function of merge sort, using the position of events in the trace for comparison; recall that events in trace slices are listed chronologically.

**Theorem 1.** After running Slicer on \( \tau \in E(X)^* \),

1. \( \Psi(\theta) \) is defined iff \( \theta \) is \( \tau \)-connected and \( \text{Dom}(\theta) = X \);
2. If \( \Psi(\theta) \) is defined, then \( \Psi(\theta) = \tau |_\theta \).

This theorem states that all trace slices corresponding to \( \tau \)-connected and complete parameter bindings can be retrieved from \( \Psi \). Below is the proof of this theorem.

**Lemma 1.** After finishing the loop on lines 2–6 in ConstructConnected, a parameter binding \( \theta \) is \( \tau \)-connected iff \( \theta \in \Omega \).

**Proof.** \((\Rightarrow)\) According to Definition 7, all parameter bindings added to \( \Omega \) on line 1 in ConstructConnected are \( \tau \)-connected because \( \Delta(\theta) \) is defined only if \( e(\theta) \in \tau \). All parameter bindings added on line 3 are also \( \tau \)-connected because \( \theta_1, \theta_2 \) are \( \tau \)-connected and compatible, and \( \theta_1 \sqcap \theta_2 \neq \bot \) from the condition on line 2.

\((\Rightarrow)\) We prove this by well-founded induction on \( \subseteq \) because the minimal element \( \bot \) exists. Suppose that the property holds for all \( \theta' \) such that \( \theta' \subseteq \theta \). It must then be shown that the property holds for \( \theta \) as well. If \( \theta \) comes from an event like
Input: $X$, $\tau = e_1(\theta_1)e_2(\theta_2) \ldots e_n(\theta_n)$
Output: $\Psi \in \{[X \rightarrow V_X] \rightarrow E^\ast\}$
Global: $\Delta \in \{[X \rightarrow V_X] \rightarrow E^\ast\}$

Function SLICE()
1. for $i \leftarrow 1$ to $n$ do
   2. HANDLE_EVENT($e_i(\theta_i)$)
   3. CONSTRUCT_CONNECTED()

Function HANDLE_EVENT($e(\theta)$)
1. if $\Delta(\theta)$ undefined then
   2. $\Delta(\theta) \leftarrow e$
   3. $\Delta(\theta) \leftarrow \Delta(\theta)e$

Function CONSTRUCT_CONNECTED()
1. $\Omega \leftarrow \{\theta \mid \Delta(\theta) \text{ is defined}\}$
2. while $\exists \theta_1, \theta_2 \in \Omega$ compatible, $\theta_1 \cap \theta_2 \neq \bot$, $\theta_1 \sqcup \theta_2 \notin \Omega$ do
   3. $\Omega \leftarrow \Omega \cup \{\theta_1 \sqcup \theta_2\}$
4. foreach $\theta \in \Omega$ s.t. $\operatorname{Dom}(\theta) = X$ do
   5. $\Gamma = \{\Delta(\theta') \mid \theta' \sqsubseteq \theta \text{ and } \Delta(\theta') \text{ is defined}\}$
   6. $\Psi(\theta) \leftarrow \operatorname{MERGETRACES}(\Gamma)$

Figure 4.9: SLICER: Trace Slicing algorithm.

in the first case of Definition 7, then the property holds because $\theta$ belongs to $\Omega$ as per line 1 in CONSTRUCT_CONNECTED. If $\theta$ is $\theta_1 \sqcup \theta_2$ like in the second case of Definition 7, then both $\theta_1$ and $\theta_2$ belong to $\Omega$ by the induction hypothesis, resulting in $\theta \in \Omega$ as per line 3.

Lemma 2. After running SLICER, $\Psi(\theta)$ is defined iff $\theta$ is connected and $\operatorname{Dom}(\theta) = X$.

Proof. (⇒) Line 6 in CONSTRUCT_CONNECTED is the only place $\Psi(\theta)$ is defined. From the condition on line 4 and Lemma 3, $\theta$ is $\tau$-connected and $\operatorname{Dom}(\theta) = X$.

(⇐) From Lemma 3, $\Omega$ contains all $\tau$-connected parameter bindings. Therefore, if $\theta$ is $\tau$-connected and $\operatorname{Dom}(\theta) = X$, then the body of the loop on lines 4–6 is executed and, consequently, defines $\Psi(\theta)$.

Lemma 3. After running SLICER, if $\Psi(\theta)$ is defined, $\Psi(\theta) = \tau \upharpoonright \theta$.

Proof. We first show that $\Psi(\theta)$ preserves the order of events as in $\tau$. $\Delta(\theta')$ preserves the order because HANDLE_EVENT processes events by chronological order and line 3 appends each event to $\Delta(\theta')$. Since MERGETRACES is the same as the merge function of a merge sort and all input lists to MERGETRACES are sorted, the result of MERGETRACES is also sorted.
Now, showing that $\Psi(\theta)$ returned from MERGETRACES keeps the base event of $e'(\theta')$ iff $\theta' \subseteq \theta$ will complete the proof.

(\Rightarrow) After running HANDLEEVENT for all events in $\tau$, if there is an event $e'(\theta')$ in $\tau$, then line 2 in HANDLEEVENT defines $\Delta(\theta')$, resulting in $\Delta(\theta') \in \Gamma$ (line 5 in CONSTRUCTCONNECTED). Since line 2 in HANDLEEVENT stores the base event of $e'(\theta')$ in $\Delta(\theta')$, MERGETRACES dispatches the base event of $e'(\theta')$ to $\Psi(\theta)$.

(\Rightarrow) $\Delta(\theta')$ keeps an event only if its parameter binding is $\theta'$ (line 3 in HANDLEEVENT), and $\Delta(\theta')$ is considered to be merged only if $\theta' \subseteq \theta$ (line 5 in CONSTRUCTCONNECTED). Thus, $\Psi(\theta)$ keeps the base event of $e'(\theta')$ only if $\theta' \subseteq \theta$. \hfill \Box

From Lemma $\odot$ and Lemma $\odot$, Theorem $\odot$ holds.

Below the complexity of SLICER is analyzed. It first calls HANDLEEVENT $n$ times, and, assuming that a self-balancing binary search tree is used for $\Delta$, the complexity of HANDLEEVENT is $O(\log n)$. The loop on lines 2–3 in CONSTRUCTCONNECTED can pick $\theta_1$ and $\theta_2$ from $\Omega \times \Omega$, and each iteration takes $O(m)$ time for checking the compatibility and combining the two parameter bindings. There are $|\Omega|$ iterations of the loop on lines 4–6, with each iteration taking $O(m)$ time. The running time of the entire algorithm is thus $O(n \log n + |\Omega|^2 \cdot m + |\Omega| \cdot m) = O(n \log n + |\Omega|^2 \cdot m)$. Since the algorithm creates all possible connected parameter bindings, $|\Omega|$ can be calculated as follows: the number of connected ones with $|\text{Dom}(\theta)| = i + 1$ is $\binom{m}{i} \cdot \binom{n}{m}^i$ because we can choose $i$ parameters and there are $\frac{n}{m}$ parameter values for each parameter. Thus, we have $|\Omega| = \sum_{i=1}^m \binom{m}{i} \cdot \binom{n}{m}^i = \binom{n}{m} + 1)^m$, and the time complexity of SLICER is $O(n \log n + (\frac{n}{m} + 1)^{2m} \cdot m) = O((\frac{n}{m} + 1)^{2m} \cdot m)$. As for the space complexity, it needs to maintain $O(|\Omega|)$ connected parameter bindings of length $O(m)$ during trace slicing. It also needs space for $(\frac{n}{m})^m$ trace slices of size $m$ as illustrated in Figure 4.8. Therefore, the space complexity is $O((\frac{n}{m} + 1)^m \cdot m + (\frac{n}{m})^m \cdot m) = O((\frac{n}{m} + 1)^m \cdot m)$.

SLICER iterates through all possible connected parameter bindings in the loop on lines 2–3 in CONSTRUCTCONNECTED. Since it turned out that this step is expensive, two optimizations have been applied. First, instead of blindly picking a pair of parameter bindings from $\Omega$ and combining them, the implementation proceeds in a bottom-up manner. At the first step, it picks two parameter bindings ($\theta_1$ and $\theta_2$) such that $|\text{Dom}(\theta_1)| = |\text{Dom}(\theta_2)| = N$, and creates $\theta_1 \cap \theta_2$, if necessary. After handling all parameter bindings with $N$ parameter bindings, it picks parameter bindings with $N + 1$ parameter bindings, and so on, until $N$ reaches the size of $X$, the set of parameters. This way, a parameter binding is considered for compatibility within only a limited window, reducing the number of iterations.

The second optimization is to group parameter bindings so that all parameter bindings in the same group bind exactly the same parameter values. Grouping also reduces the number of iterations on lines 2–3 in CONSTRUCTCONNECTED. For ex-
ample, if $\langle p \mapsto p_1, q \mapsto q_1 \rangle$ is chosen as $\theta_1$, all parameter bindings that belong to
the group corresponding to $\{R, S\}$ will be excluded from the list of candidates for $\theta_2$
because any parameter binding in this group would result in $\theta_1 \cap \theta_2 = \bot$.

4.4 Learning Parametric Specifications

A property learner takes as input a set of trace slices, generated by the trace slicer,
and infers a property. As defined in Definition 4, a trace slice is non-parametric; it
is merely a string in $E^*$. Therefore, any learner that takes a set of strings as input can
be employed as a property learner in jMiner. That is, one can use an algorithm that
is parameter-agnostic.

By default, jMiner uses a property learner based on a PFSA learner. This de-
default learner first runs an off-the-shelf PFSA learner and then refines the inferred
automaton. This section explains each step of this default learner.

4.4.1 Probabilistic Finite State Automata (PFSA) Learner

A PFSA is an FSA where each transition is labelled with how often the transition
occurs. A PFSA learner takes a set of strings as input and infers a PFSA. Several
PFSA learning approaches have been proposed, and some of them, such as Bier-
mann and Feldman [11], and Raman et al. [63], have been used to infer FSAs in
the context of specification mining. jMiner’s default learner adopts the sk-strings
algorithm [63], which is described below.

The sk-strings algorithm first constructs a PFSA that precisely accepts the
given set of strings. Each transition is then annotated with a frequency, saying how
many times that transition was observed. It then generalizes by merging states that
are sk-equivalent: two states are sk-equivalent iff corresponding sets of bounded
strings (ones that are frequently generated from each of the two states) are matched.
As a result of this approximation, two states can be merged even when they are not
strictly equivalent, making it possible for the inferred PFSA to accept not only the
input strings but also other “similar” strings.

After running the sk-strings algorithm, jMiner’s default specification learner
drops the frequency information, yielding an ordinary FSA. As an example, consider
the parametric trace shown in Figure 4.3 and $X = \langle$ArrayList, AbstractList.Itr\rangle.
The trace slicer then produces two trace slices: [add, iterator, hasNext, next, hasNext, next] from events 1, 2, 3, 4, 5 and 9; and [iterator, hasNext, next] from events 6, 7 and
8. From these trace slices, the sk-strings algorithm would infer the PFSA shown
in Figure 4.10; the frequency information is not shown here for simplicity.
4.4.2 Finite State Automata (FSA) Refiner

Although PFSA learner’s approximations are generally desirable in many application domains, the resulting FSA turned out to often be overly general in the domain of specification mining, in that the inferred FSA accepts undesirable trace slices. For example, consider the following trace slice:

```
⟨init⟩, iterator, hasNext, next, iterator, hasNext
```

The inferred FSA in Figure 4.10 is misleading because it accepts the above trace slice, which is infeasible because only one iterator event can be observed for any pair of a Collection object and an Iterator object, considering either the semantics or any observed behavior. To avoid such over-generalized and misleading properties, the default learner refines the FSA, first inferred by the SK-STRINGS algorithm.

The goal of the refiner is to eliminate transitions caused by over-generalization, while keeping desirably generalized transitions. An obvious step for avoiding over-generalization is to remove all the transitions that are never taken by any of the trace slices, provided as the input of the property learner. For example, the iterator transition from state 4 to state 2 in Figure 4.10 can be safely removed because it is never taken—the same Iterator object cannot be created twice. However, this obvious step is insufficient in that the resulting FSA still accepts infeasible interactions that contain multiple iterator events; e.g., ⟨init⟩, iterator, hasNext, add, iterator, hasNext.

The fundamental problem stems from the fact that a PFSA learner does not take into account the context of behavioral patterns when merging states. For example, the inferred FSA shows that both states 1 and 3 can move to state 1 by receiving add, because the PFSA learner has merged two contextually different states into state
Input: automaton $A = (S, E, i, \delta : [S \times E \rightarrow S], F)$, traces $T \subseteq E^*$
Output: automaton $A_r$
Locals: automaton $A' = (S', E, i', \delta', F')$, state $s, s'$, transition function $\delta_r$

Function Main()
1. $A' \leftarrow$ EXPAND($A$)
2. $\delta_r \leftarrow \perp$
3. foreach $\tau \in T$ do
4.   $s \leftarrow i'$
5.   foreach $e \in \tau$ do
6.     $s' \leftarrow s; \; s \leftarrow \delta'(s, e); \; \delta_r(s', e) \leftarrow s$
7.     if $\delta_r = \delta'$ then goto 8
8. $A' \leftarrow (S', E, i', \delta', F')$
9. $A_r \leftarrow$ MERGEIDENTICALSTATES($A'$)

Function EXPAND($A$)
Input: automaton $A = (S, E, i, \delta, F)$
Output: automaton $A' = (S', E, i', \delta', F')$
Locals: integer $n$; set of states $D$; map $\gamma : S \rightarrow 2^{S'}$
Initial: $S' \leftarrow \emptyset, F' \leftarrow \emptyset, \delta' \leftarrow \perp$
1. foreach $s \in S$ do
2.   $n \leftarrow$ COUNTINCOMINGEDGES($s, A$)
3.   if $s = i$ then $n \leftarrow n + 1$
4.   $D \leftarrow$ GETFRESHSTATES($n$)
5.   $S' \leftarrow D \cup S'$
6.   $\gamma(s) \leftarrow D$
7. foreach $s \in S$ do
8.   foreach $s' \neq s \in S$ s.t. $\delta(s', e) = s$ for some $e$ do
9.     $s'' \leftarrow$ PICKONEWITHNOINCOMINGEDGE($\gamma(s), \delta'$)
10.    foreach $s'' \in \gamma(s')$ do $\delta'(s'', e) = s''$
11.   if $s \in F$ then $F' \leftarrow F' \cup \gamma(s)$
12.   if $s = i$ then $i' \leftarrow$ PICKONEWITHNOINCOMINGEDGE($\gamma(s), \delta'$)
13. return $A'$

Figure 4.11: Refiner: FSA refining algorithm.

1: one for indicating add before an Iterator object is created; and the other for indicating add after an Iterator object has been created.

To avoid such undesirable merging, this thesis presents a refining algorithm, shown in Figure 4.11. Refiner expands each state to distinguish incoming states; more precisely, if a state $s$ has $n$ incoming edges from the other states, then $s$ is
replaced by $n$ corresponding states ($s_1, s_2, ..., s_n$). The mapping from $s$ to the corresponding set of newly created states is maintained in $\gamma$ (lines 4–6 in EXPAND).

EXPAND builds transitions in the new automaton (lines 7–12): if $\delta(s', e) = s$ is a transition in the inferred automaton and $s \neq s'$, then it chooses a state $s''$ from $\gamma(s)$ with no incoming edges at this point, and adds transitions from every state in $\gamma(s')$ to $s''$. If $s$ is a final state, then all states in $\gamma(s)$ are also final; and if $s$ is the initial state, then it chooses a state from $\gamma(s)$ with no incoming edges as the new initial state. This way, the original automaton is expanded to an equivalent automaton in which every state has a set of incoming edges, each of which corresponds to one incoming edge in the original automaton. As an example, Figure 4.12 shows the expanded FSA of the one in Figure 4.10. This expansion provides a partial context: state 1$_1$ corresponds to the case when no Iterator objects have been created, whereas states 1$_2$ and 1$_3$ correspond to the other case.

Refiner then simplifies the expanded FSA by removing all the transitions that are never taken by the given trace slices (lines 3–8 in MAIN); e.g., $1_2 \rightarrow 2_1, 1_3 \rightarrow 2_1,$ and $4_1 \rightarrow 2_2$. It also eliminates unreachable states, and merges states that have the same outgoing transitions (line 9); e.g., state 2$_2$ is eliminated, and states 1$_2$, 1$_3$ and states 3$_1$, 3$_2$ are merged, respectively. The resulting FSA for the expanded one in Figure 4.12 is shown in Figure 4.13.

Theorem 2 shows that the refined FSA accepts all the observed trace slices and, possibly, others.
**Theorem 2.** With the notation in Figure 4.11, if $T \subseteq L(A)$ and $A'$ is the automaton after running REFINER, then $T \subseteq L(A') \subseteq L(A)$.

**Proof.** If $\delta(s', e) = s$ exists in $A$, the loop on lines 8–10 in EXPAND introduces $\delta'(s'', e) = s''$ in $A'$ where $s'' \in \gamma(s')$ and $s'' \in \gamma(s)$. EXPAND chooses $i' \in \gamma(i)$ as the initial state of $A'$ on line 12. It also marks all elements of $\gamma(s)$ as final states of $A'$ on line 11 if $s$ is one of the final states in $A$. Then, for each symbol of $\omega \in T$, if $A$ transitions from $s_i$ to $s_j$ according to $\delta$, $A'$ also transitions from $s'_i \in \gamma(s_i)$ to $s'_j \in \gamma(s_j)$ according to $\delta'$. Thus, if $A$ reaches $s_f$, then $A'$ reaches $s'_f \in \gamma(s_f)$. If $s_f$ is one of the final states of $A$, $s'_f$ is one of the final states of $A'$. Since all transitions needed to accept all strings in $T$ are in $\delta_r$, if $A'$ reaches $s'_f$ using $\delta'$, then $A'$ can also reach $s'_f$ using only $\delta_r$, for any $\omega \in T$. Therefore, $T \subseteq L(A')$.

Next, we show that $L(A') \subseteq L(A)$. Based on the way EXPAND creates states of $A'$, each state $s'$ in $A'$ has one corresponding state $s$ in $A$, where $s' \in \gamma(s)$. For this reason, if $A'$ transitions from $s'_i$ to $s'_j$, then $A$ transitions from $s_i$ to $s_j$, where $s'_i \in \gamma(s_i)$ and $s'_j \in \gamma(s_j)$. Similarly, the initial state and the final states in $A'$ have corresponding states in $A$. Therefore, if a string is accepted by $A'$, that string is also accepted by $A$; i.e., $L(A') \subseteq L(A)$. \hfill \Box

**4.5 Evaluation of jMINER**

This section evaluates jMINER’s performance and usefulness. Since trace slicing, which enables jMINER to observe each interaction separately and infer an accurate specification no matter how interactions overlap, is a both important and expensive step, it first compares the performance of jMINER’s trace slicer with that of
4.5.1 Performance of Slicer

It might be questionable if Slicer, designed for specification mining, is any faster than a trace slicing algorithm for monitoring with changes for recording events and removing trace slices that correspond to unconnected parameter bindings.

To validate that Slicer is more efficient, a preliminary experiment was conducted. Since this experiment is solely for performance measurement, manually written and selected event specifications were used. For the trace slicer for monitoring, $C(X)$ from Chen and Roşu [21] was chosen because this is believed to be the most efficient algorithm. It should be noted that another algorithm, proposed in Chen et al. [22], outperforms $C(X)$, but this requires additional input that cannot be easily inferred and, therefore, it is not reasonable to assume such input, in the context of mining. This experiment was carried out on a Ubuntu Linux 7.10 machine with 1.5GB RAM and a Pentium 4 2.66GHz processor.

Table 4.1 summarizes test cases and their results. The first two columns show how the inputs were prepared: the first column shows classes that predefined event specifications were written for; and the second column gives the average length of parametric traces. These parametric traces were respectively observed by executing Apache JAMES [5] test cases, DaCapo’s benchmarks [12], Apache Lucene [6] test cases, and DaCapo’s benchmarks.

The third column shows the number of trace slices for all (both connected and unconnected) parameter bindings extracted from the given parametric traces. The numbers are given as ranges because the number of trace slices varies among programs. The last two columns represent the average elapsed time for trace slicing. Slicer always outperforms $C(X)$; “$> 10,000$” means that the trace slicer did not finish in 3 hours. In particular, when there were multiple parameters and parametric traces were long, the difference was significant. This is because Slicer combines

<table>
<thead>
<tr>
<th>Classes</th>
<th>Length of Traces</th>
<th>Number of Trace Slices</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CommandHandler, SMTPSession</td>
<td>1 K</td>
<td>11~29</td>
<td>&lt; 1</td>
</tr>
<tr>
<td>Reader, Writer</td>
<td>9 K</td>
<td>1,143~2,285</td>
<td>4</td>
</tr>
<tr>
<td>Document, IndexWriter</td>
<td>29 K</td>
<td>792~11,905</td>
<td>&gt; 10,000</td>
</tr>
<tr>
<td>Collection, Map, Iterator</td>
<td>50 K</td>
<td>79~90,775</td>
<td>757</td>
</tr>
</tbody>
</table>

Table 4.1: Comparison between $C(X)$ and Slicer

$C(X)$ [21], devised for runtime monitoring. Then, it shows the result of running JMiner, without any human intervention, on four packages of OpenJDK 6 [59]: java.io, java.lang, java.net, and java.util.
Table 4.2: Parametric traces used for the experiments.

<table>
<thead>
<tr>
<th>Packages</th>
<th># Test cases</th>
<th># Events</th>
<th># Programs</th>
<th># Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.io</td>
<td>382</td>
<td>28,835,588</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang</td>
<td>372</td>
<td>41,784,568</td>
<td>14</td>
<td>88,999,435</td>
</tr>
<tr>
<td>java.util</td>
<td>370</td>
<td>65,854,349</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.net</td>
<td>221</td>
<td>9,429,744</td>
<td>31</td>
<td>10,938,168</td>
</tr>
</tbody>
</table>

parameter bindings only if they are connected and combining them is also delayed until all the events are read, as explained in Section 4.3.3, whereas $\mathbb{C}(X)$ has to eagerly combine parameter bindings for immediate reaction and, as a result, some of combined ones are spurious.

### 4.5.2 Automated Specification Mining

JMiner is capable of inferring parametric specifications automatically, as long as unit test cases for classes of interest and programs that exercise these classes are provided as input. To see whether such automated mode can yield meaningful specifications, a full-scale experiment was also conducted.

This experiment was performed on four widely used packages in OpenJDK 6: java.io, java.lang, java.net, and java.util. OpenJDK 6 was chosen because it contains various unit test cases and its documentation, called the API specification (Section 5.1), has informal but valuable information on behavioral patterns, which enables objective assessment of the inferred parametric specifications.

Parametric traces were obtained from all the benchmarks in the DaCapo benchmark suite 9.12 [12] and all the test cases in Apache JAMES Server 2.3.1 [5]. The former was chosen because it has non-trivial benchmarks that are likely to exercise many classes in OpenJDK 6, especially in java.io, java.lang, and java.util. Being a mail server, the latter has test cases that exercise classes in java.net.

To obtain parametric traces for both event specification mining (Section 4.2) and trace slicing (Section 4.3), a JVMTI [11] agent was written and used. This agent produces a comprehensive parametric trace; i.e., for each method invocation from any class, it records a parametric event: the method name and the declaring class for the base event part; and the target object, arguments, and the return value for the parameter binding part.

Running OpenJDK 6’s unit test cases for each package is straightforward because they are well structured in the source directory. The default input of the DaCapo benchmark suite was used. For some benchmarks in this suite, the execution
Table 4.3: Inferred event specifications and parametric specifications.

<table>
<thead>
<tr>
<th>Package</th>
<th># Event Specifications</th>
<th># Parametric Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.io</td>
<td>145</td>
<td>66</td>
</tr>
<tr>
<td>java.lang</td>
<td>82</td>
<td>48</td>
</tr>
<tr>
<td>java.util</td>
<td>181</td>
<td>80</td>
</tr>
<tr>
<td>java.net</td>
<td>90</td>
<td>36</td>
</tr>
</tbody>
</table>

Table 4.4: Execution time (minutes).

<table>
<thead>
<tr>
<th>Package</th>
<th>Event Specification Learner</th>
<th>Trace Slicer</th>
<th>Property Learner</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.io</td>
<td>24</td>
<td>115</td>
<td>24</td>
</tr>
<tr>
<td>java.lang</td>
<td>38</td>
<td>112</td>
<td>75</td>
</tr>
<tr>
<td>java.util</td>
<td>59</td>
<td>133</td>
<td>86</td>
</tr>
<tr>
<td>java.net</td>
<td>59</td>
<td>14</td>
<td>1</td>
</tr>
</tbody>
</table>

time was limited to one hour, because millions of events occur and significant overhead for recording them caused the execution to take more than an hour. Table 4.2 shows statistics on parametric traces generated from OpenJDK 6’s test cases, Apache JAMES test cases, and DaCapo’s benchmarks.

From parametric traces from OpenJDK 6’s test cases, the event specification mining stage automatically inferred 498 event specifications. Among them, 230 event specifications resulted in parametric specifications as shown in Table 4.3. Parametric specifications could not be inferred from the other event specifications because neither DaCapo’s benchmarks nor Apache JAMES test cases had any interaction regarding them.

Table 4.4 shows the execution time for three components in jMiner: the event specification learner (Section 4.2), the trace slicer (Section 4.3), and the default property learner (Section 4.4). The experiment was conducted under a Windows machine with 1GB RAM and a Pentium 3GHz processor. The numbers in this table do not include the time spent on running unit test cases or applications; i.e., they are the pure overheads of jMiner components. Each number represents the total elapsed time; e.g., learning 145 event specifications for java.io took 24 minutes. Trace slicing accounted for most of the time except java.net, which has relatively fewer events and interactions.

Below are parametric specifications that were automatically mined by jMiner and manually validated. Overall, jMiner was able to mine several useful parametric specifications, although it also mined too simple or complicated ones. Many of these simple specifications are caused by the fact that the training set (i.e., DaCapo’s
benchmarks and Apache JAMES test cases) covers only part of event definitions in an event specification and, therefore, the observed patterns are partial. In contrast, many of complicated ones are caused by unnecessary event definitions that correspond to methods that can be invoked anytime. From such methods, JMINER would observe application-specific patterns and, considering that multiple programs were used, eventually infer a complicated property by combining those patterns. More specifications can be found at the JMINER webpage [45].

**Client Socket**

Figure 4.14 shows a parametric specification of a client-side stream socket. The constructor of Socket connects a new socket to the peer specified by its arguments. Then, `getInputStream()` and `getOutputStream()` return the input and the output stream, respectively, which enable data transmission using `read()` and `write()`. The specification states that data transmission can be repeatedly performed in arbitrary order until the socket is closed, which is consistent with the documentation. It also states that `close()` can be invoked multiple times, which is undocumented but correct. The specification also correctly suggests that the invocation of `close()` is optional because states 4 and 5 are also final states. In fact, calling `close()` is recommended, but not mandatory because the connection is eventually closed when the `Socket` object is reclaimed.
Figure 4.15: ServerSocket specification inferred by jMiner.

**Server Socket**

Figure 4.13 shows a specification for the server-side socket. After a ServerSocket object \( l \) is instantiated, accept() listens for a connection and accepts it, returning a new socket \( e \). getInputStream() and getOutputStream() return an InputStream object \( i \) and an OutputStream object \( o \) respectively, which can be used for data transmission. After these operations, close() can be invoked to close the connection. This behavior spans over multiple threads in most cases because multiple clients can connect to the same port represented by a single ServerSocket object, and a server needs to handle them concurrently. The trace slices used in the experiments indeed involved two threads: the data transfer was processed in a separate thread. If each thread’s trace was considered separately like in other approaches, such as [61], this specification could not be mined.

**Collection, Iterator**

Figure 4.13 shows a specification of Collection and Iterator. This specification correctly states the safety property of Collection, mentioned in Section 2.1, although it is not as comprehensive and succinct as the hand-written specification shown in Figure 2.2—the automatically inferred one does not consider clear(), offer() and so forth; and it unnecessarily distinguishes between hasNext() and next(). Yet, the inferred specification is capable of detecting bugs, and it can be also easily improved.
Reader, Writer

Figure 4.16 shows a specification of a Reader object, stating that read() can be repeatedly called before close(). It correctly does not enforce the invocation of close(), similarly to the Socket specification above. JMinER also mined a similar specification for Writer. These specifications are simple, but can detect an illegal invocation of read() or write() after close().
Chapter 5

Writing Parametric Specifications From Documentation

Although the automated approach to specification mining, explained in Chapter [4], was capable of inferring several useful parametric specifications, it also yielded inaccurate ones, which may result in false positives or negatives. Since the ultimate goal of this thesis is to achieve a preparation-free and comprehensive runtime verification tool, such inaccuracy would be problematic. As an alternative approach, this chapter presents the results of manual effort for writing parametric specifications from documentation.

5.1 Approach Overview

This section explains additional background knowledge on documentation and then provides a brief overview of the manual approach.

A Java platform, such as Java Platform Standard Edition 6, implements various libraries that are commonly needed to implement applications, such as data structures (e.g., List and HashMap), and I/O functions (e.g., FileInputStream and FileOutputStream). Besides such library implementations, a Java platform provides the API Specification, which describes all aspects of the behavior of each method on which user's programs may rely [37]. For example, the API specification for the PipedInputStream class states:

Typically, data is read from a PipedInputStream object by one thread and data is written to the corresponding PipedOutputStream by some other thread. Attempting to use both objects from a single thread is not recommended, as it may deadlock the thread. The piped input stream contains a buffer, decoupling read operations from write operations, within limits.

Ideally, the API specification includes a comprehensive set of contracts between callers (i.e., user's programs) and implementations, but this ideal is hard to achieve and the current API specification may miss some important contracts [37]. Nevertheless, the API specification is undoubtedly a good source for formalizing the Java API because it is well maintained and thoroughly written—for example, there are
255,331 words in the API specifications for four packages that this thesis covered: java.io, java.lang, java.net and java.util.

While the API specification implies desirable or undesirable behaviors, such information cannot be utilized by formal analysis tools, such as JAVAMOP, because it is written in plain English. To enable these tools to utilize such information, it is necessary to describe the implied specifications in a certain formal language, such as the JAVAMOP specification syntax.

An API specification is mostly written in documentation comments embedded in the Java source code; e.g., the documentation comment containing the above quote is embedded in PipedInputStream.java. A documentation comment, starting with /** and ending with */ , is written in HTML with a few extensions, such as the {@link} tag. Javadoc, a tool included in the Java Development Kit (JDK), extracts these comments and generates an API specification, which is typically a set of interlinked HTML pages.

Although the API specification contains information on specifications, statements that describe such information are scattered around the entire source code—some are placed before a class, and others are placed before a method or field—and it is prone to overlook them.

Another difficulty stems from the fact that specification-implying statements and others for explaining the functionality of a certain class or method, are mingled in a documentation comment. Also, in a documentation comment, there is no mark that makes them distinguishable—it is common that statements for different purposes are placed in a paragraph. Also, those specification-implying statements do not always describe specifications clearly and, thus, it is often non-trivial to write precise parametric specifications from such statements.

To avoid overlooking specification-implying statements and writing incorrect specifications, we chose a systematic approach: we marked what has been covered with special tags, put a link to the written parametric specification, and kept track of status using our own program, called PropDoc.

Figure 5.1 shows the proposed procedure. From the API specification in documentation comments, we marked each chunk of specification-implying text by wrapping it with a pair of special tags (Section 5.2.1): {@property.open} and {@property.close}. We then wrote a parametric specification from such text, as explained in Section 5.2.2, and added to the special tag a link to this specification. PropDoc, an extension of Javadoc, reads the annotated source code, and generates an augmented API specification that highlights what has not been covered and what has been covered but does not have corresponding parametric specifications. The generated document guides us to cover the entire API specification. Besides, from the user’s perspective, it can be used as more informative documentation.
5.2 Formalizing the Java API

This section explains the methodology used to write parametric specifications from the API specification. The entire API specification of Java Platform Standard Edition 6 has been inspected and all runtime-monitorable specifications have been written.

5.2.1 Separating Specification-Implying Text

As explained in Section 5.1, sentences for different purposes are mingled in the API specification, and we first separated specification-implying text from others. Consider the following paragraph written for one of PipedInputStream's constructors:

Creates a PipedInputStream so that it is connected to the piped output stream src. Data bytes written to src will then be available as input from this stream.
Unlike the quoted text mentioned in Section 2.1, where we could infer a formal specification shown in Figure 2.2, the above chunk of text does not describe any desired or undesired API usage pattern—it merely describes the functionality, and we call it *descriptive*. In contrast, we call a chunk of text that implies a specification, such as the one mentioned in Section 2.1, *specification-implying*.

While it might seem trivial to make distinction between specification-implying and descriptive text, there are unclear cases. One such example is in the API specification for `FileInputStream.available()`:

> Returns an estimate of the number of remaining bytes that can be read (or skipped over) from this input stream without blocking by the next invocation of a method for this input stream.

This implies the consequence of calling `read()` when `available()` returns 0—the calling thread would block. Although it describes the behavior of an input stream, we do not consider it as specification-implying text because the desirable behavior is not clearly implied. One may be tempted to write a formal specification that prevents calling `read()` in such case, but it might be against one's intention because many multi-threaded programs use blocking I/O.

Another unclear case is a description of conditions that involve external environments. One such example is included in the API specification for the constructor of `FileOutputStream`, as follows:

> If the file exists but is a directory rather than a regular file, does not exist but cannot be created, or cannot be opened for any other reason then a `FileNotFoundException` is thrown.

Given that avoiding a runtime exception is desirable, one might think that this implies a specification: check if a directory exists, or a file does not exist but a file cannot be created or opened, before creating a `FileOutputStream` object. However, we do not consider that this implies a formal specification because the state of the file system externally and dynamically changes without notifying the runtime monitoring system and, consequently, it is impossible to reliably check whether a file can be created or opened.

It is difficult to formalize a specific set of rules that resolves all of the unclear cases, but the rule of thumb was that a chunk of text is specification-implying only if a desirable or undesirable behavior is apparent and it is defined in terms of noticeable events, such as class loadings, method invocations, and field accesses.

### 5.2.2 Writing Formal Specifications

Based on specification-implying text extracted from the API specification, we wrote JAVAMOP specifications. As explained in Section 2.1 and shown in Figure 2.2, a typ-
ical formal specification contains three parts: event definitions, a desirable or undesirable behavioral pattern (i.e., property), and a handler. An event in the written specifications is mostly a method invocation, but it is also a field access, an end of an execution, or a construction of an object. We expressed a property in either an ERE, a FSM or an LTL formula. Depending on the pattern, we tried to choose the most intuitive formalism. Our handlers simply output a warning message in case of a violation, but one can easily alter this behavior by editing them.

However, for some specifications, an occurrence of an event, in any context, indicates a violation. In such cases, we omitted the property and the handler, and let the event definition directly output warning messages, as will be shown in Figure 5.4. Similar to handlers, one can alter this behavior since the body of each event definition can also contain arbitrary Java code.

Below we give a few cases where we intentionally did not formalize for the purpose of this thesis.

Non-monitorable behaviors

We formalized only runtime-monitorable specifications because we intended to use JavaMOP, a runtime monitoring system. Consider the following API specification for Comparable.compareTo():

The implementor must also ensure that the relation is transitive:

\[(x \text{.compareTo}(y) > 0 \&\& y \text{.compareTo}(z) > 0) \implies x \text{.compareTo}(z) > 0.\]

Although this implies a certain behavior, checking if it holds is infeasible at runtime. Not having a means of describing and checking it, we did not formalize such cases.

Unsupported monitoring

Among runtime-monitorable behaviors, there are a few cases where monitoring systems are incapable of observing necessary events. For example, the API specification for InputStream.available() states:

Note that while some implementations of InputStream will return the total number of bytes in the stream, many will not. It is never correct to use the return value of this method to allocate a buffer intended to hold all data in this stream.

It is ideal to keep track of uses of the return value of available() and check if any of them or any variable affected by them is used to allocate a buffer. Apart from performance degradation it causes, however, most runtime monitoring systems do not support local variable tracking. Thus, we did not formalize such cases.
Already enforced behaviors

Other cases that we did not formalize include those where the desirable behavior is enforced by compilers. For example, the API specification of InputStream states the requirement of its subclass:

Applications that need to define a subclass of InputStream must always provide a method that returns the next byte of input.

Java compilers enforce the requirement because read(), the method implied by the above quote, is an abstract method. Such guarantee obviates the need for additional runtime check; thus, we did not formalize such cases.

Internal behaviors

When all the events in the implied specification are never exposed to clients (e.g., they are private method invocations or field accesses), we did not formalize. Consider the API specification for GregorianCalendar.getYearOffsetInMillis():

This Calendar object must have been normalized.

One could write a JAVAMOP specification that checks if a method for normalizing the calendar object has been invoked, but it is useless because user's programs cannot invoke this method anyway, due to the access control—it is defined as private. Although JAVAMOP is capable of monitoring them, we decided not to formalize them because there is no benefit from a user's perspective.

5.2.3 Classifying Formal Specifications

The formal specifications implied by the API specification have many different characteristics. For example, a violation of some specifications merely indicates a bad practice, not a severe error. To allow users to look up such specifications and conveniently suppress violations of them, we classified the written specifications according to a few criteria.

Severity

According to the severity of a violation, we classified specifications into three groups: suggestion, warning and error. We use suggestion if a violation is merely a bad practice. StringBuffer_SingleThreadUsage, which will be discussed in Section 5.2.4, is one such specification. If a violation is not necessarily erroneous but potentially wrong, we use warning; e.g., PipedStream_SingleThread (Section 5.2.4) and Serializable_UID (Section 5.2.4). We use the last group, error, if a violation indicates an error; e.g., ShutdownHook_PrematureStart (Section 5.2.4).
Guarantee of the underlying system

Depending on what the underlying system (including the JVM and the Java Class Library) guarantees, we classified formalized specifications into three groups: always-check, sometimes-check and do-not-check. An example of the first group is a specification that warns a write operation on a closed FileOutputStream object, which is always caught by the system. The fail-fast behavior of an Iterator object is an example of the second group: a fail-fast iterator throws an exception if the underlying collection is structurally modified, but this behavior is not guaranteed. PipedStream_SingleThread (Section 5.2.4) and StringBuffer_SingleThreadUsage (Section 5.2.4) belong to the do-not-check group; the system never warns any violation.

False alarm

The last criterion is whether a violation can be a false alarm due to the incompleteness of a specification. If a specification does not have a false alarm, we classified it as no-false-alarm; otherwise, as false-alarm. An example of false-alarm is Console_FillZeroPassword, shown in Figure 5.6. This specification needs to check if the application zeroes the buffer for holding password, but it cannot always capture zeroing because there are arbitrarily many ways—for example, one can write a loop explicitly, which is difficult for a runtime monitoring system to detect.

5.2.4 Examples

We could write total 179 specifications. We believe that they are all the runtime-monitorable specifications implied in the API specification of Java Platform Standard Edition 6. A few examples are explained below. All the specifications are available at the project website: http://fsl.cs.uiuc.edu/annotated-java/.

PipedStream_SingleThread

This specification, shown in Figure 5.2, warns if a thread attempts to use both a PipedInputStream object and a PipedOutputStream object. It is based on the API specification for PipedInputStream:

Typically, data is read from a PipedInputStream object by one thread and data is written to the corresponding PipedOutputStream by some other thread. Attempting to use both objects from a single thread is not recommended, as it may deadlock the thread. The piped input stream contains a buffer, decoupling read operations from write operations, within limits.
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(* OutputStream+.write(..)) && target(o) && thread(t) { }
    event read before(PipedInputStream i, Thread t) :
        call(* InputStream+.read(..)) && target(i) && thread(t) { }

    ere : create (write* | read*)

    @fail {
        System.err.println("a violation was detected");
    }
}

Figure 5.2: JAVAMOP specification PipedStream_SingleThread.

The severity of this specification is warning because a violation does not always lead to deadlock—if the buffer is large enough to hold the data to be written, write operations and subsequent read operations will not block. That said, a violation implies a potential error because the buffer size is system-dependent and can be small in some systems. The underlying system does not check the behavior; thus, it is classified as do-not-check. This specification is also classified as no-false-alarm because it detects a violation without any false positive.

** StringBuffer_SingleThreadUsage **

This specification checks if a StringBuffer object is solely used by a single thread. If this is the case, it outputs a suggestive message stating that StringBuffer can be replaced with StringBuilder for the performance benefit:

StringBuilder is designed for use as a drop-in replacement for String-
Figure 5.3: JAVAMOP specification StringBuffer_SingleThreadUsage.

Buffer in places where the string buffer was being used by a single thread (as is generally the case). Where possible, it is recommended that StringBuilder be used in preference to StringBuffer as it will be faster under most implementations.

The formal specification is shown in Figure 5.3. This specification defines two variables, which JAVAMOP instantiates for each monitor instance, on lines 2 and 3: th remembers the thread that first accessed it, and flag remembers if multiple threads have accessed it. A use event, emitted for any method invocation on a StringBuffer object, sets the flag variable if it detects multiple threads accessing an object (lines 11–17) throughout its lifetime, which begins when a constructor is invoked (i.e., an init event occurs), and ends when either the object is garbage...
collected or the entire program terminates (i.e., an endprogram event occurs).

We classified this specification as suggestion because a violation does not indicate any potential error; it merely causes performance degradation. As the underlying system does not check such behavior, it is classified as do-not-check. Since this specification can accurately monitor all uses of a StringBuffer object from any thread, there are no false positives; thus, we classified it as no-false-alarm.

**Serializable_UID**

This specification warns if a class implementing Serializable does not declare the serialVersionUID field. This specification is based on the following paragraph in the API specification:

> If a serializable class does not declare a serialVersionUID, then the serialization runtime calculates a default serialVersionUID. However, it is strongly recommended that all serializable classes explicitly declare serialVersionUID values, since the default serialVersionUID computation is highly sensitive to class details that may vary depending on compiler implementations, and can thus result in unexpected InvalidClassExceptions during deserialization.

The formal specification is shown in Figure 5.4. Unlike other specifications, where the desirable or undesirable condition can be specified solely by the pattern of method invocations or field accesses, this specification needs to retrieve more detailed information, such as the modifiers and the type of a field, in order to describe the undesirable condition precisely. Thus, we placed the precise condition check inside the staticinit event handler (lines 2–27), emitted when a static initializer of a serializable class is invoked. On lines 4–6, the enclosing class of the static initializer (i.e., the serializable class) is assigned to the klass variable. Then, the modifiers and the type of the serialVersionUID field are retrieved using reflection (lines 10–17). Three conditional statements on lines 19–21 verify that the field is static, final and of type long, as stated in the API specification. If the field does not exist, a warning message is printed on line 24.

Since the lack of this field does not cause an immediate error, we classified it as warning. Although having the field is strongly recommended, the underlying system does not check the violation; thus, this specification was classified as do-not-check. This specification was classified as no-false-alarm because it is accurate and does not cause any false alarm.

---

1. `endProgram()`, used to define the endprogram event on line 19, is the JAVAMOP's pointcut for specifying the end of an execution; i.e., when an execution terminates, an endprogram event is emitted.
2. A static initializer of a class is executed during class initialization after class loading.
Serializable_UID() {
    event staticinit after() :
        staticinitialization(Serializable+) {
            Signature initsig =
                thisJoinPoint.getStaticPart().getSignature();
            Class klass = initsig.getDeclaringType();

            if (klass != null) {
                try {
                    Field field =
                        klass.getDeclaredField("serialVersionUID");
                    int mod = field.getModifiers();
                    Class fieldtype = field.getType();

                    boolean isstatic = Modifier.isStatic(mod);
                    boolean isfinal = Modifier.isFinal(mod);
                    boolean islong = fieldtype.getName() == "long";

                    if (!isstatic) System.err.println("non-static");
                    if (!isfinal) System.err.println("non-final");
                    if (!islong) System.err.println("wrong type");
                }
                catch (NoSuchFieldException e) {
                    System.err.println("undeclared");
                }
            }
        }
    }
}

Figure 5.4: JAVAMOP specification Serializable_UID.

ShutdownHook_PrematureStart

ShutdownHook_PrematureStart warns if a shutdown hook is either running at the time of registration or the user starts it after registration. According to the API specification on Runtime.addShutdownHook(), a shutdown hook and the requirement are defined as follows:

A shutdown hook is simply an initialized but unstarted thread. When the virtual machine begins its shutdown sequence it will start all registered shutdown hooks.

This implies that it is illegal to register a started thread or to manually start a thread that is registered as a shutdown hook, because either operation makes the thread no longer qualified as a shutdown hook.
Figure 5.5: JAVAMOP specification ShutdownHook_PrematureStart.

Figure 5.5 shows the written formal specification. First, this catches an already started thread being registered by observing a bad_register event (lines 6–8). Second, it catches a shutdown hook being manually started by checking if the usage pattern matches the ERE on line 16: the user can start a thread (userstart) that was successfully registered (good_register), only if the thread has been unregistered (unregister). Here, a userstart event (lines 13–14) occurs when a thread is started by the user’s code explicitly, not by the JVM during the shutdown sequence. Any violation of this pattern, such as either bad_register or good_register followed by userstart, will result in a warning on line 19.

The severity of this specification is error because a violation indicates that the user-defined cleanup operation has prematurely started performing. The underlying system does not always detect the error: although it warns if an already started thread is registered, it does not warn when the user starts the registered thread. Thus, we classified it as sometimes-check. This specification is classified as no-false-alarm because it is accurate.
Console_FillZeroPassword(Object pwd) {
    event read after() returning(Object pwd) :
        call(char[] Console+.readPassword(..)) {}

    event zero before(Object pwd) :
        call(* Arrays.fill(char[], char)) && args(pwd, ..) {}

    event endprogram before() : endProgram() {}
    ltl : [](read => o zero)
    @violation {
        System.err.println("a violation was detected.");
    }
}

Figure 5.6: JAVAMOP specification Console_FillZeroPassword.

Console_FillZeroPassword

Console_FillZeroPassword warns if a password retrieved by Console.readPassword(), is not zeroed by invoking Arrays.fill(). This specification is based on the API specification on Console:

    Security note: If an application needs to read a password or other secure data, it should use readPassword() or readPassword(String, Object...) and manually zero the returned character array after processing to minimize the lifetime of sensitive data in memory.

Unlike other specifications shown in this section, this specification is not precise because of two reasons: there are arbitrarily many ways to manually zero the array and, consequently, it is hard to detect zeroing comprehensively; and it is impossible to define the appropriate lifetime of the password. We compromised these problems by writing an approximate specification that may cause false alarms and miss violations. First, we assume that zeroing is always performed using Arrays.fill(), because this is one of the easiest ways. Thus, this specification would yield a false alarm if one zeroes the array using other means. Second, we considered zeroing at anytime until the program ends as minimizing the lifetime of the password; i.e., only if the program never zeroes during execution, it is considered to fail to minimize it.

Based on these approximations, we wrote the formal specification as shown in Figure 5.6. We formalized the desirable pattern in an LTL formula (line 10): □(read → o zero), which can be interpreted as “it is always the case that the next event of read should be zero”. If the program never zeroes, the next event would be
endprogram, which causes a violation of the desired property.

We classified it as warning because a violation does not indicate an error. Since the underlying system does not test whether the password is zeroed, we classified it as do-not-check.

5.3 Evaluation

In this section, we evaluate the usefulness of the written parametric specifications. We first give evidence that those specifications are likely to be correct. We then evaluate the usefulness of the specifications by showing the result of monitoring all the 14 benchmarks of DaCapo 9.12 against all of those specifications.

It took about five person-months to cover four packages: java.io, java.lang, java.net and java.util. Table 5.1 shows statistics on the number of words (Section 5.2.1) and the number of formalized parametric specifications (Section 5.2.2). We have completely categorized all the documentation comments of those packages, and formalized all the runtime-monitorable specifications that they imply.

5.3.1 Correctness of Specifications

An incorrect specification can yield false positives and/or false negatives. Although one can ensure that a formal specification is not likely to yield false positives by monitoring mature programs that are unlikely buggy against it, it is more difficult to ensure that a specification does not yield false negatives because it can be hard to find a faulty program that violates the specification, especially when it involves rarely occurring events.

To reduce possible false negatives, all the specifications were reviewed by at least two people who are knowledgeable about Java and JAVAMOP. In addition to peer review, we also wrote small defective Java programs, for each of 81 non-trivial specifications, and tested if the formal specification can reveal defects. We have written 106 programs in total, and all of tests revealed the inserted defects, which gives ev-
“e”, “w” and “s” respectively represent error, warning and suggestion categories, explained in Section 5.2.3.

Table 5.2: The number of specifications, violated specifications and violations.

<table>
<thead>
<tr>
<th>Package</th>
<th>java.io</th>
<th>java.lang</th>
<th>java.net</th>
<th>java.util</th>
</tr>
</thead>
<tbody>
<tr>
<td>Severity</td>
<td>e w s</td>
<td>e w s</td>
<td>e w s</td>
<td>e w s</td>
</tr>
<tr>
<td># Specs</td>
<td>19 6 5</td>
<td>23 11 14</td>
<td>31 12 1</td>
<td>43 11 3</td>
</tr>
<tr>
<td># Viol. specs</td>
<td>3 1 3</td>
<td>2 0 11</td>
<td>1 2 0</td>
<td>6 4 1</td>
</tr>
<tr>
<td># Violations</td>
<td>19 14 12</td>
<td>36 0 4,724</td>
<td>3 2 0</td>
<td>14 134 60</td>
</tr>
</tbody>
</table>

idence that these specifications are capable of detecting errors.

5.3.2 Bug Finding

To show that the parametric specifications are useful to find bugs and bad practices, we collected all violations from DaCapo 9.12. Table 5.2 summarizes the number of specifications, the number of violated specifications, and the number of violations for each severity level.\(^3\) When counting the number of violations for each program, we counted all violations caused by the same call site as one. The results show that the specifications are capable of revealing many violations, even from programs mature enough to be included in a benchmark suite.

Since there were too many violations, we could not inspect all of them and confirm that they are true positives—some specifications may cause false positives, as discussed in Section 5.2.3. We chose instead to inspect all the causes of violations of error specifications and confirm at least one true positive for each violated warning or suggestion specification. Here, we explain only violations of error specifications and a few others. More information on others can be found at http://fsl.cs.uiuc.edu/annotated-java/.

Reader_ManipulateAfterClose, which warns if a read operation is performed after a Reader object has been closed, was violated by 13 out of 14 benchmarks of DaCapo 9.12. In fact, read() failed and the reader was immediately closed, but read() was invoked again on that closed reader. The latter read() call is reached because there is a method that discards the exception raised at the first failure and returns as if there are no errors. Since the Reader implementation raises an IOException exception anyway and each benchmark properly handles the exception, this violation does not result in a notable failure. Nevertheless, we believe that it is a bad practice to rely on an exception even when a violation is predictable.

ShutdownHook_LateRegister, which warns if one registers or unregisters a shut-
down hook\(^4\) after the JVM’s shutdown sequence has begun, was violated by \texttt{h2}—this program attempted to unregister a shutdown hook. One may think that such attempt would be safe as long as the resulting exception is properly handled, but it is indeed unsafe because registered hooks are started in unspecified order and, consequently, the hook to be unregistered may have been already started.

Collections\_SynchronizedCollection, which warns if a synchronized collection is accessed in an unsynchronized manner, was violated by \texttt{JYTHON}. This program created a synchronized collection, using \texttt{Collections.synchronizedList()}, but iterated over the collection without synchronizing on it, which may result in nondeterministic behavior, according to the API specification.

Besides these violations that imply notable problems, the written specifications could also reveal many minor yet informative violations that static analysis might not be able to detect. One such example is a violation of Math\_ContendedRandom, which recommends one to create a separate pseudorandom-number generator per thread for better performance if multiple threads invoke \texttt{Math.random()}. Another example is StringBuffer\_SingleThreadUsage (Section \ref{sec:buffer}). To detect such violations without false positives, it is necessary to accurately count how many threads access an object or a method, which is impossible for static checkers in full generality.

\footnote{According to the API specification, a \textit{shutdown hook} is an initialized but unstarted thread.}
Chapter 6

Monitoring Parametric Specifications

A large number of parametric specifications, either from an automated approach (Chapter 4) or from a manual approach (Chapter 5), pose an unprecedented challenge in runtime monitoring systems, such as JAVAMOP. Prior to this work, monitoring systems had only several parametric specifications for gaining confidence in correctness and measuring performance. This chapter presents a new runtime monitoring system, and a few techniques for monitoring a large number of specifications simultaneously.

6.1 A New Monitoring System

This section explains potential limitations of monolithic design, adopted by most existing runtime monitoring systems. As an alternative design, it then presents RV-MONITOR, the core module that implements only indispensable features of a runtime monitoring, and JAVAMOP 4.0, an integrated runtime monitoring system built on RV-MONITOR.

6.1.1 Limitations of Monolithic Design

As explained in Section 3.2, there are a number of runtime monitoring systems. A natural question then is: “why yet another system?”

Most existing systems enforce a predefined means of specifying conditions for firing events; e.g., both JAVAMOP [55] and TRACEMATCHES [9] employ AspectJ. It seems that the only exceptional case is MOPBox [56], a library that implements the module for handling event (similar to RV-MONITOR). Being a pure Java library, MOPBox does not have limitations explained in this section, but it has its own disadvantage: one should construct a FSM, at runtime, by setting alphabets, states, and transitions using its API, which can be harder than writing a specification; and the constructed monitors and MOPBox itself are not efficient, as Section 6.2.5 will show. Compared to MOPBox, RV-MONITOR provides a convenient means of stating specifications, and generates optimized code.

AspectJ and other existing instrumentation tools are sufficiently expressive in
most cases, but there are certain cases that cannot be expressed due to their limitations as explained below. AspectJ is mainly discussed, but any instrumentation tool shares some or all of the limitations.

First, an instrumentation tool may not enable one to specify the exact condition for firing events. For example, consider a specification that states “a StringBuilder object should not be used by multiple threads.” It is necessary to make a distinction between a legal invocation of `append()` and an illegal one (i.e., invocation by another thread), but, for example, a pointcut in AspectJ, even with `if` conditionals, cannot.

Second, an instrumentation tool may not provide means of picking out certain events. For example, consider a specification that checks whether a certain action is performed only after a lock has been obtained. To write this specification, one should define an event fired when a lock is obtained, but, for example, AspectJ does not provide a join point for a synchronized block. Thus, the event definition is, at best, incomplete.

The above limitations may be resolved by introducing AspectJ extensions. However, there are certain cases where arbitrary code should fire events. For example, suppose that the lock in the above specification is implemented using Dekker’s algorithm [28] rather than Java’s standard way. It is impossible to specify the place where the outer `while` loop terminates, which indicates entering a critical section in this algorithm. Also, one may want to fire an event on a certain line. These cases are unlikely to be supported by AOP tools because there is no elegant pattern-based way to match them.

For these reasons, we believe that there is no silver bullet language for specifying conditions of firing events and, therefore, a runtime monitoring system cannot be completely universal if it is tied with one language. As a solution, we designed our new system in such a way that firing events is achieved through an interface between the core monitoring module and the event-firing module, which can be implemented using any instrumentation tool, including but not limited to ones mentioned in Section 2.2.3.

In addition to expressiveness, an instrumentation tool may have limitations or bugs, which can restrict the uses of systems that are built on it. For example, JAVA-MOP 2.3 was not able to monitor one application in DaCapo 9.12 against the 179 specifications due to AspectJ’s limitation (Section 6.2.4). We were able to mitigate that limitation by modifying AspectJ because the source code was fortunately available and well maintained. However, if this was not the case, JAVA-MOP 2.3 and other monitoring systems that depend on AspectJ would have been useless for such applications, unless a major change in those monitoring systems was made.

It is also widely believed that modular design has advantages, such as improved maintainability, and this is the case in a monitoring system. A clear separation of
two concerns—firing an event in a certain condition, and handling the event—is achievable by having a simple but universal interface, as explained in Section 6.1.2. Thanks to this separation, for example, if the performance of event handling needs to be improved, one needs to look into only RV-Monitor, which is simpler than a monolithic system; and, similarly, if one devises a way to suppress insignificant events, only the module for firing events needs to be modified.

6.1.2 RV-Monitor: A Runtime Verification Library Generator

As explained in Section 6.1.1, we believe that no single language is expressive enough to specify events. We therefore claim that two different concerns—firing events and monitoring the program based on the observed events—should not be implemented together in a monolithic system. With this in mind, we developed a stand-alone application, called RV-Monitor, that generates a Java library, according to the given specifications, that implements all the core monitoring functionality. With the other module for firing events, an integrated monitoring system can be built, as explained in Section 6.1.3.

RV-Monitor takes one or multiple RV-Monitor specifications as input. One example is shown in Figure 2.1. As explained in Section 2.1, an RV-Monitor specification does not specify when an event is fired, because firing events is not its concern; it simply declares events with their parameters.

For the given specifications, RV-Monitor generates a plain Java library that contains methods, each of which corresponds to an event definition. These methods can be thought of as the interface between the generated monitoring functionality and the module for firing events. That is, invoking one such method is firing an event. This approach enables one to build customized runtime monitoring systems on the top of generated library, as long as the environment permits invocations of Java methods—any language that runs under the JVM does.

From the specification shown in Figure 2.1, for example, RV-Monitor generates the Collection_UnsafeIteratorRuntimeMonitor class, named after the given specification, and a few other supporting data structures. Within this class, three static methods are defined, one for each event definition, as shown in Figure 6.1.

To fire a createIterator event, one can simply call the createIterator() method, together with two arguments; then, the method performs all the required operations for monitoring (Sections 2.3 and 2.4), such as trace slicing, creating or updating monitor instances, and invoking the @match handler (on lines 8–11 in Figure 2.1) if a trace slice matches the property (on line 6).
public class Collection_UnsafeIteratorRuntimeMonitor {
    public static void createIterator(Collection c, Iterator i) {
        // auto-generated event handling routine
    }
    public static void modifyCollection(Collection c) {
        // auto-generated event handling routine
    }
    public static void useIterator(Iterator i) {
        // auto-generated event handling routine
    }
}

Figure 6.1: Collection_UnsafeIteratorRuntimeMonitor, a class generated from the Collection_UnsafeIterator specification, shown in Figure 2.1, by RV-Monitor.

6.1.3 JAVAMOP: An Integrated Runtime Monitoring System

The Java class generated by RV-MONITOR is sufficient for monitoring per se, as long as one inserts method invocations for firing events into a program to be monitored. For the purpose of monitoring an existing program against specifications, however, we believe it is more convenient to use a weaving tool, such as AspectJ.

Since the generated library performs all the required operations for monitoring, one can build a runtime monitoring system by simply adding a module for notifying the library of events. As an example, this thesis presents JAVAMOP 4.0\(^1\) that preserves backward compatibility for JAVAMOP 2.3 specifications, so that one can use all the existing JAVAMOP specifications\(^2\). Built on RV-MONITOR, JAVAMOP 4.0 needs only a simple front-end that takes one or multiple JAVAMOP specifications as input, and generates an AspectJ aspect and one or multiple RV-MONITOR specifications, as shown in Figure 6.2.

For example, consider the specification shown in Figure 2.2. JAVAMOP 4.0 first extracts event definitions (without pointcuts), properties and handlers; generates the corresponding RV-MONITOR specification, shown in Figure 2.1; and runs RV-MONITOR in order that a Java library is generated. It then extracts pointcuts from the event definitions, and generates an aspect, which depends on methods in the generated library, as shown in Figure 6.3.

A pointcut for specifying the condition for firing a createIterator event is written on lines 2–3, and the corresponding advice is written on lines 4–6. Since all the monitoring-related routines are implemented by the generated library, an advice

---

\(^1\)3.x has been already taken by an intermediate version that has never been published.

\(^2\)Only the class name of the built-in logging functionality has been changed.
simply fires an event with captured parameters by calling the corresponding method in the generated library. By weaving this aspect and a program, one can monitor that program against the Collection_UnsafeIterator specification.

Although this thesis presents only one way of firing events, one can freely choose any technique, depending on the purpose. For example, consider that one wants to define an event fired when a thread is about to wait, which is not supported by any instrumentation tools (including AOP tools), to the best of our knowledge. Unlike existing monitoring systems, which are tied with a certain instrumentation method and, therefore, capturing such event is hard without modifying the system, the new design presented in this thesis enables one to build a specialized system with minimum effort: writing a JVMTI agent that forwards events notified by the JVM, to the generated library. Without the separation of concerns, enabling such events would cause complicated ramification.

Also, if none of existing tools are suitable, one can either devise a new one or manually insert invocations of the generated methods into the program. This way, one can fire events in arbitrary places, including a loop termination, which is necessary to detect an acquisition of a lock based on Dekker’s algorithm.

The separation also gives two independent stages for optimization, as mentioned in Section 6.1.1. In RV-MONITOR, one can focus on reducing the overhead of handling events and maintaining monitor instances without the worry of instru-

---

3 A JVMTI agent can listen to synchronization-related events as well as method invocations.
public aspect Collection_UnsafeIteratorAspect {
  
  pointcut createIterator(Collection c) :
      call(Iterator Iterable+.iterator()) && target(c);
  after(Collection c) returning(Iterator i) : createIterator(c) {
    Collection_UnsafeIteratorRuntimeMonitor.createIterator(c, i);
  }

  
  pointcut modifyCollection(Collection c) : /* omitted */ ;
  before(Collection c) : modifyCollection(c) {
    Collection_UnsafeIteratorRuntimeMonitor.modifyCollection(c);
  }

  
  pointcut useIterator(Iterator i) : /* omitted */ ;
  before(Iterator i) : useIterator(i) {
    Collection_UnsafeIteratorRuntimeMonitor.useIterator(i);
  }
}

Figure 6.3: Collection_UnsafeIteratorAspect, an aspect generated from the Collection_UnsafeIterator specification, shown in Figure 2.2 by JAVAMOP 4.0.

mentation, which is also part of this thesis (Section 6.2). In contrast, in the other part of JAVAMOP, one can focus on suppressing unnecessary events; e.g., considering the specification on StringBuilder, mentioned in Section 6.1.1. If static analysis ensures that a StringBuilder object is locally used, this object would cause no violations; therefore, one can skip firing events, and this can be simply achieved by not calling the method in the generated library.

6.2 Monitoring Multiple Specifications Simultaneously

This section discusses a few optimization techniques for efficient monitoring; more specifically, it discusses how to handle events efficiently. Since all the core monitoring functionality is implemented in RV-MONITOR, according to the new design (Section 6.1), the main focus of this section lies in RV-MONITOR, except Section 6.2.4, which addresses an instrumentation problem.

6.2.1 Overhead Analysis

To analyze overheads in the presence of multiple specifications, an experiment using JAVAMOP 2.3 was conducted. At the time of writing this thesis, JAVAMOP 2.3 is the most efficient system, according to Jin et al. [44], that does not have any known major problem. For the experiment, the LUSEARCH and PMD benchmarks of Da-
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Memory (KB)</th>
<th>GC time (sec)</th>
<th>Total time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUSEARCH†</td>
<td>13,191</td>
<td>0.1</td>
<td>22.5</td>
</tr>
<tr>
<td>LUSEARCH‡</td>
<td>104,749</td>
<td>2.1</td>
<td>108.1</td>
</tr>
<tr>
<td>PMD†</td>
<td>30,720</td>
<td>0.2</td>
<td>21.6</td>
</tr>
<tr>
<td>PMD‡</td>
<td>926,729</td>
<td>9.5</td>
<td>162.0</td>
</tr>
</tbody>
</table>

Table 6.1: Peak memory, and GC time and total execution time with (†) and without (‡) the 179 specifications.

Capo 9.12 [12] were executed with and without all the 179 specifications from Chapter 5, because they showed large overheads in an exploratory testing. To measure overhead, VisualVM [69] was attached to the JVM.

JAVA-MOP 3.0, which incorporated all the optimizations in Jin [43], is available, but it has a fault that is non-trivial to fix. It uses $\mathcal{D}(X)$, a monitoring algorithm introduced in Chen et al. [22], which should separately keep a timestamp for each monitor instance. As part of intensive optimizations, however, JAVA-MOP 3.0 moves the timestamp information to the weak reference, which can be shared among multiple monitor instances. Consequently, timestamps of different monitor instances can wrongly affect each other, causing some monitor instances to ignore events. For example, suppose that JAVA-MOP 3.0 is handling an event that carries $\langle P \mapsto p_1, Q \mapsto q_1 \rangle$ and creating the weak reference for $q_1$. Then, the timestamp, which records the time this event is handled, is stored at the weak reference for $q_1$. When an event that carries $\langle P \mapsto p_2, Q \mapsto q_1 \rangle$ occurs later, the timestamp stored at the time of handling the previous event will be retrieved, and this causes JAVA-MOP 3.0 to undesirably skip this event. In contrast, in the correct implementation, the stored timestamp would not be used because the previous event and the later one do not share the same monitor instance. This fault was validated by a concrete example and confirmed by the author of Jin [43].

It was meaningless to analyze the overhead of JAVA-MOP 3.0 and compare the performance of it with the new one, because it is faulty. Also, it seemed that writing a fixed JAVA-MOP 3.0 unfortunately requires significant amount of work because it was caused not by a trivial mistake but by a wrong assumption, and it was revealed only after significant modifications for the separation of concerns (Section 6.1) and performance improvement have been started. For these reasons, overhead analysis and performance comparison were made using JAVA-MOP 2.3.

The results, summarized in Table 6.1, show that monitoring imposes significant

---

\[4\] JAVA-MOP 3.0 keeps the timestamp in one of parameters. Here we assume that JAVA-MOP 3.0 kept timestamps in weak references of $Q$.  

72
overhead on memory and, consequently, increases both minor and major garbage collection time. Jin [43] presents a few techniques for reducing the memory overhead of JAVAMOP: avoiding creating multiple weak references for the same object, and combining indexing trees (Section 3.2) that share the same prefix.

Table 6.1 also shows that threads under monitored executions spent significant time in the “blocked” state; in particular, monitoring hindered LUSEARCH and PMD’s concurrent execution. This is mainly because JAVAMOP 2.3 uses one global lock in a coarse manner; if multiple events happen to occur in different threads at the same time, all the other threads should wait until the first arriving thread finishes handling the event. This thesis proposes fine-grained locking to reduce such hindrance, as explained in Section 6.2.2.

In addition to the “blocked” state, the total time in the “runnable” state increased 5–7 times—this overhead includes the cost of the monitoring procedure, explained in Sections 2.3 and 2.4. Jin [43] suggests that invoking System.identityHashCode() is surprisingly expensive and the return value should be cached, instead of invoking it frequently; more specifically, whenever JAVAMOP 2.3 retrieves monitor instance(s) for a parameter binding.

The statistics on hot spots showed that an event that updates a set of monitor instances is expensive. To investigate the cause, we ran AVRORA in DaCapo 9.12, which also showed large overhead, against the CollectionUnsafeIterator specification, shown in Figure 2.2. The resulting statistics showed that 4% of entire CPU time is spent on handling modifyCollection events, and this is the second most time-consuming spot in the execution, preceded by an internally used method in RV-MONITOR. This result is surprising, considering that the modifyCollection event occurred fewer than the other two events by an order of magnitude; the number of occurrences of createIterator, modifyCollection and useIterator were 6.3M, 0.7M and 10M, respectively.

The main reason for the large overhead of handling modifyCollection events is that there were numerous monitor instances that transition upon an occurrence of that event. This can happen when there is a long-lived Collection object that has created many Iterator objects—recall that, according to Definition 3, a modifyCollection(Collection→c0) event should be dispatched to all the trace slices that correspond to ⟨Collection→c0, Iterator→i1⟩, ⟨Collection→c0, Iterator→i2⟩, ..., ⟨Collection→c0, Iterator→in⟩, where i1, ..., in are the Iterator objects that c0 has created, using the iterator() method.

5 Information on minor and major collections can be found in Java SE 6 HotSpot Virtual Machine Garbage Collection Tuning [40].

6 This method returns the hash code based on the object’s identity, not class-specific hash function.

7 In an extreme case, there were about 300,000 monitor instances for a single Collection object. This number may differ according to the heap size or the threshold for triggering the garbage collector.
In this case, terminating monitor instances that would never violate the property, proposed in Jin et al. [44], is not useful because usesIterator(Iterator↦𝑖), where 1 ≤ 𝑗 ≤ 𝑛, causes any of the above trace slices to reach the violation state. The number of monitor instances for that Collection object, 𝑛, continues to grow until memory pressure reaches a certain threshold and, consequently, the JVM triggers a garbage collection.

When a modifyCollection(Collection↦𝑐0) event occurs, JAVA-MOP first finds the corresponding set of monitors by looking up the middle tree of Figure 3.1 or the left tree of Figure 3.2. Then, it sequentially sends this event to each monitor instance in the set. Although this behavior is correct and looks normal, it turned out that only a small number of monitor instances are actually affected by such an event; all other monitor instances stay at the same state due to the self-loop. This thesis addresses such overhead by introducing a new implementation for a set of monitor instances, as explained in Section 6.2.3.

6.2.2 Fine-Grained Locks

As explained in Section 6.2.1, the current version of JAVA-MOP uses one global lock throughout all the operations for handling an event, which may involve multiple global weak reference table (GWRT) accesses and indexing tree lookups. This can significantly hinder concurrent execution in the presence of multiple specifications because it is likely that more events occur simultaneously.

To reduce this hindrance, RV-MONITOR removes the global lock, and instead uses multiple fine-grained locks, considering that each of GWRTs and indexing trees is independently accessed. First, each GWRT is separately synchronized because GWRTs do not interfere with each other. This enables multiple threads to run concurrently, unless they handle the same type of parameters—one GWRT is created for each parameter type. Second, each level of an indexing tree is separately synchronized. For example, consider a createtIterator event in Figure 2.1, which brings two parameters: 𝑐 and 𝑖. When looking up the left tree in Figure 3.2 to retrieve the monitor instance corresponding to the carried parameter values, RV-MONITOR first acquires a lock corresponding to the first level. On retrieving the node at the second level according to the object bound to 𝑐, it immediately releases the lock. This way, another request on the first level of this tree can be served with relatively short delay.

To promote concurrent execution further, RV-MONITOR moved to thread-local

---

8 In some extreme cases, only about 3,000 monitor instances, out of 300,000, were actually affected.
9 A GWRT is a data structure that keeps the mapping from strong references to weak references, for each parameter type. This data structure was introduced in JAVA-MOP 3.0, in order to reduce memory overhead by creating at most one weak reference for each strong reference.
storage (TLS) two caches: the cache in each indexing tree and the cache in each GWRT. This is based on the observation that most objects bound to parameters are solely used in a single thread. With this change, if a request is served by the cache, no synchronization is performed, at the cost of adding a few cache entries to each GWRT and each level of indexing trees, per thread.

RV-MONITOR drops JAVA-MOP 3.0’s another multi-entry cache, indexed by code locations of programs under monitoring, for each GWRT. The rationale for having this cache is explained in Jin [43], but it did not present the amount of performance improvement. A preliminary experiment with JAVA-MOP 3.0 showed that the benefit is negligible; none of benchmarks in DaCapo 9.12 showed consistently less overhead with this additional cache. Also, RV-MONITOR has no means of retrieving code locations, unlike JAVA-MOP 3.0, which could rely on AspectJ for obtaining a unique identifier for the code location where instrumentation is performed. It is possible to add a parameter for such identifier to each method that RV-MONITOR generates, and enforce the module for firing events to provide it, but it is doubtful if it is worth implementing this cache.

In addition to fine-grained locking, the implementation of a monitor instance has been also modified in such a way that each access is thread-safe. This modification is necessary because, unlike JAVA-MOP 2.3 where a coarsely used global lock guarantees atomicity, RV-MONITOR allows multiple threads to handle events simultaneously. To make it thread-safe, we made each transition in a monitor instance atomic using the compare-and-swap operation.

6.2.3 Optimization for Kleene Star

As explained in Section 6.2.1, the current version of JAVA-MOP sends an event to every monitor instance in the set that corresponds to the event—recall that the set of monitor instances is retrieved if an event does not carry all the parameter values—even when most monitor instances do not need to receive the event.

To avoid sending an event to unaffected monitor instances in a set, RV-MONITOR introduces a new implementation for a set of monitor instances. At the heart of this technique is partitioning a set according to the state of each monitor instance. When an event occurs, for each partition, the new implementation first checks whether this partition is affected. This check can be easily implemented by sending the event to one element in the partition and checking whether the state has changed, because the set is partitioned according to the state and all the elements has the exactly same transition table. If that partition is affected, the implementation sends the event to each monitor instance in the partition. If the partition is unaffected, the entire partition can be ignored.
One assumption that this technique makes is that an event does not have any side-effect; i.e., the body of each event definition is empty, like all the three event definitions in Figure 2.1.

Maintaining partitions according to states is unfortunately non-trivial because a monitor instance usually belongs to multiple sets. For example, consider the Collection_UnsafeIterator specification (Figure 2.1) and the indexing trees for this specification (Figure 3.2). There are two kinds of sets for this specification. One kind is for handling a modifyCollection event, which carries only Collection object, and all sets of this kind will be used in the left tree. In contrast, the other kind, which can handle a useIterator event is used in the right tree. A monitor instance that corresponds to \(\langle \text{Collection} \mapsto c, \text{Iterator} \mapsto i_1 \rangle\) will then belong to both one set in the left tree and another set in the right tree, in order that both \(\text{modifyCollection}(\text{Collection} \mapsto c_1)\) and \(\text{useIterator}(\text{Iterator} \mapsto i_1)\) events can be efficiently handled. When a useIterator(\text{Iterator} \mapsto i_1) event occurs, the left indexing tree is not used and, as a result, the set that holds the monitor instance for \(\langle \text{Collection} \mapsto c_1, \text{Iterator} \mapsto i_1 \rangle\) in the left tree is unable to move this instance to the proper partition, according to the state change. This would result in a violation of the property that this new set implementation should keep: “a monitor instance belongs to a partition according to its state.” Similarly, a modifyCollection(\text{Collection} \mapsto c_1) event can cause a set in the right tree to violate the property.

To avoid this problem, RV-MONITOR notifies the other set of a state change, after handling an event in the corresponding set. In the above example, if a useIterator(\text{Iterator} \mapsto i_1) event occurs, RV-MONITOR first retrieves the set for \(i_1\) from the right indexing tree, and sends the event to all the state-changing monitor instances that correspond to \(\langle \text{Collection} \mapsto c_1, \text{Iterator} \mapsto i_1 \rangle, \langle \text{Collection} \mapsto c_2, \text{Iterator} \mapsto i_1 \rangle, \ldots, \langle \text{Collection} \mapsto c_n, \text{Iterator} \mapsto i_1 \rangle\), where \(c_1, \ldots, c_n\) are related to \(i_1\)—in this particular specification, \(n\) will be at most 1 due to the semantics, but there can be multiple monitor instances in general. After such normal operation, RV-MONITOR notifies the sets in the other indexing tree of a state change. If the state of the monitor instance corresponding to \(\langle \text{Collection} \mapsto c_1, \text{Iterator} \mapsto i_1 \rangle\) has changes, RV-MONITOR retrieves the set for \(c_1\) from the left tree, and notifies of a state change, which causes the set to eventually move the monitor instance corresponding to \(\langle \text{Collection} \mapsto c_1, \text{Iterator} \mapsto i_1 \rangle\) to the appropriate partition. RV-MONITOR repeats this for \(c_2, \ldots, c_n\). This operation is not expensive—the number of related objects for an Iterator object is 1 and, as a result, at most one additional indexing tree and set accesses are added for this event.

However, handling a modifyCollection event is indeed expensive, because there can be numerous related objects for a single Collection object. Although it may add huge overhead for this event, keeping a partitioned set turns out to be still beneficial.
in practice, because runs of the same events are often observed. When the same events consecutively occur, only the first one typically requires expensive handling; then, most monitor instances move to the state that has a self-loop for that event, because a property has only a few Kleene stars. Once all the monitor instances move to such state, any subsequent event is handled with minimum operations: sending an event to one monitor instance for each partition, in order to check whether the partition is affected. Given that the set is partitioned according to states, the number of partitions is bound to the number of states, which is typically small.

This new set implementation seems efficient, at least, for this particular specification. After replacing the default set implementation by this new one, the overhead of handling modifyCollection events was reduced to 0.2%. However, notifying the other set of a state change seems very complicated when the number of parameters is larger than 2. Thus, this set implementation is employed only when a specification involves exactly two parameters.\(^{10}\)

### 6.2.4 Weaving for Multiple Specifications

As mentioned in Section 6.1.1, an instrumentation failure was encountered during an execution of FOP in DaCapo 9.12 against 179 specifications; AJC, the AspectJ compiler, terminates with an error message “code size too big.” Such failures, at best, preclude the program from firing events, and the missing events can cause false positives and negatives. In particular, if load-time weaving (LTW) is enabled, such failures even terminate the entire process. The cause and fix of this problem are explained below. For the purpose of experiments, the fix will be specific to AspectJ, but the idea of avoiding the problem can be applied to other instrumentation tools.

While weaving, AspectJ inserts into a matched join point a chunk of code for invoking corresponding advices. For example, consider the aspect shown in Section 6.3. If there is a method that invokes ArrayList.iterator(),\(^{11}\) then AspectJ will insert code for invoking the advice, defined on lines 4–6, into that method, because it has a join point that matches with the condition on line 3. Although the size of the inserted code for each join point is moderate, a method with excessive number of matched join points can cause the size of the method to exceed 64KB, Java’s limit \([6, §4.9.1]\).

To continue the experiment and, more importantly, mitigate the possibility of such problems, we modified AspectJ in such a way that it extracts a method from a join point and replaces the join point by an invocation of the extracted method, which is similar to what the extract method refactoring does. Since the matched join point has been moved, the additional chunk of code is inserted into the extracted

---

\(^{10}\)When a specification involves only one parameter, a set is not needed.

\(^{11}\)The ArrayList class implements the Iterable interface, specified on line 3 in Figure 6.3.
method, instead of the originating method. At the cost of adding a method, this replacement avoids the increment in the size of the originating method because the size of instructions for invoking such an extracted method is no larger than that of the extracted join point. More detailed explanation on the modification can be found in Appendix A.

Since the modified AspectJ extracts a method from each join point, the number of methods in the enclosing class of that method would be increased. There is also a limit on the number of methods in a class—a class can have at most 65,535 methods [67, §4.11]—and weaving may fail if there are a lot of matched join points in a class. However, it is believed that the odds that happen are not much. With this modification, it was possible to monitor all the benchmarks of DaCapo 9.12 against the 179 specifications simultaneously.

6.2.5 Evaluation

This section evaluates JAVAMOP 4.0, which is built on RV-MONITOR and the modified AspectJ (Section 6.2.4), by comparing it with MOPBox [56] and JAVAMOP 2.3. For performance measurement, we ran all the 14 benchmarks of DaCapo 9.12 [12] with the default input. The experiment was conducted using Java Platform Standard Edition 6 (build 1.6.0_35) under a system that runs Windows 8 (64 bit) with a 3.1 GHz Intel Core i3 and 12GB of memory.

Comparison with JAVAMOP 2.3

To see the performance improvement, JAVAMOP 2.3, the most recent version that does not have a major problem, and JAVAMOP 4.0, the new one this thesis presents, are compared. Since both of them accept the same type of specifications, all the 179 specifications were used for this experiment.

The execution time for each benchmark and setting are summarized in Table 6.2. In both tables, the “Original” columns show the unmonitored runs of benchmarks, whereas all the other columns show the monitored runs with the 179 specifications. To obtain the execution time under a steady state, the -converge option with 5 windows was used.

The result shows that RV-MONITOR has significantly less overhead than the state-of-the-art for a few cases. In particular, the overhead was much less, thanks to fine-grained locking (Section 6.2.2), when a benchmark is multi-threaded and there

---

12 This number does not include methods that are inherited from superclasses or superinterfaces [67, §4.11]. However, this number can be restricted by the limit on the size of constant pool, which is 65,535, because each extracted method is referred to by the enclosing class and, consequently, occupies an entry in the constant pool.
Table 6.2: Execution time for JAVAMOP 2.3 and JAVAMOP 4.0 (with (†) and without (‡) the optimization for Kleene Star, explained in Section 6.2.3) in seconds.

is little interaction between threads, such as LUSEARCH and XALAN [7]. The optimization for Kleene Star (Section 6.2.3) does not always reduce overhead, because it has its own overhead for maintaining partitions. As explained in Section 6.2.3, this optimization can be effective when there are long-lived objects that create many related objects. One such case is AVRORA; in extreme cases, one event caused more than 300,000 monitor instances to follow self-loops.

It should be noted that the overhead is significant, for some cases, because monitoring some benchmarks against 179 specifications is indeed a challenging task. Most benchmarks emitted millions of events; in particular, AVRORA, H2 and PMD emitted 32,804,400, 65,647,663 and 48,866,293 events, respectively.

Comparison with MOPBox

As explained in Section 6.1.1, MOPBox [56] requires one to construct a FSM by setting alphabets, states, and transitions using its API. Also, unlike JAVAMOP or RV-MONITOR, where a property can be written in various formalisms, only an FSM is permitted and, therefore, one should convert an ERE or an LTL formula into an equivalent FSM.

Since this preparation requires significant time and effort, we tested only Collection_UnsafeIterator specification, the most heavily used specification in most benchmarks of DaCapo 9.12. Figure 6.4 shows part of code for constructing the FSM template for this specification. Events and parameters are declared on lines 3–4.
class CollectionUnsafeIteratorTemplate
extends FSMMonitorBenchmarkTemplate<Event, Param> {
public enum Event { Create, Modify, UseIter }
public enum Param { C, I }

public CollectionUnsafeIteratorTemplate() {
    this.initialize();
}

@Override
protected void fillAlphabet(IAlphabet<Event, Param> alphabet) {
    this.addEvent(alphabet, true, Event.Create, Param.C, Param.I);
    this.addEvent(alphabet, false, Event.Modify, Param.C);
    this.addEvent(alphabet, false, Event.UseIter, Param.I);
}

@Override
protected State<Event> setupStatesAndTransitions() {
    State<Event> initial = this.makeState(false);
    State<Event> iterating = this.makeState(false);
    State<Event> modified = this.makeState(false);
    State<Event> error = this.makeState(true);

    this.addTransition(initial, Event.Create, iterating);
    this.addTransition(iterating, Event.UseIter, iterating);
    this.addTransition(iterating, Event.Modify, modified);
    this.addTransition(modified, Event.UseIter, error);
    return initial;
}
}

Figure 6.4: A hand-written FSM template for the CollectionUnsafeIterator specification, shown in Figure 2.1.

In fillAlphabet() on lines 10–15, three events, together with their parameters, are defined—the purpose of this method is similar to event specifications, shown on lines 2–4 in Figure 2.1. The second argument of addEvent() represents whether this event is a creation event, similar to the creation keyword used in an RV-MONITOR specification; i.e., this event may create a new monitor instance. The FSM, which corresponds to the property originally written in an ERE, is constructed in setupStatesAndTransitions() on lines 17–30. The argument of makeState() represents whether this state is one of the final states; e.g., only the error state is final here.

MOPBox and the FSM template, defined in Figure 5.4, implements the core
functionality for monitoring, similar to RV-Monitor. To notify MOPBox of events, one needs to invoke MOPBox’s API; for the experiment, we used the modified AspectJ, explained in Section 6.2.4, to add invocations of the API. Figure 6.5 shows the aspect used during the experiment. This hand-written aspect is similar to one generated by JavaMOP 4.0 (Figure 6.3); pointcuts are exactly the same because the conditions for firing events are the same. The purpose of each advice is the same, but MOPBox requires an additional step that creates a VariableBinding object for specifying the parameter binding. The created parameter binding is then passed to the FSM template, referred to by tpl. The template, initialized on line 4, performs all the required operations for monitoring (Sections 2.3 and 2.4), such as trace slicing, manipulating monitor instances, and invoking the handler, though they are omitted in Figure 6.4.

This aspect was woven into each benchmark in DaCapo 9.12. Among a few monitoring algorithms that MOPBox implements, $\mathbb{C}^+(X)$ [22] was chosen because this is known to be the most efficient one among them. The maximum heap size was set to 8GB, and, to obtain the execution time under a steady state, we used the -converge option with 5 windows.

Table 6.3 shows the execution time and the number of garbage collections. Even though only one specification was monitored, the overhead of monitoring was prohibitively high, except three benchmarks that fire few events: Tomcat, Tradebeans, and Tradesoap. Most benchmarks barely finished the first iteration in 10 minutes. Luindex managed to run multiple iterations, but the elapsed time of each iteration continued to increase; at the last iteration before it failed to converge, it took 483.42 seconds. For most cases, garbage collections were frequently performed, which indicates that memory overhead is significant.

The result shows that the overhead of MOPBox with one specification is by far more than that of RV-Monitor with 179 specifications. One of fundamental reasons is that performance was an important issue in RV-Monitor, whereas it seems that MOPBox sacrifices performance for consistent interfaces, unaffected by the given specifications. One can clearly see the difference between the interface of RV-Monitor and that of MOPBox from Figures 6.3 and 6.5. Since RV-Monitor generates the interface, each method for firing an event is specialized; e.g., the interface takes two arguments for the createIterator event, whereas it takes one for the other events. In contrast, MOPBox has one universal method for firing any event, processEvent(), regardless of the number of parameters. This universal interface enables MOPBox to be easily integrated into an integrated development environment (IDE) for stateful breakpoints [15], but it imposes the overhead of creating a VariableBinding object whenever an event occurs.

Also, MOPBox uses $\mathbb{C}^+(X)$ [22], which is less efficient than $\mathbb{D}(X)$ [22], used
public aspect Collection_UnsafeIteratorAspect {
  private final static CollectionUnsafeIteratorTemplate tpl;
  static {
    tpl = new CollectionUnsafeIteratorTemplate();
  }

  pointcut createIterator(Collection c) :
    call(Iterator Iterable+.iterator()) && target(c);
  after(Collection c) returning(Iterator i) : createIterator(c) {
    VariableBinding<Param, Object> binding =
      new VariableBinding<Param, Object>();
    binding.put(Param.C, c);
    binding.put(Param.I, i);
    tpl.processEvent(Event.Create, binding);
  }

  pointcut modifyCollection(Collection c) : /* omitted */ ;
  before(Collection c) : modifyCollection(c) {
    VariableBinding<Param, Object> binding =
      new VariableBinding<Param, Object>();
    binding.put(Param.C, c);
    tpl.processEvent(Event.Modify, binding);
  }

  pointcut useIterator(Iterator i) : /* omitted */ ;
  before(Iterator i) : useIterator(i) {
    VariableBinding<Param, Object> binding =
      new VariableBinding<Param, Object>();
    binding.put(Param.I, i);
    tpl.processEvent(Event.UseIter, binding);
  }
}

Figure 6.5: A hand-written aspect for the Collection_UnsafeIterator specification, shown in Figure 2.2.

by RV-MONITOR. It is believed that MOPBox uses the less efficient one probably because it requires much effort to implement \( \overline{D}(X) \). In addition to the monitoring algorithm, MOPBox does not have an efficient data structure for monitor instances, such as indexing trees used in RV-MONITOR.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Execution time (s)</th>
<th># GCs</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVRORA</td>
<td>&gt; 600</td>
<td>142</td>
<td>timed out after 0 iterations</td>
</tr>
<tr>
<td>BATIK</td>
<td>&gt; 600</td>
<td>195</td>
<td>timed out after 0 iterations</td>
</tr>
<tr>
<td>ECLIPSE</td>
<td>&gt; 600</td>
<td>216</td>
<td>timed out after 0 iterations</td>
</tr>
<tr>
<td>FOP</td>
<td>&gt; 600</td>
<td>193</td>
<td>timed out after 1 iteration</td>
</tr>
<tr>
<td>H2</td>
<td>&gt; 600</td>
<td>176</td>
<td>timed out after 0 iterations</td>
</tr>
<tr>
<td>JYTHON</td>
<td>&gt; 600</td>
<td>209</td>
<td>timed out after 0 iterations</td>
</tr>
<tr>
<td>LUIINDEX</td>
<td>383.76</td>
<td>199</td>
<td>failed to converge</td>
</tr>
<tr>
<td>LUSEARCH</td>
<td>&gt; 600</td>
<td>111</td>
<td>timed out after 1 iteration</td>
</tr>
<tr>
<td>PMD</td>
<td>&gt; 600</td>
<td>198</td>
<td>timed out after 1 iteration</td>
</tr>
<tr>
<td>SUNFLOW</td>
<td>&gt; 600</td>
<td>73</td>
<td>timed out after 1 iteration</td>
</tr>
<tr>
<td>TOMCAT</td>
<td>1.37</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>TRADEBEANS</td>
<td>6.84</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>TRADESOAP</td>
<td>3.70</td>
<td>87</td>
<td></td>
</tr>
<tr>
<td>XALAN</td>
<td>&gt; 600</td>
<td>88</td>
<td>timed out after 1 iteration</td>
</tr>
</tbody>
</table>

Table 6.3: Execution time and number of GCs for MOPBox.

6.3 Preparation-Free Monitoring

Previous sections show that runtime monitoring is capable of monitoring many parametric specifications simultaneously (Section 6.2.5) and reporting many violations (Section 5.3.2). Despite its usefulness, runtime monitoring systems typically require tedious and non-trivial preparation steps, and it might be one of valid reasons for the reluctance to adopt them. This section presents a system that eliminates such preparation step.

6.3.1 Difficulties in Preparation for Monitoring

There are a few approaches and several tools for enabling a runtime monitoring system to observe events during execution, as explained in Section 2.2.3. One way is to write a JVMTI [41] agent that listens to JVM’s events—such as entering a method and returning from a method—and then notifies the monitoring system of an event, such as invoking the method generated by RV-MONITOR. Using a JVMTI agent has major benefits. First, it does not require any preparation in the program under monitoring; i.e., one can use the program as it is. One can simply enable the agent by providing the path to the agent on the command line, as explained in Section 2.2.3. Second, it is easy to capture every single event, no matter how a method in the program under monitoring is loaded, or where a method is invoked. For example, one can capture a method invocation from any call site in a dynamically loaded class or even in the runtime (rt.jar), which is hard to achieve using instrumentation. Third, one can capture not only method invocation or field access but also other
moments, such as garbage collection (JVMTI_EVENT_GARBAGE_COLLECTION_START) and monitor (in the context of locking) wait (JVMTI_EVENT_MONITOR_WAITED).

However, this approach has a few crucial disadvantages, which makes it unideal for firing events in monitoring systems. First, there is no way to selectively listen to method invocation events; i.e., an invocation of any uninteresting method will invoke the JVMTI agent. This significantly degrades overall performance of the execution, and it is even advised not to enable this event [11]. Second, this approach is platform-dependent because some JVM may not implement JVMTI. Third, a JVMTI agent is not portable and should be recompiled, because JVMTI provides a native interface and, consequently, an agent should be written in C/C++ or other low-level languages that can call native functions.

Instrumenting a program is another approach to notify a monitoring system of events. This approach can be more efficient because one can listen to invocations of a certain set of methods. Additionally, being injected into the program, the routine for firing events is treated as ordinary Java code—involving the method for firing an event is implemented as an invokevirtual instruction, a Java instruction for invoking a static method, optionally preceded by instructions for pushing arguments onto the stack. As a result, there is no overhead for context switch, which is necessary for a JVMTI agent to fire an event. Also, such code can be further optimized by the just-in-time (JIT) compiler during execution.

Although instrumentation is desirable for performance reasons and, in fact, used by most monitoring systems, compile-time instrumentation has two major drawbacks. First, it may require some non-trivial change in the build procedure. One needs to insert a new phase for instrumentation into the existing build script, which requires knowledge on the build system. In particular, if the final artifact is an executable JAR, one additionally needs to manipulate its manifest file in such a way that the Class-Path field in this file refers to the paths to the runtime libraries of both the instrumentation tool and the monitoring system. Furthermore, this instrumentation procedure should be repeated whenever the program to be monitored or specifications are modified.

Second, it can be difficult to thoroughly instrument a program and its dependencies. Dependencies should be supplied to the instrumentation tool because the hierarchy of types is needed for matching: picking out places where an event should be fired. For example, consider a specification on the use of the Iterator interface, a library that defines a class that implements this interface, and a program that is built on this library. To decide whether an invocation on that class should fire an event, the instrumentation tool needs the library because the program itself does not state

---

13 The environment variable CLASSPATH and any class path specified on the command line is ignored by the JVM if the -jar option is used [38].
whether the class implements Iterator. Also, supplying dependencies may require a non-trivial task; e.g., if a program is run by a script, one needs to analyze the script to see what libraries are possibly loaded at runtime.

### 6.3.2 Architecture

As explained in Section 6.3.1, instrumentation is desirable for performance reasons, but it requires possibly difficult preparation. To achieve preparation-free monitoring, as an JVMTI agent could potentially provide, this thesis discusses runtime instrumentation and a new system.
public aspect Collection_UnsafeIteratorAspect {
  pointcut createIterator(Collection c) :
    call(Iterator Iterable+.iterator()) && target(c);
  after(Collection c) returning(Iterator i) : createIterator(c) {
    // auto-generated event handling routine
  }
  pointcut modifyCollection(Collection c) : /* omitted */ ;
  before(Collection c) : modifyCollection(c) {
    // auto-generated event handling routine
  }
  pointcut useIterator(Iterator i) : /* omitted */ ;
  before(Iterator i) : useIterator(i) {
    // auto-generated event handling routine
  }
}

Figure 6.7: Collection_UnsafeIteratorAspect, an aspect generated from the Collection_UnsafeIterator specification, shown in Figure 2.2 by JAVAMOP 3.0.

The work presented in this section is based on a preliminary one that has been done before the separation of concerns—firing an event and handling the event—was made. As a result, it will assume an old version of JAVAMOP, which does not separate the two concerns. However, the ideas are transferable to the new JAVAMOP.

Figure 6.6 shows the architecture of the system. A new system, called JAVAMOP-IJW, first takes one or multiple JAVAMOP specifications as input, and passes them to JAVAMOP 3.0, which yields an AspectJ aspect. Since JAVAMOP 3.0 does not separate the two concerns, the generated aspect contains not only event handling routines—equivalent to the routine generated by RV-MONITOR—but also pointcuts for specifying conditions for firing events. As an example, Figure 6.7 shows part of the aspect generated from the Collection_UnsafeIterator specification. This aspect may be thought of as an aspect generated by JAVAMOP 4.0 (Figure 5.3), where methods generated by RV-MONITOR (Figure 5.1) are inlined.

From the generated aspect, JAVAMOP-IJW generates a JAVAMOP-AGENT. A JAVAMOP-AGENT is a Java agent [39] that listens to the JVM’s class load event and modifies each loaded class in such a way that event handling methods can be invoked whenever the execution hits places where events should be fired.

Since the generated JAVAMOP-AGENT is independent of programs under monitoring, updating it is needed only when one wants to modify existing specifications or add new ones, which is unlikely to happen frequently. Suppressing benign violations or false alarms does not require updating it; editing the external configuration
file can prevent them (Section 6.3.3).

While generating a JAVAMOP-AGENT, JAVAMOP-IJW puts all of its dependencies into a JAR package, so that one does not need to manipulate the CLASSPATH environment variable, the command line option, or the Class-Path field in the manifest file. Thus, one can enable monitoring by simply adding the -javaagent flag:

```
$ java -javaagent:javamop_agent.jar Foo
$ java -javaagent:javamop_agent.jar -jar bar.jar
```

where Foo is a class file and bar.jar is a JAR package. One can also write a simple shell script that can be used as a drop-in replacement for the java executable.

### 6.3.3 Generating JAVAMOP-AGENT

Since a JAVAMOP-AGENT is the only file that is distributed to the user, it should be able to not only instrument loaded classes at runtime but also handle events. For this reason, a JAVAMOP-AGENT contains the runtime instrumentation module (Section 6.3.4), the event handling module (which is based on the code generated by JAVAMOP 3.0), and the startup module. The startup module initializes internal data structures and registers the runtime instrumentation module in such a way that it can listen to the JVM’s class load event. After this registration, the JVM invokes the instrumentation module whenever a class is about to be loaded, and this module instruments the class, as explained in Section 6.3.4. This section explains how the event handling module is generated from the aspect generated by JAVAMOP 3.0.

It would have been easier to build JAVAMOP-IJW on the modified AspectJ (Section 6.2.4), but, at the time of building this system, we did not consider fixing AspectJ’s instrumentation problem that causes the size of a method to exceed 64KB, because AspectJ is sophisticated and low-level bytecode manipulation is needed, as presented in Appendix A. Instead, we decided to develop our own instrumentation module that completely replaces AspectJ (Section 6.3.4).

This replacement requires modifications in the aspect generated by JAVAMOP, because AspectJ-specific constructs, such as aspects, advice and pointcuts, in the aspect are no longer valid. First, the generated aspect is transformed into an ordinary Java class because they are similar in the sense that they can contain fields and methods; in fact, an AspectJ compiler typically transforms an aspect into a singleton class. While methods and fields defined in the aspect are copied to the class as they are, advice and pointcuts are transformed, as explained below, and then copied.

Each advice is converted into a static method; e.g., the advice for handling the createIterator event, shown on lines 4–6 in Figure 6.7, is converted into the following:

```java
public static void createIterator(JoinPoint thisJoinPoint,
   Collection c, Iterator i)
```
The generated method takes all the parameters that the advice does, and thisJoinPoint. In AspectJ, thisJoinPoint is a special variable, available within an advice, that exposes information about the join point where the advice is inserted. Since this special variable is unavailable in a Java method, JAVAOP-IJW emulates it by taking an additional parameter and ensuring that the caller supplies it (Section 6.3.4).

The last construct, pointcut, cannot be placed in the generated class because there is no Java construct that corresponds to it, whereas an aspect or an advice has its counterpart in Java. JAVAOP-IJW collects all the information about pointcuts, and stores it in a separate file in its own format that enables the runtime instrumentation module (Section 6.3.4) to instrument loaded classes.

This file defines a list of event tuples, each of which corresponds to an event definition. Each event tuple consists of the name of the method for handling an event, the order between this method and the matched code, conditions for firing an event, and parameter bindings. For example, the event tuple for the createIterator event will be:

```java
Collection_UnsafeIteratorAspect.createIterator // method
AFTER // order
call Iterator Iterable+.iterator() // static condition
$0 c Collection // parameter binding and dynamic condition
$_ i Iterator // parameter binding
```

The first line indicates that Collection_UnsafeIteratorAspect.createIterator() is the method for handling this event, and the second line represents that handling the event should be done after the matched code is executed; in other words, the code for calling this method should be inserted after the matched code. The third line expresses the static condition for firing this event; in this case, an event may be fired whenever iterator() specified by the Iterable interface is called. The last two lines represent the parameter bindings and the dynamic conditions for parameters; c is bound to the target object ("$0"), and i is bound to the return value ("$_"). The third columns on these lines specify the expected types; e.g., the actual type of the target object should be Collection or its subclass (line 4).

It is notable that the reference type of the target is Iterable (line 3) but the actual type is expected to be Collection (line 4). This may look absurd—one may think that the reference type could be Collection as well—but the above is indeed the precise condition. For example, consider the following code:

```java
Collection c = new ArrayList();
```
Iterable iterable = c; // implicit upcasting

Iterator i = iterable.iterator();

If one sets the reference type to Collection, iterator() on line 3 would fail to fire an event, although the object is Collection. When the expected type in the dynamic condition is different from the reference type, the runtime instrumentation module adds code that checks whether the condition is satisfied using runtime type information, as explained Section 6.3.4.

After transforming the generated aspect into a pure Java class, JAVAMOP-IJW runs a Java compiler and then packages this compiled event handling module, the startup module, and instrumentation module as a JAVAMOP-AGENT.

### 6.3.4 Runtime Instrumentation

As explained in Section 6.3.3, a JAVAMOP-AGENT has its own instrumentation module, which is activated at runtime, in order to avoid AspectJ’s limitation, explained in Section 6.2.4. During initialization of a JAVAMOP-AGENT at runtime, this module reads all the event tuples (Section 6.3.3) that were written in a separate file when the JAVAMOP-AGENT was generated. The startup module in the JAVAMOP-AGENT then registers the instrumentation module to the JVM.

Once the registration is done, whenever a class is about to be loaded, the JVM invokes the instrumentation module with a byte array that represents the class in the class file format. This module then instruments each method and constructor in the class, and returns to the JVM the instrumented class in a byte array, which will be finally loaded to the JVM.

The runtime instrumentation module is built on Javassist, a Java library for Java bytecode manipulation. This library was chosen because it provides a high-level API for inserting code.

Below how each method is instrumented is explained; one can assume that a constructor is similarly instrumented. Although instrumentation is explained at the source code level for readability, it is actually done at the bytecode level.

The basic flow during instrumenting a method consists of two steps: matching—this picks out places where an event may be fired—and inserting an invocation of the corresponding method for handling the event, before or after each matched place.

To find such places, the instrumentation module iterates over each expression in a method, and checks whether the expression matches with the static condition in an event tuple. This check involves checking the method name, the enclosing class, the return type, and the parameter types, but does not consider the actual types of

---

14To be invoked by the JVM, this instrumentation module implements the ClassFileTransformer interface, and it is registered by invoking Instrumentation.addTransformer().
parameters. For example, the following method invocation will be considered being matched with the iterator event, mentioned in Section 6.3.3, regardless of the actual type of iterable:

```java
Iterable iterable = // not necessarily Collection
Iterator i = iterable.iterator();
```

In this example, matching does not necessarily result in firing an event because the dynamic condition in the event tuple states that the actual type should be Collection or its subclass. Since the actual type is unknown during instrumentation, deciding whether this method invocation should fire an event is deferred until that code is executed. To check the actual type later, the instrumentation module inserts a guard, as explained below.

For each matched place, an invocation of the method for handling this event is inserted. When a guard is necessary, the instrumentation module wraps the guard around that invocation, so that the method is invoked only when the dynamic condition is satisfied. For example, the above code fragment is converted into the following (each of inserted lines is prefixed by "+"):

```java
+ if (iterable instanceof Collection) {
+     JoinPoint thisJoinPoint =
+         JoinPoint.fromStaticInfo("java.lang.Iterable");
+     Collection_UnsafeIteratorAspect.createIterator(thisJoinPoint,
+         (Collection)iterable, i);
+ }
```

Here the guard is shown on line 3. According to the dynamic condition in the event tuple, this guard ensures that an event occurs only if the target object is Collection. On lines 6–7, the generated static method for handling this event is invoked and, as a result, the event is handled by the JAVAMOP-generating code. In this example, the inserted code is placed after the matched code, as specified in the event tuple; if BEFORE is specified, the inserted code would be placed between lines 1 and 2.

As mentioned in Section 6.3.3, AspectJ’s thisJoinPoint needs to be emulated and the caller should supply it. For this purpose, an object is created and passed as an argument on lines 4–5 in the above code. In the context of monitoring, the uses of thisJoinPoint are limited to four kinds: accessing the enclosing file name, the line number, the unique identifier of the matched join point (as an integer), and the Class object corresponding to the matched code (e.g., Iterable in this example). Providing the first two kinds is straightforward because they can be obtained by walking the call stack. To provide the other two kinds, a JAVAMOP-AGENT inserts an
invocation of JoinPoint.fromStaticInfo(), which returns a JoinPoint object. By assigning a unique number for each object, the unique identifier can be served, and the Class object can be served through reflection; i.e., calling Class.forName().

Instrumentation increases the size of a method; e.g., the above instrumented code has additional chunk of code (lines 3–8). Although the size of such a chunk is moderate, a method may have excessive number of additional chunks because it may have many matched places and each place may need to fire multiple events, especially when multiple specifications are monitored. As a result, careless instrumentation may cause the size of a method to exceed 64KB, Java’s limit, like AspectJ does (Section 6.2.4).

To avoid this problem, the runtime instrumentation module also extracts a method from the matched code and replaces the code by an invocation of the extracted method. The main idea is the same as the modification this thesis presents for AspectJ, explained in Section 6.2.4 and Appendix A, but this module applies such technique only if the matched code needs to fire multiple events, assuming that it might be expensive to invoke a method and the JIT compiler may fail to inline it. In contrast, the modified AspectJ unconditionally applies that technique due to technical difficulties. With this solution, the instrumentation module was able to instrument all the benchmarks in DaCapo 9.12 (Section 6.3.6).

6.3.5 Configuration

Delaying instrumentation until runtime results in another benefit; one can alter the behavior of monitoring by simply editing the configuration file. As mentioned in Section 6.3.2, the external configuration file enables one to suppress violations of certain specifications.

Instrumentation can be also configured to print event log, which is useful to trace the cause of a violation. Since additional code should be inserted for logging, compile-time instrumentation would require one to weave the program again. In contrast, one can enable or disable it without the need to generate a JAVAMOP-AGENT again.

6.3.6 Evaluation

This section discusses the convenience of JAVAMOP-IJW and the runtime overhead of a JAVAMOP-AGENT.

A JAVAMOP-AGENT does not retrieve the Class object during instrumentation; instead, it inserts code for the retrieval, because the resulting Class objects would not be the same and, consequently, may cause a problem if the user performs operations on this object. They are different because a JAVAMOP-AGENT uses its own class loader and the JVM separately considers classes loaded by different class loaders, though they have the same fully qualified name.
Preparation Step

We generated a JAVAOP-AGENT from the 179 specifications discussed in Chapter 5, by simply passing them to JAVAOP-IJW. This single JAVAOP-AGENT could be reused throughout the entire experiment, because it is independent of programs to be monitored; it can be used for any program.

From a usability perspective, it is beneficial to have such universal agent, because it eliminates significant effort and time that compile-time instrumentation usually requires. For example, to instrument DaCapo 9.12, one needs to do a series of tasks: unzipping the DaCapo 9.12 package, instrumenting each benchmark and each dependency in it, manipulating the manifest file (Section 6.3.1) for each instrumented file, and finally zipping all the instrumented files and others. In particular, instrumenting DaCapo 9.12 also requires some knowledge on it because it keeps all the dependencies in a JAR file in the package, and uses its own class loader to find classes in them; without knowing such details, instrumentation would be incomplete. These entire tasks took about 12 minutes when they were done in a batch mode. In contrast, the JAVAOP-AGENT could use the original DaCapo 9.12 package as it is.

Instrumentation

As mentioned in Sections 6.3.3 and 6.3.4, instrumentation can cause the size of the resulting method to exceed the limit. The instrumentation module in the JAVAOP-AGENT was able to instrument all the benchmarks of DaCapo 9.12, which shows that our technique that extracts a method from a join point (Section 6.3.4) is effective in handling such an extreme case.

Runtime Overhead

Excessive execution time or memory usage would greatly limit the effectiveness of a JAVAOP-AGENT. In order to test whether overhead could be maintained at acceptable levels, we measured overhead while monitoring each benchmark of DaCapo 9.12 with all the 179 specifications simultaneously. We used the DaCapo’s default data input size, and the -converge option, which guarantees that the resulting execution times converge within 3%. These experiments were performed on an Intel Core 2 Duo 2.40GHz-based machine with 4 GB of memory under Windows 7 (64 bit) with Java Platform Standard Edition 6 (build 1.6.0_21).

Table 6.4 summarizes the converged execution time and the peak memory usage. For most benchmarks, millions of events were observed; in particular, AVRORA, H2 and Pmd emitted 32,804,400, 65,647,663 and 48,866,293 events, respectively, as mentioned in Section 6.2.3. For some benchmarks, millions of parameter bindings
Table 6.4: Converged execution time (in second) and peak memory usage (in MB).

<table>
<thead>
<tr>
<th></th>
<th>Execution time (s)</th>
<th>Peak Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
<td>JAVAOP-AGENT</td>
</tr>
<tr>
<td>AVRORA</td>
<td>4.52</td>
<td>41.69</td>
</tr>
<tr>
<td>BATIK</td>
<td>1.65</td>
<td>2.88</td>
</tr>
<tr>
<td>ECLIPSE</td>
<td>40.72</td>
<td>41.60</td>
</tr>
<tr>
<td>FOP</td>
<td>0.50</td>
<td>32.32</td>
</tr>
<tr>
<td>H2</td>
<td>6.72</td>
<td>14.09</td>
</tr>
<tr>
<td>JYTHON</td>
<td>3.47</td>
<td>16.68</td>
</tr>
<tr>
<td>LINDEX</td>
<td>1.61</td>
<td>2.31</td>
</tr>
<tr>
<td>LUSEARCH</td>
<td>4.33</td>
<td>15.45</td>
</tr>
<tr>
<td>PMD</td>
<td>2.73</td>
<td>37.52</td>
</tr>
<tr>
<td>SUNFLOW</td>
<td>7.81</td>
<td>18.91</td>
</tr>
<tr>
<td>TOMCAT</td>
<td>3.71</td>
<td>17.13</td>
</tr>
<tr>
<td>TRADEBEANS</td>
<td>6.66</td>
<td>7.08</td>
</tr>
<tr>
<td>TRADESOAP</td>
<td>58.10</td>
<td>54.16</td>
</tr>
<tr>
<td>XALAN</td>
<td>5.87</td>
<td>16.70</td>
</tr>
</tbody>
</table>

were created at runtime; e.g., AVRORA and PMD, respectively, caused the JAVAOP-AGENT to create 1,012,665 and 1,021,405 parameter bindings. In such extreme cases, the execution time was at most 14 times slower. The execution time of FOP was much longer, but this is because there were 4,570 violations and reporting a violation takes significant time due to walking the call stack for comprehensive error messages. When we disabled violation reports, the execution time was reduced to 5.06 seconds. The memory overheads in those extreme cases were at most 421%. Memory overhead may look very significant in some cases, such as LINDEX, but this was mainly because the JAVAOP-AGENT needs certain amount of memory regardless of programs—it needs to load its modules and maintain its own type hierarchy for instrumentation—and that amount can be relatively large when the original benchmark consumes small amount of memory.
Chapter 7

Conclusion

This chapter explains the limitations of the presented work, and then concludes.

7.1 Limitations

The learning process of jMiner (Chapter 3) is limited to the observed behaviors, which is an inherent limitation of all dynamic approaches. For example, the specifications in Figures 4.14 and 4.15 wrongly enforce the order between getInputStream() and getOutputStream() because this was consistently observed in the training set. Another surprising result is shown in Figure 7.1: unlike one expects, the inferred specification allows the invocation of nextToken() and hasMoreTokens() in an arbitrary order.

⟨init⟩(s)

Figure 7.1: StringTokenizer specifications inferred by jMiner.

This pattern is based on an actual interaction observed from xalan, a benchmark in DaCapo 9.12—it invoked nextToken() without calling hasMoreTokens(). After inspecting the source code of xalan, we could see that the interaction is not defective because it first retrieves the number of tokens by calling countTokens() and then consecutively calls nextToken() as many times as specified by countTokens(). Due to countTokens(), a specification on StringTokenizer cannot be stricter than the one in Figure 7.1. Considering countTokens() as well does not improve the specification because jMiner cannot infer that the return value of this method indicates the number of allowed nextToken() calls. This limitation is inherent to all FSA-based approaches: an FSA cannot count.

One obvious and significant limitation of the methodology for writing specifications from documentation (Chapter 5) is that it is time-consuming. Approximately
five person-months were spent on formalizing the four packages of the Java API, including writing defective programs (Section 5.3.1).

Another drawback of this methodology is that some specifications may be missing for three reasons. First, the current API specification, provided by a Java platform, may miss important contracts—it is almost unachievable for platform designers to comprehensively describe all the contracts. Second, there can be optional yet practically desirable patterns. For example, if an OutputStream object is constructed on top of an underlying ByteArrayOutputStream object, it should be flushed or closed before the underlying object's toByteArray() is invoked. This behavioral pattern is indeed desirable because failing to fulfill the requirement may cause toByteArray() to return incomplete contents; however, this pattern is undocumented because it is not required to follow it all the time. Third, we may have overlooked specification-implying text, although we systematically kept track of what we have read, by adding tags and developing PropDoc, a tool for collecting coverage statistics and unread chunks of text, in order to avoid this as much as possible.

One drawback of a JAVAMOP-Agent is that its runtime instrumentation module cannot automatically find classes that only a custom class loader can find. This module needs to find classes in order to retrieve the type information and match with event definitions. For example, in order to determine whether a method is matched with Iterator+.next(), which means next() of Iterator or any of its subclasses, it should first read the enclosing class and determine whether it implements Iterator. Although a JAVAMOP-Agent can automatically find any class that the default class loader would find, it requires the user to provide the paths to classes if they can be found only through a custom class loader. However, it is believed that this is not a severe limitation because ordinary programs usually use only the default class loader. Also, providing the paths is still more convenient than instrumenting the dependencies, which requires one to find them, instrument them, and manipulate manifest files.

7.2 Conclusion

Runtime verification has not been adopted by developers and users as an essential tool, despite its usefulness and many improvements on performance. Reasons of reluctance include that existing runtime monitoring systems and papers define at most several specifications and measure performance overhead based only on them, and, with such limited experiments, developers may not be convinced of the usefulness and still wonder “will this really work and yield useful results if I manage to write hundreds of formal specifications?” One may also wonder if preparation steps and runtime overheads are reasonable.
This thesis attempted to answer these questions: it presented 179 parametric specifications that are carefully written and ready to be used, as well as an automated mining system that can be used if one does not want to spend time on writing specifications; and an approach to a monitoring system that is efficient, convenient and extensible. On the top of a system that is already efficient and supports various formalisms, this presented work added further improvements and engineering effort for monitoring multiple specifications, and thoroughly tested it using the 179 specifications and real world applications. Also, the modular design presented in this thesis enables one to build a new system, if a different instrumentation method is needed, which will still be powered by all the optimizations that several researchers have devised for several years.

The empirical study in this thesis also showed that runtime monitoring is indeed capable of revealing bugs and suggestions. Based on this experience, it seems safe to claim that runtime monitoring systems like the one presented in this thesis are already useful and it is worth trying them.
Appendix A

Weaving for Monitoring Multiple Specifications

As we explained in Section 6.2.4, weaving increases the code size, which can result in a failure due to Java’s limit on the size of a method. To avoid such failures, Section 6.2.4 presents a technique that extracts a method from a join point and replaces the join point by an invocation of the extracted method. Although this technique is motivated by the desire to enable runtime monitoring in extreme cases, it could also be adopted as a general purpose technique by the AspectJ developers. In fact, this is a known issue reported by several users in the AspectJ community. This appendix discusses this technique.

Among various pointcuts, some pointcuts, such as execution and static initialization, cannot be matched more than once in a method, and it is very unlikely that they cause a failure. In contrast, the method call, constructor call, field reference and field set pointcuts can be matched arbitrarily many times, and each match results in at least several additional instructions, as briefly explained in Section 6.2.4. In fact, the failure we observed in ror of DaCapo 9.12 is caused by excessive number of join points that match method call and constructor call pointcuts in a method. With this in mind, we focused on avoiding the increment in the code size for these pointcuts.

To avoid the increment, we extract a method from each matched join point and replace the join point by a method invocation of the extracted method. As a result, all the necessary instrumentation is performed in the extracted method, instead of the originating method, because the matched join point has been moved from the latter to the former. For example, consider the following code fragment:

```java
void originating(Collection c) {
    c.add("hello world");
    Iterator i = c.iterator();
    i.hasNext();
}
```

When this code is monitored against the Collection_UnsafeIterator specification, shown in Figure 2.1, each of lines 2–4 has a matched join point. With our technique, three methods will be therefore extracted:

```java
static boolean extracted_from_line2(Collection c, Object elem) {
```
Also, each matched join point will be conceptually replaced by a method invocation, as follows:

```java
void originating(Collection c) {
    extracted_from_line2(c, "hello world");
    Iterator i = extracted_from_line3(c);
    extracted_from_line4(i);
}
```

Here the replacement is described at the source code level for readability, but the replacement is actually performed at the bytecode level. We below show that this replacement does not increase the code size of the originating method.

### A.1 Method Call Pointcut

In Java, there are a few instructions for invoking a method, such as `invokevirtual` and `invokestatic`, and all of them have the almost same calling convention: the caller pushes the target object (if this exists) and arguments (from left to right) onto the stack, and then the callee consumes them and pushes the return value onto the stack (if it exists). For example, the call site of `Collection.add()` (line 2 in the original code) is compatible with that of `extracted_from_line2()` (line 2 in the modified code) because both of them expect two objects on the stack. In general, the following two call sites are compatible:

\[
\text{ret} = \text{target}.\text{non}\_\text{static}\_\text{method}(\text{arg}_1, \text{arg}_2, \cdots, \text{arg}_n); \\
\text{ret} = \text{static}\_\text{method}(\text{target}, \text{arg}_1, \text{arg}_2, \cdots, \text{arg}_n);
\]

Since the presented technique extracts a method in such a way that it is compatible with the replaced callee, the only modification made in the caller is to replace the original invoke instruction (`invokevirtual in this case`) with an `invokestatic`
instruction—the extracted method is always a static method. Replacing the single instruction also suffices for a static method; in this case, the replaced and the extracted methods will have the exactly same signature. Also, the size of an `invokestatic` instruction is shortest among all the method invocation instructions; i.e., replacing an instruction does not increase the code size. Therefore, the technique avoids the increment in the code size in case of a method call pointcut.

### A.2 Field Reference and Field Set Pointcuts

Java instructions for getting and setting the value of a field are also similar to method invocation instructions in the sense that the caller pushes the target object and the new value onto the stack (if they exist) and the callee pushes the retrieved value onto the stack (if it exists). In other words, one can view getting a value as invoking a method that takes no parameters and returns a value, and setting a value as invoking a method that takes one parameter and returns nothing. As a result, the following two different statements manipulate the stack in the same way:

```java
t = target.non_static_field;
ret = static_method(target);
```

The presented technique extracts a method for a field access in such a way that a drop-in replacement is possible, like it handles the method call pointcut. Since the size of any field access instruction is no shorter than that of an `invokestatic` instruction for invoking the extracted method, the technique does not increase the code size for a field access.

### A.3 Constructor Call Pointcut

One may be tempted to handle a constructor call pointcut by replacing a single instruction, like a method call pointcut, because calling a constructor is indeed implemented by an `invokespecial` instruction, one of the method invocation instructions. However, this results in a verification failure because a constructor can be invoked only on an uninitialized object, created by a `new` instruction, but the replacement, which moves the `invokespecial` instruction to the extracted method, causes the intra-procedural verifier to fail to recognize the newly created object as uninitialized.

To avoid such verification failures, the presented technique moves not only the `invokespecial` instruction but also the corresponding `new` instruction and some

---

1A typical error message is "expecting to find uninitialized object on stack."
others. This movement requires a careful instruction manipulation because creating an object and assigning it to a variable are done in a series of instructions. For example, consider the following Java code fragment:

\[
newobj = \text{new ClassName}(arg_1, arg_2, \ldots, arg_n);
\]

From this code, a Java compiler generates the following:

1. new // create an object of ClassName type
2. dup // duplicate the created object
3. ... // prepare arg_1, arg_2, ..., arg_n
4. invokespecial // invoke the constructor
5. astore_1 // store the created object in 'newobj'

Here a Java compiler inserts \texttt{dup} because the created object is used twice: once for providing the target object of the constructor call (line 4), and for storing in the variable (line 5). What is represented by line 3 can be arbitrarily many instructions because multiple arguments can exist and preparing an argument may require many instructions. Moreover, it may contain another object creation—one of arguments can be another newly created object. In order to find the exact \texttt{new} and \texttt{dup} instructions that correspond to the \texttt{invokespecial} instruction, the presented technique considers the stack depth while iterating over instructions backwards.

After identifying the corresponding \texttt{new} and \texttt{dup}, it first extracts a static method from lines 1, 2 and 4:

\begin{verbatim}
static ClassName from_124(type_1 arg_1, ..., type_n arg_n) {
  return new ClassName(arg_1, arg_2, ..., arg_n);
}
\end{verbatim}

where \texttt{type}_i is the type of \texttt{i}-th parameter of the constructor. At the bytecode level, the body of this method is the following (each instruction moved from the caller is prefixed by “+”):

\begin{verbatim}
+ new
+ dup
... // load arg_1, arg_2, ..., arg_n
+ invokespecial
  areturn // return the created object
\end{verbatim}

Then, in the originating method, \texttt{invokespecial} is replaced by \texttt{invokestatic}, so that the extracted method is invoked; consequently, the remaining code looks like the following (an instruction that newly appears is prefixed by “+”):

\footnote{If there is no need to store the created object, however, \texttt{dup} may not appear.}
... // prepare arg_1, arg_2, ..., arg_n
+ invokestatic // invoke the extracted method
astore_1 // store the created object in 'newobj'

Since the extracted method takes the exactly same arguments as the constructor, it is possible to pass the arguments, which is prepared on line 1, to the extracted method. Also, the stack after executing the above invokestatic instruction contains exactly one reference to the created object, which is the same as the stack after executing the replaced invokespecial instruction. Therefore, this replacement is correct. The code size of the caller is not increased because new (and also dup if this exists) is moved out and invokespecial is replaced by invokestatic, which is no longer than invokespecial.
References


