5-2011

Exploiting Program and Property Structure for Efficient Runtime Monitoring

Rahul Purandare
University of Nebraska - Lincoln, rpuranda@cse.unl.edu

EXPLOITING PROGRAM AND PROPERTY STRUCTURE FOR EFFICIENT RUNTIME MONITORING

by

Rahul A. Purandare

A DISSERTATION

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Doctor of Philosophy

Major: Computer Science

Under the Supervision of Professor Matthew B. Dwyer

Lincoln, Nebraska

May, 2011

EXPLOITING PROGRAM AND PROPERTY STRUCTURE FOR EFFICIENT RUNTIME MONITORING

Rahul A. Purandare, Ph. D.

University of Nebraska, 2011

Adviser: Matthew B. Dwyer

Modern software systems are complex and often built using components that are provided with their application programming interface (API) to assist a user. However, this API is informal and if used incorrectly, may lead to bugs that are hard to detect. In order to address the problem of API conformance checking, researchers have proposed various analysis techniques including static and dynamic typestate analysis. However, it is extremely challenging to develop a static analysis that is both precise and scalable. On the other hand, dynamic analysis or runtime monitoring of programs may incur heavy overhead, thereby limiting its application only to a subset of realistic programs. This heavy overhead could be a result of handling of the monitors that are created during runtime, or the events generated by program instrumentation, or some other factors related to program and property interaction.

Our research focuses on developing techniques that optimize program instrumentation to reduce the monitoring overhead without compromising error reporting. The techniques are guided by the cost models that we have developed for runtime monitoring and based on a hybrid approach that combines static with dynamic typestate analysis to exploit the benefits of both approaches. In addition, the approach also leverages the property structure to make monitor optimization more effective.

In this dissertation, we present cost models for runtime monitoring that are based on our understanding of the existing monitoring tools. The cost models describe key factors

that influence the monitoring overhead as well as the relationship among them. We develop two novel analysis techniques, namely the residual analysis and the stutter-equivalent loop transformation, that target the number of events as that is a primary factor associated with the total cost of monitoring. We present the results of their evaluation based on some open source applications and benchmarks that show that the techniques can effectively reduce the monitoring overhead.

DEDICATION

To my late and beloved grandfather, Sadashiv Tryambak Purandare, who introduced me to the wonderful world of science.

ACKNOWLEDGMENTS

I owe my deepest gratitude to my adviser Dr. Matthew Dwyer for his guidance, mentoring, patience and support. Working with him has been a truly rewarding and pleasant experience. The depth and breadth of his knowledge about my field of research and his positive approach have been strong driving factors for me and my research. I attribute a major part of my interest in the field of research to him. I would also like to thank my Ph.D. committee members Dr. Sebastian Elbaum, Dr. Witawas Srisa-an and Dr. Shane Farritor for giving me their valuable time and suggestions. My special thanks to Dr. Elbaum as I worked with him on many research projects and greatly benefited from his approach and immense knowledge of this field of research.

My sincere thanks go to the CSE faculty for the inspiration as well as the training that they provided to me. I also thank the entire staff of the CSE department for providing their support that helped the students, including me, to stay focused on our goals. I am grateful to the ESQuaRed lab members for their help, patience, and technical, as well as fun, discussions that I regularly had with them. Special thanks to Dr. Suzette Person and Jiangfan Shi for the kind help and friendship that they offered. I also thank Dr. Eric Bodden for his help and the technical discussions that I had with him through email.

I decided to do a Ph.D. after a long career in the IT industry. I received strong encouragement from my previous teachers from the University of Pune, Achyut Roy and Rustom Mody. Dr. Rajeev Alur and Avinash Marathe offered valuable advice that helped me proceed in the right direction when I started with my Ph.D. applications. I am grateful to all of them for giving me their time, providing me valuable guidance and also trusting my abilities.

This venture would not have been possible without the support of my family. I am grateful to my parents, Sulochana and Ashok, my aunt, Sudha, and my brother, Atul for the

love and support they have provided throughout my life. My early interest in science was stimulated by my beloved and late grandfather, Sadashiv, who ensured that his grandson would read informative and inspirational books about science and scientists at a tender age. He has been a constant source of inspiration for me and my family.

Finally, I thank my wife, Swapna and my kids, Salil and Maitreyi for their unconditional love and support. On many occasions, Salil and Maitreyi showed a maturity well beyond their age while dealing with the challenges they faced during this journey. Swapna has always been the rock of my family and because of her support I am certain we will have many more fascinating ventures in the future.

Contents

Contents **viii**

List of Figures **xiii**

List of Tables **xviii**

1 Introduction **1**

1.1 Motivation 1

1.2 Approach 5

1.3 Simple Code and Property Example 6

1.3.1 Cost Models 8

1.3.2 Residual Typestate Analysis 10

1.3.3 Stutter-equivalent Loop Transformation 13

1.4 Theses 15

1.5 Contributions 16

1.6 Outline of Dissertation 18

2 Background and Related Work **19**

2.1 A General Monitoring Scheme 20

2.2 Monitoring Internals 21

2.3 Monitoring Fundamentals . . . 23
2.3.1 Complete Matching . . . 26
2.3.2 Suffix Matching . . . 28
2.4 Property violation and Monitor Correctness . . . 30
2.5 Related Work . . . 31
2.5.1 Static Approaches . . . 31
2.5.1.1 Type-system based approaches . . . 31
2.5.1.2 Static analysis based approaches . . . 33
2.5.2 Dynamic Approaches . . . 34
2.5.2.1 Static instrumentation based approaches . . . 34
2.5.2.2 Dynamic instrumentation based approaches . . . 36
2.5.2.3 Sampling based approaches . . . 37
2.5.3 Hybrid Approaches . . . 40
2.5.4 Specification Mining . . . 46
2.5.4.1 Machine learning based approaches . . . 47
2.5.4.2 Pattern template based approaches . . . 48
2.5.4.3 Static approaches . . . 49

3 A Cost Model for Runtime Monitoring 50

3.1 Introduction . . . 50
3.2 Motivation and Background . . . 52
3.2.1 Unexplained variation . . . 52
3.2.2 Our Motivation . . . 55
3.2.3 Monitoring approaches . . . 56
3.2.3.1 Basic Definitions . . . 63
3.2.3.2 Monitor Indexing . . . 65

3.3 Cost Models . . . 67
3.3.1 C_C: Creating Monitors . . . 68
3.3.2 C_R: Index Trees . . . 70
3.3.3 C_I: Inserting Monitors . . . 73
3.3.4 C_V: Traversing Monitors . . . 75
3.3.5 C_T: Performing Transitions . . . 77
3.3.6 Comparing operational costs . . . 78
3.4 Reexamining unexplained variation . . . 78
3.5 Partial Validation of the Cost Models . . . 81
3.6 Revisiting Previous Findings . . . 88
3.6.1 Techniques' scope and generalization . . . 89
3.6.2 Conjectures about sources of overhead . . . 91
3.6.3 Missed optimizations opportunities . . . 92
3.7 Evolution of Cost Models . . . 94

4 Residual Analysis **96**

4.1 Introduction and Motivation . . . 96
4.2 Overview . . . 99
4.2.1 A Simple Static Typestate Analysis . . . 101
4.2.2 Leveraging Inconclusive Analysis Results . . . 103
4.3 Soundness and Completeness . . . 105
4.4 Residual Program Analysis . . . 108
4.4.1 Typestate Checking . . . 108
4.4.2 Safe program regions . . . 110
4.4.3 Calculating safe regions . . . 112
4.4.4 Program analysis residue . . . 119

4.5 Evaluation . . . 119
4.5.1 Implementation . . . 120
4.5.2 Artifacts . . . 122
4.5.3 Results and Discussion . . . 125
4.6 Lessons Learned from the Evaluation . . . 127
4.7 Acknowledgements . . . 128

5 Stutter-Equivalent Loop Transformation 129

5.1 Introduction and Motivation . . . 129
5.2 Overview . . . 130
5.3 Our Approach . . . 133
5.4 Terminology and Definitions . . . 136
5.4.1 Runtime Path Property Monitoring . . . 136
5.4.2 Stutter Equivalence . . . 137
5.4.3 Phrase Stuttering . . . 139
5.4.4 Stutter Equivalent Loops and Programs . . . 140
5.5 Analysis and Algorithms . . . 143
5.5.1 Static Analysis . . . 143
5.5.2 Checking for Unit Stuttering . . . 144
5.5.3 Transforming Unit Stuttering Loops . . . 149
5.5.4 Generalizations . . . 151
5.6 Soundness and Completeness . . . 155
5.6.1 Checked exceptions . . . 156
5.6.2 Unchecked exceptions . . . 158
5.6.3 Try-catch blocks inside loops . . . 159
5.7 Evaluation . . . 160

5.7.1 Artifacts . . . 160
5.7.2 Design . . . 161
5.7.3 Measures . . . 162
5.7.4 Infrastructure . . . 163
5.7.5 Results and Discussion . . . 166
5.8 Observations and Lessons Learned for the Evaluation . . . 168
5.9 Acknowledgements . . . 172

6 Conclusion and Future Work 173
6.1 Conclusion . . . 173
6.2 Future Work . . . 175
6.2.1 A Novel Optimization Technique . . . 175
6.2.1.1 Monitor Reclamation Analysis . . . 176
6.2.2 Symbol-based Monitoring . . . 177
6.2.2.1 Targeting Property Self-loops . . . 182
6.2.3 Improvement Techniques . . . 186
6.2.4 Broader Study . . . 188
6.2.5 Beyond Error Detection . . . 190

A Java programs used in the study with JavaMOP 191

B Java programs used in the study with Tracematches 200

Bibliography 217

List of Figures

1.1 Property SocketChannel. 2
1.2 Property FailSafeIter. 7
1.3 A code snippet based on a method from `DaCapo` benchmark `bloat`. 8
1.4 The instrumented code snippet for the property `FailSafeIter`. 9
1.5 The output of residual analysis. 12
1.6 Stutter-equivalent Loop Transformation Example: original (left) and transformed (right). 14

2.1 General Monitoring Scheme 19
2.2 Monitoring Internals 21
2.3 Property HasNext for complete matching. 23
2.4 Property HasNext for suffix matching. 23
2.5 Property FailSafeIter for suffix matching. 24
2.6 A code snippet based on a method from `DaCapo` benchmark `bloat`. 27
2.7 Simple complete matching algorithm. 28
2.8 Simple suffix matching algorithm. 29

3.1 Simple object-based monitoring scheme for complete matching. 57
3.2 Property FailSafeIter 58
3.3 Simple object-based monitoring algorithm. 59

3.4 Simple state-based monitoring scheme for complete matching. 61
3.5 Basic state-based monitoring algorithm. 62
3.6 Multi-level indexing corresponding to Figure 3.1(a) 66
3.7 Cost components for object-based monitoring 68
3.8 Cost components for state-based monitoring 69
3.9 C_C: Cost of creating monitors. 70
3.10 C_R: Cost of accessing monitors. Object-based monitoring (Top) and State-based monitoring (Bottom). 70
3.11 C_I: Cost of inserting monitors into pools. Object-based monitoring (Top) and State-based monitoring (Bottom). 74
3.12 C_V: Cost of traversing monitor pools. Object-based monitoring (Top) and State-based monitoring (Bottom). 75
3.13 C_T: Cost of performing transitions. Object-based monitoring (Top) and State-based monitoring (Bottom). 77
3.14 Property 1: `create ((update empty) | (hasNext next))+ empty` 86
3.15 Property 2: `create (update | next)+ empty` 87

4.1 Residual Typestate Analysis Architecture. 98
4.2 SocketChannel API Example 99
4.3 SocketChannel API Typestate FSA 101
4.4 Example CFG and Static Typestate Analyses 103
4.5 An instrumented code snippet. 106
4.6 Residual Analysis Output. 107
4.7 Refactored `transformData` Example 109
4.8 Calculating Safe Regions. 114
4.9 Region matrix and typestate flow example. 115

4.10 Mark Safe Regions. 116
4.11 Folder API Typestate FSA. 123
4.12 Residual Folder API Typestate FSA 123
4.13 Conditional Close Refactoring. 124

5.1 Example from class `SSAPRE`: original (left) and transformed (right) 131
5.2 `HasNext` property, ϕ_{it}, as an FSA 132
5.3 Example from class `SSAPRE`: Modified to show the effect of m. 145
5.4 Loop phrase automaton construction 146
5.5 Checking a Loop for Unit Stuttering Distance. 147
5.6 Shadow property automaton 147
5.7 Unit Stuttering Loop Transformation 150
5.8 Original loop (left) and Transformed loop (right). 152
5.9 More Precise Algorithm for Checking a Loop for Unit Stuttering Distance. . . . 153
5.10 Efficient and Precise Algorithm for Checking a Loop for Stuttering Distance 2. 155
5.11 Stutter distance checking in the presence of exceptions. 157
5.12 Architecture for the stutter-equivalent loop transformation analysis. 164

6.1 Monitor Reclamation. 177
6.2 Simple symbol-based monitoring scheme for complete matching. 178
6.3 Simple object-based monitoring scheme for complete matching. 179
6.4 Transitioning from state 1 to state 2. 184
6.5 Conditionally transformed loop: original code. 186
6.6 Conditionally transformed loop: transformed code. 187

A.1 Changing the number of associated monitors keeping the number of uni-monitor events same. Scale = 1. 192

A.2 Changing the number of associated monitors keeping the number of uni-monitor events same. Scale = 10. 193
A.3 Changing the number of associated monitors keeping the number of multi-monitor events same. Scale = 1. 194
A.4 Changing the number of associated monitors keeping the number of multi-monitor events same. Scale = 10. 195
A.5 Changing the number of uni-monitor events keeping the number of associated monitors same. Scale = 1. 196
A.6 Changing the number of uni-monitor events keeping the number of associated monitors same. Scale = 10. 197
A.7 Changing the number of multi-monitor events keeping the number of associated monitors same. Scale = 1. 198
A.8 Changing the number of multi-monitor events keeping the number of associated monitors same. Scale = 10. 199

B.1 Changing the number of associated monitors keeping the number of uni-monitor events same (Property 1). Scale = 1. 201
B.2 Changing the number of associated monitors keeping the number of uni-monitor events same (Property 1). Scale = 10. 202
B.3 Changing the number of associated monitors keeping the number of multi-monitor events same (Property 1). Scale = 1. 203
B.4 Changing the number of associated monitors keeping the number of multi-monitor events same (Property 1). Scale = 10. 204
B.5 Changing the number of uni-monitor events keeping the number of associated monitors same (Property 1). Scale = 1. 205

B.6 Changing the number of uni-monitor events keeping the number of associated monitors same. Scale = 10. 206
B.7 Changing the number of multi-monitor events keeping the number of associated monitors same (Property 1). Scale = 1. 207
B.8 Changing the number of multi-monitor events keeping the number of associated monitors same (Property 1). Scale = 10. 208
B.9 Changing the number of associated monitors keeping the number of uni-monitor events same (Property 2). Scale = 1. 209
B.10 Changing the number of associated monitors keeping the number of uni-monitor events same (Property 2). Scale = 10. 210
B.11 Changing the number of associated monitors keeping the number of multi-monitor events same (Property 2). Scale = 1. 211
B.12 Changing the number of associated monitors keeping the number of multi-monitor events same (Property 2). Scale = 10. 212
B.13 Changing the number of uni-monitor events keeping the number of associated monitors same (Property 2). Scale = 1. 213
B.14 Changing the number of uni-monitor events keeping the number of associated monitors same (Property 2). Scale = 10. 214
B.15 Changing the number of multi-monitor events keeping the number of associated monitors same (Property 2). Scale = 1. 215
B.16 Changing the number of multi-monitor events keeping the number of associated monitors same (Property 2). Scale = 10. 216

List of Tables

3.1 Variation in the runtime overheads caused by different combinations program, property, input and tool configurations. * Indicates that the process was stopped after 7200 seconds. JM = JavaMOP and TM = Tracematches. 53

3.2 Comparing operational costs. 79

3.3 JavaMOP: Overhead due to events and monitors associated with receiver objects (Time as relative overhead). 83

3.4 Tracematches (Property 1): Overhead due to events and monitors associated with receiver objects (Figures indicate relative overhead). 85

3.5 Tracematches (Property 2): Overhead due to events and monitors associated with receiver objects (Figures indicate relative overhead). 85

3.6 Property `HastNext` - Overheads are expressed in percentages over the original uninstrumented versions (4.8s for bloat and 4.6s for pmd with a default input load). Optimization corresponds to the loop transformation to reduce the number of monitored events. 88

3.7 Property `FailSafeIter` - Overheads are expressed in percentages over the original uninstrumented versions (4.8s for bloat and 4.6s for pmd with a default input load). Optimization corresponds to the loop transformation to reduce the number of monitored events. * Indicates that the process was stopped after 172800 seconds and overhead of 3599900. 88

3.8 Percentage of overhead across program-property pairs when running object- and state-based monitors in complete-matching mode. 92

4.1 Sample programs and analysis times. TQ = `TimeQuery`, POP3 = `Gmail POP3`. SC = `Socketchannel`, FR = `Folder`. Times in seconds and Overhead in percentage. 125

5.1 Stutter distance for ϕ_{it} on $hasNext; next$ 135

5.2 Events processed during monitoring 166

5.3 Monitoring time and loops optimized 167

5.4 Effects of scaling on jgrapht-HasNext. Overhead in percentage. 167

Chapter 1

Introduction

1.1 Motivation

With advancements in tools and technologies, modern software systems have grown in size as well as in complexity. At the same time, user expectations about performance and reliability of those systems have also grown. In order to limit development time and also increase reliability, systems are often built using ready-made frameworks, libraries and components. The legal usage of these components is described informally in documentation that is provided with their application programming interface (API). Although useful, the API documentation may contain rules that are ambiguously stated. Moreover, the complexity of the modern components may make their usage rules complex. If used incorrectly, this may lead to a software system that may either crash, produce unexpected results or perform poorly. The bugs introduced by inappropriate API usage may be hard to detect as they are often caused by a sequence of operations spread over an entire program. Moreover, they may cause a failure only under unusual conditions that can easily go undetected during testing.

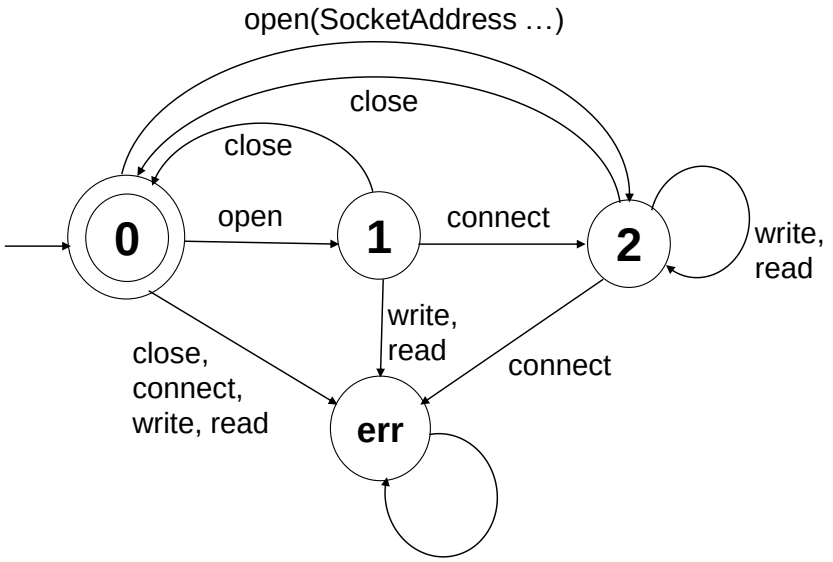


Figure 1.1: Property SocketChannel.

Ensuring that a program uses a component's API correctly, by performing all the sequences of operations related to the component in a legal manner, is a nontrivial problem that is hard to solve just by a code inspection for large software applications. Therefore, in order to address this problem of API conformance checking, researchers have proposed typestate analysis techniques [71]. These analyses try to ensure that a program demonstrates a behavior that is correct with respect to a typestate property. A typestate property describes a legal sequence of operations that can be performed on objects that constitutes a proper usage of the API. These rules can often be expressed as a regular expression and modeled as a finite state automaton (FSA).

A typestate analysis models an instance of a type as evolving over its lifetime through different abstract states. A method call on an instance can maintain the instance in its current state or cause it to move to a new state. If a call moves the instance to the *error* state then that method call can be regarded as the final step in a program execution that violates the typestate property. For example, Figure 1.1 shows the property `SocketChannel`

modeled as a FSA. A Java program that attempts to read from a `java.net.Socket` instance when it is not in its *connected* state, which is state 2, is an error, as is one which fails to eventually return the socket to its *closed* state, which is state 0. A fault in a program does not necessarily result in a failure. However, an early fault detection would be a key to the prevention of a future failure. Since a typestate property behavior is associated with the path taken during a program execution, a typestate analysis provides an effective tool to understand the root causes of a fault. Hence, reasoning about typestate properties offers the potential to increase the observability of program behavior with respect to complex API usage scenarios, thereby offering the potential to detect errors more easily and closer to the fault.

Ideally, software systems would be checked automatically during compilation-time for compliance with a typestate property. In practice, static analyses often produce numerous *false positives*, in which case a program would be identified as erroneous even though it may exhibit legally correct behavior during runtime [23, 24]. On the other hand, if static analyses are made extremely precise, they may not scale to real programs [23, 24, 31]. Scaling to large software systems and providing precise analysis results in the presence of aliasing, data, thread and context-sensitive program behavior is very difficult [31]. In recent years, researchers have combined multiple techniques to dramatically improve the cost-effectiveness of static typestate analyses [10, 11, 12, 29, 40].

For example, Fink et al. present [40] a multi-stage static typestate analysis, in which each subsequent stage of analysis is more precise and expensive than the previous. However, each stage eliminates several points of potential failure (PPF) before starting the next stage, which reduces burden on the next stage. On the other hand, Bierhoff et al. present an intraprocedural analysis [11, 12], that eliminates the need for expensive whole-program analysis. Instead, the authors make use of method annotations that encode access permissions that provide information about typestate changes and also reason about aliasing.

Although impressive in many ways, these approaches still may produce false positives. For example, Bierhoff et al. report that two out of the 12 classes analyzed produced five false positives [12], whereas Fink et al. could verify 93% of PPFs after the final stage of analysis leaving 7% (more than 340) for further analysis or manual inspection [40]. It would take a nontrivial manual effort to ensure that none of the remaining warnings could lead to an error during the program execution.

As a result, dynamic analysis or runtime monitoring of programs to ensure legally correct program behavior with respect to typestate properties has gained considerable attention in this decade [2, 5, 25, 26, 27, 33, 36, 62]. Runtime monitoring tools typically work by instrumenting a program to be monitored with some extra code that generates *events*. These events allow the associated monitors to keep track of typestates. Researchers have proposed several tools and techniques to ensure that monitoring can be efficiently performed on software applications to verify their typestate property compliance. In spite of this effort, monitoring may occasionally incur heavy runtime overhead in practice, as high as 11000% [62] or 3500% [16] which would be unacceptable in a production or even in a testing environment. This heavy overhead limits the application of runtime monitoring only to a subset of real applications and properties. A high runtime overhead could be a result of handling of the large number of monitors that are created during runtime, or the number of events generated by the inserted instrumentation. In our study, for the program `bloat` from the Dacapo benchmark suite [14] and property `HasNext`, we observed about 160 million events and about 2 million monitors, whereas for the property `FailSafeIter` the observed events were over 80 million and it created more than 4 million monitors. The runtime overhead in the case of multi-object properties such as `FailSafeIter`, that involve more than one object, also depends on the number of monitors associated at a time with receiver objects. In this case, every single event observed results in a transition on every monitor associated with the related set of objects. For these reasons, typestate

properties that involve multiple objects have often been found to be more challenging to monitor.

The fact that the strengths and weaknesses of static and dynamic analyses approaches are somewhat opposite has prompted researchers to develop hybrid approaches that would enjoy the benefits of both worlds. Flanagan [41] describes hybrid type checking problems in which programs are first subjected to static checking and then, if property conformance cannot be assured, dynamic checks are employed. A common example involves array bounds checking for languages such as Java in which static techniques have been developed to eliminate the checks [22, 63] that would otherwise be deployed at run-time by the virtual machine. The research on such approaches was mostly limited to *state property* checking, i.e, properties that describe legal program control and data states. Typestate properties are an instance of *path properties*, i.e, properties that describe the legal sequences of program control and data states. However, in recent years a few researchers have tried to apply this approach to path properties. This includes works of Bodden et al. [16, 17, 19, 18, 20, 21], Martin et al. [60] and that of Naeem and Lhoták [66]. Our work [37, 68] is different than these works, as it is guided by the cost models that we have developed and uses approach that exploits the structures of a program and a property. We believe that our work, along with this complementary work done by other researchers, forms a solid foundation for this promising line of research.

1.2 Approach

The primary goal of our research is to develop optimization techniques that reduce the run-time overhead of monitoring without compromising the soundness or the completeness of analysis results. We have developed techniques that identify statements in a program for which instrumentation can be dropped either permanently [37] or during certain parts of an

execution [68]. Our hybrid approach combines static and dynamic typestate analyses techniques. It also exploits the typestate property structure to identify targets for optimization that make the approach more effective.

Our research is guided by the cost models that we have developed based on our understanding of the existing tools [26, 2, 53]. The approach is also driven by the findings and the observations we have made in previous studies. For example, we observed that only a small subset of events generated by some statements may contribute to an error and hence, skipping the remaining ones does not impact error reporting. We used this observation as a motivation in our previous work [68]. In order to identify such statements, we developed a scalable static typestate analysis, which can efficiently analyze benchmarks of the size of DaCapo [14], that ensured that our technique does not compromise the correctness of runtime monitoring, while making it more efficient. In this dissertation, we present cost models that guide our research and two analysis techniques that primarily target the number of events for monitor optimization. Here, we introduce the cost models and the solutions with the help of a small example that highlights the hybrid features of our approach.

1.3 Simple Code and Property Example

Suppose that we are monitoring an iterator property `FailSafeIter` that says that a collection may not be updated while it is being iterated. In other words, it specifies that there may not be a call to any method that updates a collection after an iterator is created and before an element is accessed by a call to method `next`. This is a safety property that ensures that the program behavior is well-defined while iterating over a collection. The property can be specified as a regular expression `(create;next*;update*)`, and can be modeled as a FSA as shown in Figure 1.2.

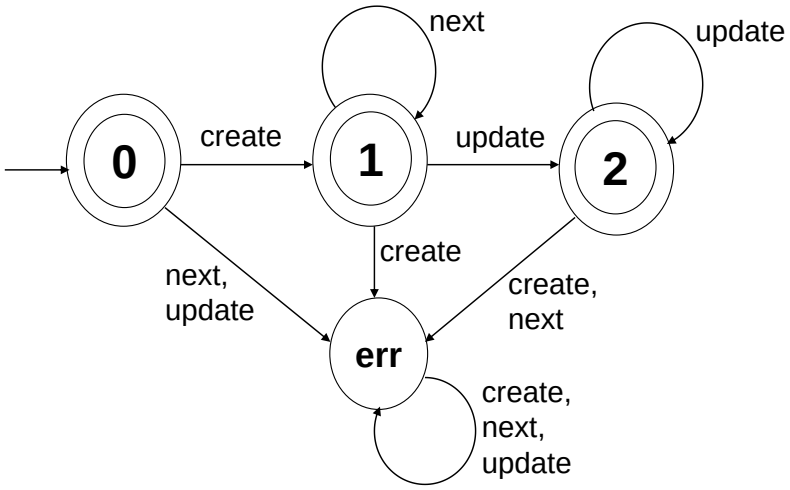


Figure 1.2: Property FailSafeIter.

Figure 1.3 shows a snippet of code that is a modified version of a method from `DaCapo` benchmark `bloat`. Let us assume that we want to monitor a program that contains this method for the property `FailSafeIterator`. A runtime monitoring scheme would then instrument the statements in the code related to the property with code that would keep track of monitor states. We call such statements as *observable statements* or simply *observables*. The observable statements in this snippet are statements 1 and 2.

Figure 1.4 shows the instrumented code that consists of calls to methods from class `OBSERVE` associated with the observable statements. Each one of these calls would constitute an event during runtime. An event would trigger an execution of code that is necessary to track monitor states shown in Figure 1.2.

```
void makeEquiv(Node node1, Node node2) {
  Set s1 = equivalent(node1);
  Set s2 = equivalent(node2);

  if (s1 != s2) {

    Iterator iter = s2.iterator();    // Statement 1

    while (iter.hasNext()) {
      Node n = (Node) iter.next();    // Statement 2
      equiv.put(n, s1);
    }
  }
}
```

Figure 1.3: A code snippet based on a method from `DaCapo` benchmark `bloat`.

1.3.1 Cost Models

Monitoring tools perform several operations that are required to create, maintain and track monitors. For example, in the example shown in Figure1.4, a monitoring tool would create a monitor after receiving a `create` event corresponding to the method call `iterator` on collection referenced by $s2$. A monitor essentially consists of references to the related objects and an instance of FSA to keep track of its state. Some implementations may only keep a local copy of the current state and share a global FSA, however these choices are implementation-dependent as well as driven by the tool architecture. For every event `next` in Figure1.4, a monitoring tool would retrieve a list of associated monitors and track them.

This small example shows that a monitoring tool needs to perform several tasks that include creating monitors, maintaining their pools, locating the associated monitors, traversing the list of retrieved monitors and then executing a transition on each one of them. All these operations incur a cost that depends on the number of events incurred, the number of monitors created, the tool architecture and so on. The cost also depends on the tracking history that dictates the number of monitors associated with the objects. This number in-

```
void makeEquiv(Node node1, Node node2) {
  Set s1 = equivalent(node1);
  Set s2 = equivalent(node2);

  if (s1 != s2) {

    Iterator iter = s2.iterator();    // Statement 1
    OBSERVE.create(s2, iter);

    while (iter.hasNext()) {
      Node n = (Node) iter.next();    // Statement 2
      OBSERVE.next(iter);
      equiv.put(n, s1);
    }
  }
}
...
public class OBSERVE {
     ...
     void create(Collection c, Iterator i){
          ...
     }
     void update(Collection c){
          ...
     }
     void next(Iterator i){
          ...
     }
}
```

Figure 1.4: The instrumented code snippet for the property `FailSafeIter`.

fluences the cost of traversing a monitor list and performing transitions. Similarly the cost also depends on the efficiency of the data structures used, for example, to provide access to the monitors, and on the execution environment that includes the virtual machine (VM). The effectiveness of some optimizations may depend on how frequently a garbage collector is invoked by the VM.

We identified to high-level algorithmic approaches to tool building, namely, *object-based* or *state-based*. These and are explained in Chapter 3. The cost of handling an event

by a monitoring tool may differ considerably depending on the tool-building approach, the type of the event and even the states of the associated monitors. We developed detailed cost models to understand the operational costs of monitoring. We developed separate cost models for object-based and state-based tools due to their fundamentally different ways of handling events. The cost models highlight the key factors that influence the cost of monitoring for each approach and express the relationship among the factors. These factors form optimization targets for our techniques. The analyses that we have developed, including those that we plan to develop in future, are guided by the cost models.

In the following sections, we first introduce the residual typestate analysis that targets the number of events by identifying program statements for which instrumentation can be dropped permanently. It is presented in detail in Chapter 4. In the following section, we explain it using the example from Figure 1.3. Later in this chapter, we introduce the stutter-equivalent loop transformation that also targets the number of events , however, by transforming program loops that are major source of events. It is presented in detail in Chapter 5. The optimization target for both of the analyses is based on the insights provided by the cost models. The cost models define the cost of handling each event and the total cost of monitoring as the sum of costs of handling all events. We prioritize our research by targeting the number of events first, as targeting events indirectly targets all operational costs.

1.3.2 Residual Typestate Analysis

As mentioned earlier, the statements 1 and 2 in Figure 1.4 create events when monitoring the property `FailSafeIter`. Statement 1 would create a monitor corresponding to the set object referenced by $s2$ and the iterator object referenced by $iter$. Let us assume that

the top-down whole-program static analysis that we perform on this code concludes the following:

1. The iterator object was freshly created just before the loop. Hence at the entry of the loop, the property monitor can only be in state 2.

2. The reference variable `iter` is assigned only once at statement 1.

3. The method `next` has no side-effect on the iterator object referenced by `iter` in the context of the property other than executing call `next`.

4. The statement `equiv.put(n, s1)` does not have any side-effects that are related to the collection object referenced by $s2$ or the iterator object referenced by `iter`. In other words, the method call does not result in the generation of events on objects referenced by `s2` or `iter`.

5. The cumulative effect of all possible executions of the `while` statement including the one that does not enter the loop, on the property monitor automaton would be to keep it in state 2 at the end of the method.

It may be easy in this case to see that the conclusions are valid by manually inspecting the program. However, it may be almost impossible to verify this for large complex programs especially in the presence of complex aliasing of references. The final conclusion about the property monitor state which is based on the previous four, is the one that is important from the point of view of instrumentation. It tells us that the program region enclosed by the loop that contains the event generating statement does not change the state of the monitor. Thus it need not be monitored. Hence, we call it a *safe* region. The objective of *residual analysis* is to find as many safe regions as possible in the code that may not contribute to an error.

```
void makeEquiv(Node node1, Node node2) {
  Set s1 = equivalent(node1);
  Set s2 = equivalent(node2);

  if (s1 != s2) {

     Iterator iter = s2.iterator();
     OBSERVE.create(s2, iter);

     while (iter.hasNext()) {
         Node n = (Node) iter.next();
         equiv.put(n, s1);
     }
  }
}
```

Figure 1.5: The output of residual analysis.

The output of residual analysis is shown in Figure 1.5. Note that only the iterator creation statement is instrumented by adding a method call `OBSERVE.create(s2, iter)`. The call to method `next` is not monitored.

Note that the analysis does not try to analyze path conditions. However, it could still conclude that the region is safe only because it precisely calculates reachable states at every statement. In the absence of this information, the analysis would have considered all states as reaching the region and then would have failed to infer the exact exit state. For example, for 3 as entry state, the exit state can be the *error* state if the loop is executed or it could be state 3, if the loop is not entered. This would have made the behavior of the loop *nondeterministic* with respect to the analysis, and as a result, the analysis would have retained the program instrumentation. Also note, that if the static analysis had indicated possible side-effects for the statement `equiv.put(n, s1)`, the residual analysis would not have identified the loop region as safe and would have retained the instrumentation.

This shows that imprecision in the static analysis may result in the analysis not being able to drop instrumentation inside a loop. The impact of this failure could be big, as it may

get executed during every loop iteration creating large number of events. Our second analysis, introduced in the following section, tries to overcome this deficiency by transforming stutter-equivalent loops. This analysis targets statements inside loops that may contribute to an error only during certain parts of executions.

1.3.3 Stutter-equivalent Loop Transformation

Programs spend most of their time executing loops. Hence, a statement inside a loop may generate an event in every iteration of the loop. However, we have observed that often only the events encountered during the first few iterations are able to change the monitor state. The remaining iterations only *stutter* with respect to the property , i.e. they do not change the state of the monitor. Hence, the analysis restricts its scope to program loops that contain statements generating events and checks if these loops are eligible for transformation.

For the example shown in Figure 1.3, the analysis does not try to reason about feasible monitor states at the entry of a loop. Instead, it assumes that the monitor could be in any state. Nevertheless, it performs intraprocedural *points-to* analysis using the method summaries to ensure necessary object aliasing conditions and the absence of side-effects. It concludes that no matter what state the property monitor is in at the entry of the loop, the state at the exit of the loop would be correctly determined if we observe the loop only during its first iteration assuming that the loop was executed at least once. For example, if the monitor was in state 1 at the entry of the loop, it would be in state err at the end. If it was in state 2 at the entry of the loop, it would be in state 2 at the end, and if it was in state 3, it would be in state err at the end. The events can be skipped for the remaining iterations as they do not change the monitor state, after the first iteration. Such a loop can be transformed as shown in Figure 1.6. For simplicity, some details are not shown in this figure, but are presented in Chapter 5.

```
void makeEquiv(Node node1,
               Node node2) {
  Set s1 = equivalent(node1);
  Set s2 = equivalent(node2);

  if (s1 != s2) {

    Iterator iter =
             s2.iterator();
    OBSERVE.create(s2, iter);

    while (iter.hasNext()) {
      Node n =
         (Node) iter.next();
      OBSERVE.next(iter);
      equiv.put(n, s1);
    }
  }
}
```

```
void makeEquiv(Node node1,
               Node node2) {
  Set s1 = equivalent(node1);
  Set s2 = equivalent(node2);

  if (s1 != s2) {
     Iterator iter =
              s2.iterator();
     OBSERVE.monitor(s2, iter);

     while (iter.hasNext()) {
         Node n =
             (Node) iter.next();
         OBSERVE.next(iter);
         equiv.put(n, s1);
         break inner;
     }
     while (iter.hasNext()) {
         Node n =
             (Node) iter.next();
         equiv.put(n, s1);
     }
  }
}
```

Figure 1.6: Stutter-equivalent Loop Transformation Example: original (left) and transformed (right).

The analysis creates a pair of loops and instruments only the statement in the first loop with a method call `OBSERVE.next(iter)`. The instrumentation on the iterator creation statement remains unchanged as the analysis restricts its scope only to loops. The *break* statement in the first loop forces the control to jump to the second loop after the instrumented code in the first loop is executed. It should be noted that the transformed pair of loops preserves the semantics of original loop, the only difference being in the events observed by the monitoring system. If the loop in the original program is executed 1000 times during some execution, it will generate 1000 events corresponding to the method call `next` that will be monitored. However, the one in the transformed example would generate only

one event during the single iteration that the predecessor loop will undergo. The successor loop will undergo 999 iterations without generating any events. The rest of the program behavior that is unrelated to monitoring remains unchanged, which shows that the transformation preserves program semantics. Our study [68] showed that this optimization can significantly reduce the number of observed events that may reduce the runtime overhead of monitoring.

The effectiveness of this analysis is enhanced by the fact that it targets only the observables inside loops. The observables that occur before or after the loop execution are irrelevant. Hence, any imprecision in the analysis that makes it unable to reason about those observables does not reduce the ability of the analysis to identify loops eligible for transformation. This also results in higher scalability.

1.4 Theses

The dissertation makes the following theses.

1. The cost models explain the overhead incurred by runtime monitoring because (i) they make the key factors that influence the cost of monitoring as well as the relationships among them explicit, (ii) they are based on the understanding of the existing runtime monitoring tools, and (iii) they have been partially validated.
2. The optimization techniques that we present in this thesis effectively reduce the monitoring overhead without compromising error reporting, because (i) they are based on analyses that perform sound approximations, and (ii) they target the number of events, a major source of overhead based on the insights provided by the cost models.

1.5 Contributions

The brief overview provided in the previous sections of the analyses that we have developed shows our hybrid approach that integrates static typestate program analysis with runtime monitoring. At the same time, it also exploits the structure of a typestate property to make monitor optimization more effective. For example, in the analysis that transforms stutter-equivalent loops, we exploit the looping structures inside a program as well as within a property FSA. In the residual typestate analysis, we safely approximate the behavior of program paths with respect to a property by dropping some instrumentation points or by replacing some of those by reduced set of points using summary transitions based on the property structure. Both analyses preserve monitor correctness and program semantics.

Previous static approaches [10, 11, 12, 40] ignore the structure of a typestate property, whereas the hybrid approaches [17, 19, 18, 20, 21, 29, 66] analyze a given program but perform no or only a limited property analysis such as checking the existence of events required to reach the error or the matching state. Many of the dynamic approaches [2, 5, 27] consider neither the program structure nor the property structure in their analysis. However, Chen et al. [26] perform optimizations based on the property structure that reduce the number of unnecessary monitors. Avgustinov et al. [8] analyze the property structure and present two techniques for monitor optimization in the context of Tracematches.

The novelty of our research approach lies not only in its integrative nature, but also in the fact that it is based on the insights provided by the cost models that describe the factors influencing monitoring overhead. In other words, the cost models lay a foundation for the optimization techniques that we have developed and also for the ones that we plan to develop in future. In addition, the techniques that we have developed are complementary to one another.

The main contributions of our research are summarized as follows.

1. We describe cost models for runtime monitoring that are based on our understanding of current monitoring tools. The cost models highlight the key factors that are responsible for monitoring overhead and that should form targets for monitor optimization.

2. We present two typestate analyses, namely, the residual typestate analysis and stutter-equivalent loop transformation that we have developed using a hybrid approach that combines static and dynamic typestate analyses. In addition, the approach exploits the program and the typestate property structure. Based on the insights provided by the cost models, the analyses primarily target the number of events, but can also be extended to target the number of monitors. We present the results of their evaluation based on some open source and DaCapo benchmarks, that show that the techniques can effectively reduce the runtime overhead. In addition, we also show that the analyses produce sound and complete results except in the presence of unchecked exceptions if certain conditions are satisfied. We describe the conditions in Chapters 4 and 5.

3. We provide directions for future research, that include a novel technique for monitor optimization as well as an improvement over existing techniques. The optimization technique that we propose aims to reduce the cost of monitor traversal as well as the cost of performing transitions, by reclaiming unnecessary monitors. We also propose a monitoring approach that we call symbol-driven monitoring and describe an optimization based on it.

1.6 Outline of Dissertation

The primary focus of the research is on developing techniques to reduce the overhead of runtime monitoring. Hence, in this dissertation, we first provide a general background for runtime monitoring including its generic architecture in Chapter 2. In the same chapter, we present the related work. The cost models that we have developed, provide insights into current runtime monitoring approaches that help us prioritize our efforts to develop new analyses. We present the cost models in Chapter 3. In the same chapter, we revisit some published results and explain those using a new perspective provided by the models. Based on these new insights, we have developed two techniques that target the number of events for optimization. Chapters 4 and 5 present the two analyses, the residual analysis and the analysis for loop transformation respectively, along with the results of their evaluation. Finally, in Chapter 6, we provide the conclusion and future research directions.

Chapter 2

Background and Related Work

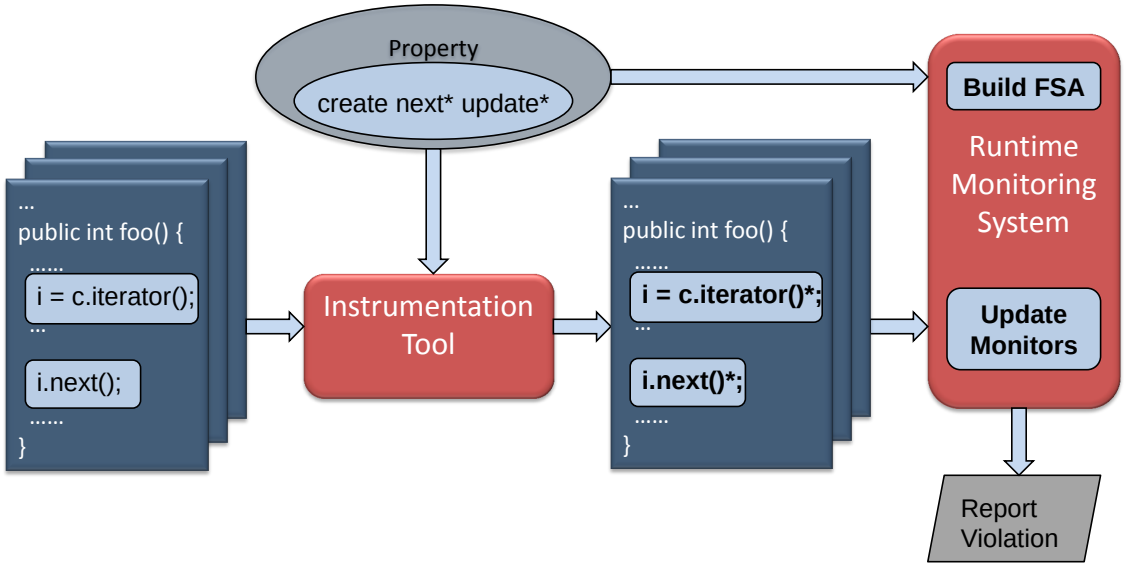


Figure 2.1: General Monitoring Scheme

2.1 A General Monitoring Scheme

Runtime monitoring is performed to analyze program behavior with respect to a property during program execution. A property can be specified using various formalisms [34] such as regular expressions. For example, the property `FailSafeIter` can be expressed as a regular expression `create next* update*` as shown in a general monitoring scheme in Figure 2.1. Equivalently, it can also be represented as a FSA as shown in Figure 1.2. A monitoring system or tool typically builds a FSA model for any property specified as a regular expression, as it is convenient to track a finite state machine. A modeled property FSA could be deterministic or nondeterministic depending on the design choices made while building a monitoring tool. Neither of these choices affect the expressiveness of the tool, as both forms of FSA are equivalent in terms of their expressions, that means the languages that they accept. For run-time monitoring, a common form of path property specification used is a deterministic FSA [49]. In this dissertation, we assume that a property FSA is deterministic unless mentioned otherwise.

The monitoring tool shown in Figure 2.1 accepts a program and a property to be monitored as input. The tool reports an error if and only if it detects it. As a first step, the instrumentation tool, which may also be a part of the monitoring tool, identifies observable statements in the program that are relevant to the property. The instrumentation tool then instruments the observables with some code that updates states of corresponding monitors waiting for that event. For example, Figure 1.4 shows observable statements with respect to the property `FailSafeIter` that are instrumented. After receiving an event, the system retrieves the associated monitors and performs a transition corresponding to the input symbol `next` on all the monitors. Alternatively, it creates a monitor if it does not exist corresponding to the event. During a program execution, if any of the monitors goes to the error state, then a property violation is reported. Otherwise, it does not report a

violation. These steps are explained in more detail in the next section. JavaMOP [26], Tracematches [2], and QVM [5] are examples of state-of-art monitoring tools.

2.2 Monitoring Internals

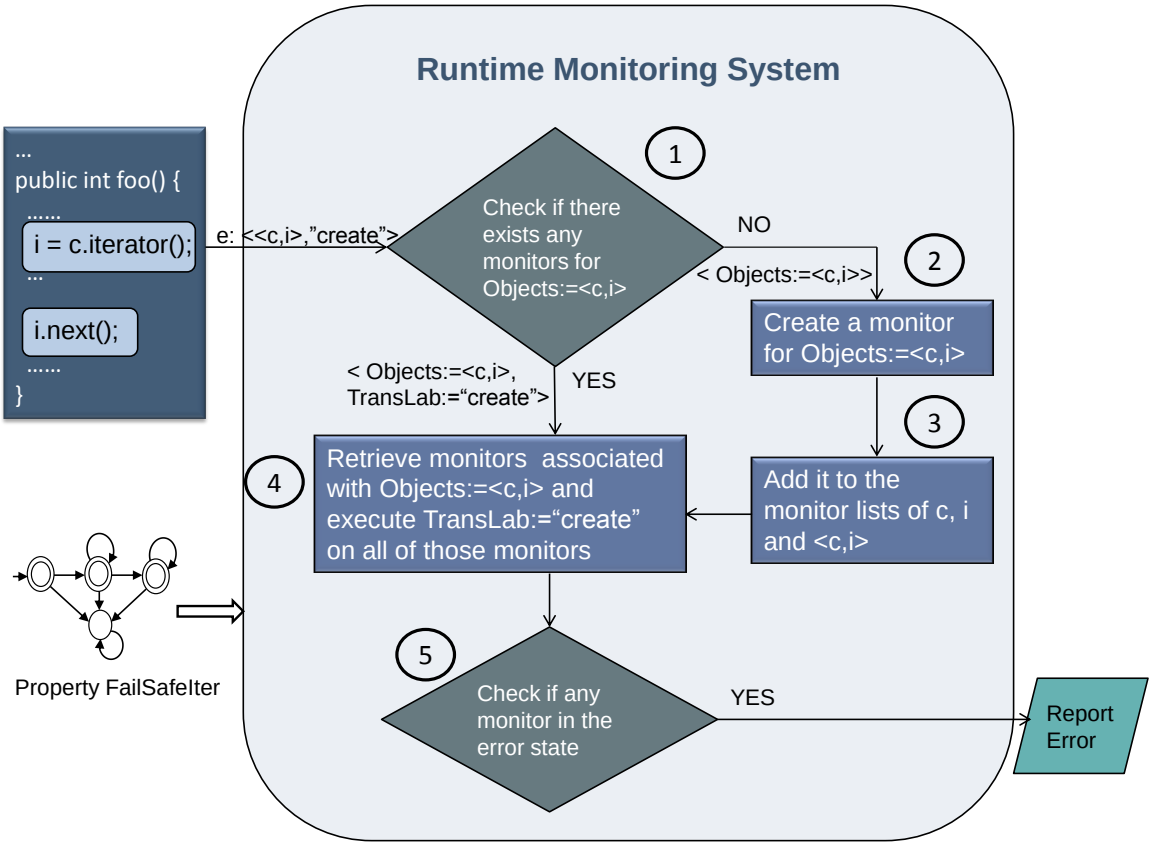


Figure 2.2: Monitoring Internals

Figure 2.2 shows a schematic of monitoring internals and the sequence of operations that take place inside a monitoring tool after it receives an event. For the sake of simplicity, only the most relevant details are shown. We consider the same property example that we considered in Section 2.1, and see what happens when the tool receives an event with associated symbol `create`. Note that every event is parameterized with the related set

of objects and the symbol. In this case, the related set of objects consists of the objects referenced by the variables c and i. The operations that happen inside the monitoring tool are as follows.

1. After receiving event `create` that is generated by a call to method `iterator`, the tool first checks if there are any existing monitors associated with the related objects, in this case, both c and i.

2. If not, it creates a monitor associated with the set of objects (in this case, both c and i).

3. It inserts the monitor, in the pools of monitors associated with related set of objects. In this case, there would be three pools associated, one each with the following sets of objects: i) c, ii) i and iii) both c and i. The details of this step would vary considerably depending on the tool-building approach and the implementation.

4. The tool retrieves the list of monitors associated with the related objects and performs a transition related to the symbol `create` on all the monitors in the list. In this case, there would only be one monitor corresponding to c and i, as the tool would create a monitor for every combination of c and i. For event `next`, the tool executes step 4 directly after step 1, as it finds and retrieves the list of monitors corresponding to the object referenced by i, containing the monitor that was created by previous event `create`.

5. Finally, if any of the monitors goes to the error state, a property violation or error is reported.

We explain in detail the costs associated with these operations in Chapter 3.

2.3 Monitoring Fundamentals

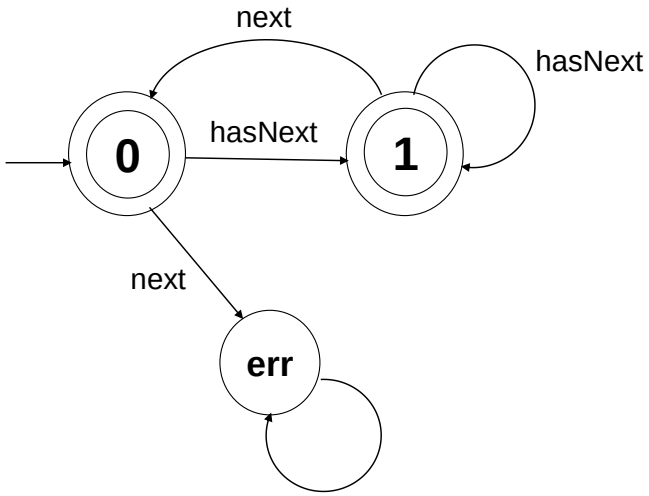


Figure 2.3: Property HasNext for complete matching.

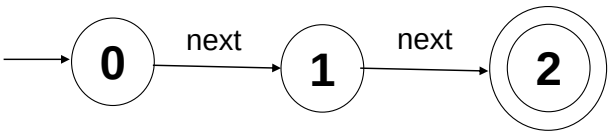


Figure 2.4: Property HasNext for suffix matching.

A typestate property may be specified either for a *complete* matching or for a *suffix* matching. A property specified for a complete matching, typically expresses complete legal behavior of an object or interacting objects with respect to the property. Any illegal behavior that pushes the property FSA to the *error* state, would be reported as an error. On the other hand, a property specified for suffix matching would typically express only a

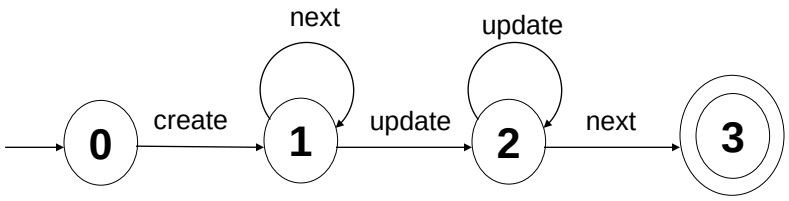


Figure 2.5: Property FailSafeIter for suffix matching.

sequence or sequences of operations that may lead to an error. Hence, a match would correspond to an illegal behavior. This may be explained with the help of an iterator property `HasNext` which says that every call to the method `next` on an iterator object must be preceded by a call to `hasNext`. This can be represented as a regular expression `(hasNext+ next)* hasNext*` for complete matching and modeled as FSA as shown in Figure 2.3. Here, moving a monitor to state `err` would imply an error or illegal behavior. Alteratively, it would be sufficient to specify the suffix that may lead to an error, which can be expressed as `next next` and modeled as FSA as shown in Figure 2.4. Here, moving of a monitor to state 2 would imply a *match*, which in turn, would imply an error or an illegal behavior. Similarly, the property `FailSafeIter` represented by the FSA in Figure 1.2, can be expressed as `create next* update+ next` and modeled as an FSA as shown in Figure 2.5 for suffix matching, where a match would indicate an illegal behavior.

For suffix matching, a monitor may exist simultaneously in several current states. Suffix matching would try to match every symbol from the current monitor states as well as from the beginning. A monitoring tool may support this by keeping a monitor that has multiple current states or by keeping multiple monitors each with a unique current state. This is different from complete matching where a monitor is in exactly one state at a time. A consequence of this is a larger effort in tracking as all monitor states need to be updated for

every symbol encountered. An incurred symbol may destroy a pattern that may result either in the deletion of a current state or in the destruction of a monitor depending on the implementation. For example, suppose that a monitoring tool is monitoring property `HasNext` for suffix matching as shown in Figure 2.4. Suppose that the tool is about to receive event `next` and there exists a monitor corresponding to the iterator object associated with the event that has only one current state 1. As soon as, the tool receives the event it updates state 1 for the event and moves the monitor to state 2 that corresponds to a match. Hence, the tool would report a violation. At the same time, it begins another match starting with the start state which is 0. It updates this current state to 1. Hence, the monitor will now have two current state, namely 1 and 2. Now suppose the tool receives another event `hasNext` associated with the same iterator object. Since there is no transition corresponding to symbol `hasNext` originating from either state 1 or state 2, it will fail a *partially* built pattern at state 1, and restart building a pattern for state 2. In other words, for both current states, the tool will restart building a pattern from state 1. Since the current states are contained in a set, this means that there will now be only one current state. Alternatively, this may result in the destruction of a monitor, if the implementation supports a model with multiple monitors each holding exactly one current state.

The analysis that we consider in this research is based on the concept of typestate, first introduced by Strom and Yemini [71]. A typestate associates a context with an object, that determines legal operations that may be performed on the object in that context. A typestate property specifies a legal sequence of operations on a receiver object that is associated with a correct API usage. For example, as mentioned in the previous section, an iterator property `HasNext` enforces the rule that every call to method `next` should be preceded by a call to method `hasNext`.

A typestate analysis checks if a program complies to a given set of API rules that are expressed as a typestate property. Even though the original concept of typestate was pro-

posed for a single object, many properties of interest involve multiple interacting objects. Hence, the concept is extended for the purpose of program analysis to express the properties that model interactions among multiple objects and specify legal call orderings [21, 66]. In our research, along with the uni-object properties, we also consider these multi-object properties and for convenience, we call them *multi-object typestate* properties.

Regardless of the formalism used for typestate property specification, an *observation* of a program behavior is defined in terms of an execution of an observable statement, such as a method call or return, coupled optionally with a predicate defined over program variables. In practice, an observation typically would be a result of an event generated by the instrumentation code associated with an observable. In our presentation, we abstract away from such details by assuming the definition of a property *alphabet*, Σ, which is a set of symbols that encode observations of program behavior that are relevant to a property.

2.3.1 Complete Matching

For all the properties that can be specified as a regular expression, the corresponding property FSA can be expressed as a tuple $\phi = (S, \Sigma, \delta, s_0, A)$ where: S is a set of states, Σ is the alphabet of symbols, $s_0 \in S$ is the initial state, $A \subseteq S$ are the accepting states and $\delta\colon S \times \Sigma \rightarrow S$ is the state transition function. We use $\Delta\colon S \times \Sigma^+ \rightarrow S$ to define the composite state transition for a sequence of symbols from Σ; we refer to such a sequence as a *trace* and denote it π. We define the *error* state as $\mathrm{err} \in S$ such that $\neg\exists \pi \in \Sigma^* : \Delta(\mathrm{err}, \pi) \in A$. A property defines a *language* $L(\phi) = \{\pi \mid \pi \in \Sigma^* \wedge \Delta(s_0, \pi) \in A\}$; for convenience we overload L so that $L(r)$ denotes the language defined by regular expression r.

Complete matching can be explained using example in Figure 2.6. We assume that code shown in the figure is being monitored for property `HasNext`. The example in Figure 2.6

```
void makeEquiv(Node node1, Node node2) {
  Set s1 = equivalent(node1);
  Set s2 = equivalent(node2);

  if (s1 != s2) {

    Iterator iter = s2.iterator();

    while (iter.hasNext()) { // Statement 1
      Node n = (Node) iter.next();    // Statement 2
      equiv.put(n, s1);
    }
  }
}
```

Figure 2.6: A code snippet based on a method from `DaCapo` benchmark `bloat`.

is basically same as the example in Figure 1.3, the only difference being in the observable statements which in this case are calls to methods `hasNext` and `next`.

In this example, for every object referenced by variable $iter$, a monitoring tool working in complete matching mode, would create a monitor for the first occurrence of a call to any of the two methods, `hasNext` and `next`. It would then keep track of the monitor states as they are shown in the FSA in Figure 2.3. A simple monitoring algorithm for complete matching is presented in Figure 2.7.

Lines 4–7 correspond to the step that checks if there are any monitors associated with the set objects l related to an event e. If there are not any, the tool will create a monitor and add it to the pool of monitors. Lines 8–12 show the next step that retrieves the list of monitors associated with l and perform a transition corresponding to the symbol b on each one of them. If any of the monitor moves to the error state, an error will be reported.

```
CompleteMatching(φ = (S, Σ, δ, s0, A), e = (l, b))
 1 let L be the set of sets of objects that receive events
 2 let M be the set of monitors
 3 let LM : L → M be a map
 4 if LM(l) = null then
 5    m ← new monitor(l)
 6    m.cur ← s0
 7    M ← M ∪ m
 8 ms ← LM(l)
 9 for m ∈ ms do
10    m.cur ← δ(m.cur, b)
11    if m.cur = err then
12       report error
```

Figure 2.7: Simple complete matching algorithm.

2.3.2 Suffix Matching

In the case of suffix matching, a runtime monitor may simultaneously be in multiple states. For all those properties that can be specified as a regular expression, the corresponding property FSA can be expressed as a tuple $\phi = (S, \Sigma, \delta, s_0, A)$ as in Section 2.3.1. We define a state transition function for a set of states $\delta' \colon 2^S \times \Sigma \rightarrow 2^S$ as $\delta'(S', b) = \{s | (\exists s' \in S' : \delta(s', b) = s) \vee (\delta(s_0, b) = s)\}$. Similarly, we have a composite state transition $\Delta' \colon 2^S \times \Sigma^+ \rightarrow 2^S$ for a trace π.

Note that, in the case of suffix matching, a monitor in an accept state would indicate an error. Hence, we define a *matching* state as any state s such that $s \in A$. A property defines a language $L(\phi) = \{\pi \mid \pi \in \Sigma^* \wedge ((\Delta'(\{s_0\}, \pi) \cap A) = \emptyset)\}$. In other words, a property ϕ defines a language L, which is a set of traces, none of which may generate a match. For convenience, we overload L so that $L(r)$ denotes the language defined by regular expression r.

Suffix matching can be explained using the same example in Figure 2.6. For every object referenced by variable $iter$, a monitoring tool working in the suffix matching mode, would create a monitor for the first occurrence of a call to any of the two method calls,

```
CompleteMatching(φ = (S, Σ, δ, s0, A), e = (l, b))
 1 let L be the set of sets of objects that receive events
 2 let M be the set of monitors
 3 let LM : L → M be a map
 4 if LM(l) = null then
 5    m ← new monitor(l)
 6    m.cur ← {s0}
 7    M ← M ∪ m
 8 ms ← LM(l)
 9 for m ∈ ms do
10    m.cur ← m.cur ∪ {s0}
11    for s' ∈ m.cur do
12       s' ← δ(s', b)
13       if s' ∈ A then
14          report error
```

Figure 2.8: Simple suffix matching algorithm.

`hasNext` and `next`. This step is very similar to the corresponding step in complete matching. The only difference being that the field cur, instead of keeping a current state, would keep a set of current states as in suffix matching may exist in multiple states at the same time. It would then keep track of the monitor states as they are shown in FSA in Figure 2.4. A simple monitoring algorithm for suffix matching is presented in Figure 2.8.

Lines 4–7, as in the previous section, correspond to the step that checks if there are any monitors associated with the set of objects l related to an event e. If there are not any, the tool will create a monitor and add it to the pool of monitors. Lines 8–14 show the next step that retrieves the list of monitors associated with l and perform a transition corresponding to the symbol b on each one of its current states. If any of the monitor moves to an accept state which is a matching state, an error will be reported.

Both algorithms, suffix matching and complete matching, have been simplified by ignoring many details including underlying monitoring tool approach. They do not consider the data structures used for monitor organization. Adding these details would change the presentation of algorithms considerably. However, the purpose of the presentation of these

algorithms here is to make the fundamental difference between complete matching and suffix matching clear. These algorithms highlight the fact that for suffix matching, a monitor may exist in multiple current states at a time and the match is always started from the beginning, that is from the start state, along with all other current states. This is in contrast with complete matching, where the monitor can only be in one current state at a time. The suffix matching algorithm assumes that a monitor keeps a set of current states. However, in practice, a monitoring tool may maintain separate monitors with exactly one current state, instead of one monitor with several current states. Such issues are implementation-dependent, and are discussed in more detail in Section 3.2.3.

2.4 Property violation and Monitor Correctness

FSA monitoring involves detection of each occurrence of an observation, $b \in \Sigma$. For complete matching, a *simple* runtime monitor stores the current state, $s_c \in S$, which is initially s_0, and at each occurrence of an observation b, it updates the state to $s_c = \delta(s_c, b)$ to track the progress of the FSA in recognizing the trace of the program execution. We say that a program execution *violates* a property, ϕ, if the generated trace, π, ends in a non-accepting state, i.e., $\Delta(s_0, \pi) \notin A$; violations can be detected as soon as the monitor enters the error state, i.e., $s_c = err$.

For suffix matching, a simple runtime monitor stores the set S_c of current states, where $S_c \subseteq S$, which is initially $\{s_0\}$, and at each occurrence of an observation b, it updates the state to $S_c = \delta'(S_c, b)$ to track the progress of the FSA in recognizing the trace of the program execution. We say that a program execution *violates* a property, ϕ, if some prefix $\widehat{\pi}$ of the generated trace, π, may end in a matching state, i.e., $\Delta'(\{s_0\}, \widehat{\pi}) \cap A \neq \emptyset$.

A runtime monitor is correct with respect to the property ϕ, if it is sound as well as complete with respect to ϕ. The definitions of soundness and completeness are given below.

Definition 2.4.1 (Monitor Correctness) *A runtime monitor for property ϕ observing execution trace π is* **sound** *if it reports a violation if $\pi \notin L(\phi)$, and* **complete** *if it reports a violation only if $\pi \notin L(\phi)$.*

Soundness guarantees that no observed violation will be missed, whereas completeness guarantees that false reports of violations will not occur.

Both the analyses that we have developed and presented in this proposal are sound and complete in the absence of unchecked exceptions. In the presence of unchecked exceptions, the static analysis may not safely approximate the program behavior as the unchecked exceptions do not appear in method signatures. Hence, the analysis may produce incorrect results in that case if certain conditions are met. These conditions and the monitor correctness are discussed in detail in Sections 4.3 and 5.6.

2.5 Related Work

There has been a huge body of research work on software verification in general. However, a part of it is about checking programs for state properties, which are typically specified as some predicates about the data states of the program at program points of interest. Our research is about verifying typestate properties which are path properties that reason about a sequence of program's control and data states. In this section, we will present work that is related to the verification of path properties.

2.5.1 Static Approaches

2.5.1.1 Type-system based approaches

Bierhoff and Aldrich [11] present a type-system based approach for typestate verification. Their analysis is completely intra-procedural which eliminates a need for expensive whole-

program analysis and makes it scalable. The authors make use of method annotations in the form of access permissions that specify typestate changes and also reason about aliasing. These annotations that represent API usage rules should be supplied by the user. In order to specify access rules the authors make use of linear logic. The typestate changes can be enforced by using the temporal logic rules and types of accesses it allows on the objects. Authors provide the flexibility, which is necessary to support aliasing, by providing rules for splitting and merging access permissions. A downside of this approach is that, like all other static approaches, it may still produce false positives. However, it is sound and does not produce false negatives.

Bierhoff et al. present a tool `Plural` based on this approach and evaluate it using a few applications [12]. The authors mention the *price* of modularity as the effort in developing annotations. For one application, the authors produced about 15 annotations in 75 minutes, whereas for the other they developed about one annotation for every 30 lines of code. This indicates that an effort for large applications can be prohibitive. For an application with 65 methods, the tool reported a total of nine warnings out of which five were false positives. This data shows that, although impressive, the approach still may produce false positives. Compared to this, a typical static analysis approach or a static analysis phase of a hybrid analysis approach may produce more false positives for which a hybrid approach performs runtime monitoring. However, these approaches do not require any method annotations, which would be an additional burden on a programmer. Moreover, unless automated, the process of developing annotations, may be prone to errors.

Beckman et al. have extended Bierhoff and Aldrich's approach to check correct usage of atomic blocks to avoid race condition [10]. The Java-like language presented in the paper is based on a sound type system. However, the evaluation of their prototype is based on a small application. Hence, whether the approach would scale to large applications in practice remains unclear.

DeLine and Fähndrich's type-system based approach [29] is similar and like other type-system based approaches it also suffers from a laborious and an error-prone process of developing annotations. The annotations express the typestate change and aliasing rules over objects as method pre- and post-conditions. The authors have implemented their solution in a tool named Fugue and is used to analyze Microsoft .NET-based programs.

2.5.1.2 Static analysis based approaches

Fink et al. present a multi-staged static analysis that is flow-sensitive as well as context-sensitive [40]. In order to make the approach scalable, intraprocedural and flow-insensitive analyses are run in the beginning. These analysis are less precise but efficient, reducing the workload for the later stages. The later stages use pointer analysis techniques capable of making *strong updates* that are essential for checking typestate properties. Strong updates overwrite the old content of an abstract memory location with a new value [30]. A disadvantage of this approach is that unlike previous approaches it is non-modular and hence more expensive. However, it needs no annotations. The evaluation of the framework by the authors showed that staging improved performance of their framework from verification of 30% points of potential failure (PPFs) to 93% PPFs. However, the approach could not verify correctness for about 7% (more than 340) PPFs. Manual inspection of 340 PPFs would be a time-consuming effort.

Das et al. [28] present ESP, that can be viewed as an analysis tool for typestate-like verification which works on a technique that the authors call *property simulation*. Property simulation limits the dataflow information by not tracking the branches that may not affect the property states. This way the authors succeed in scaling up the analysis to verify an application of the size of `gcc` [44] for a property that checks if a file is written only when it is opened. In comparison, our residual typestate analysis is path-insensitive for the purpose of scalability. It safely approximates the paths by merging property states at the join points.

It might be interesting to implement property simulation approach to see if an increase in the precision results in higher benefits without compromising with the scalability. ESP's inter-procedural analysis has a context-sensitive approach that is similar to residual typestate analysis's approach. Both use method summaries at the call sites and merge those at the entry and the exit points appropriately. Both approaches would reanalyze a called method, only if the summary information is inadequate, i.e. it does not have an information about a relevant property state. It would be interesting to see if ESP can scale to more complex properties that involve multiple objects and for languages like Java that make heavy use of dynamic dispatching.

2.5.2 Dynamic Approaches

Static approaches for typestate checking are useful. However, they still produce false positives. This has given rise to research on novel tools and techniques for runtime monitoring.

2.5.2.1 Static instrumentation based approaches

Allan et al. present Tracematches [2], a tool that can monitor a typestate property using its specification language *tracematches* which is a pattern-based specification language and is an extension of AspectJ [51]. Tracematches performs only suffix matching and the properties may be expressed as regular expressions. Tracematches specifications may include a code that is to be run when a pattern is matched. This code is woven directly into the base code. Tracematches implementation is based on AspectJ compiler `abc` [7] and Soot [70]. Its evaluation using a moderate-size application showed that it incurred a heavy overhead of over 850%. However, it did not incur significant memory overhead. Avgustinov et al. [8] propose and implement two optimizations, one of which reduces memory leak making memory usage as well as tracking efficient by eliminating unnecessary monitors. The other

optimization provides indexing for efficient access to monitors that further improve tracking efficiency. The evaluation on a few benchmark-property combinations shows dramatic improvement in the performance on many but not all combinations. Occasionally, the optimizations result in similar or even worse performance. As pointed by the authors, the dependency on a garbage collector that optimizes memory leaks means that programs that have larger heap available may perform poorly compared to programs that have smaller heap available, because the garbage collector may be invoked less frequently in the former case. The evaluation performed by other researchers [62, 61] using DaCapo benchmarks [14] shows that Tracematches occasionally may incur a very high overhead in excess of 11000%. In Chapter 3, we analyze this overhead using the cost models and identify the contributing factors.

JavaMOP [25, 62] generates aspects based on a specification provided by a user for Java that can be woven into a program using any AspectJ compiler. JavaMOP is more expressive as far as property specification is concerned. It allows programmers to specify properties using various formalisms including context free grammars, FSAs, extended regular expressions and PTLTL. JavaMOP provides sophisticated indexing that can be either centralized in which indexing trees are shared by all objects or decentralized in which monitor references are stored by the objects as an additional field. Decentralized indexing normally provides faster access to monitors compared to centralized indexing, but also requires additional instrumentation for inserting a new field which may restrict its usage to testing environment. JavaMOP has been evaluated [61, 62] on a wide variety of properties using DaCapo benchmarks [14] and its performance has been compared with Tracematches [2] and PQL [60]. PQL is a query language that allows users to check property conformance. The tool with this specification language also performs static analysis to that filters unnecessary events and hence it is discussed in Section 2.5.3. The results show that JavaMOP outperforms Tracematches and PQL for most of the benchmarks used. The

results also show that JavaMOP could efficiently monitor most of the program-property combinations for DaCapo benchmarks. However, JavaMOP incurs 1112% overhead for `bloat-HasNext` combination, whereas it incurs 627% for `bloat-FailSafeIter` combination [61]. Our work on the stutter-equivalent loop transformation is motivated by the high overhead that is incurred by some of these benchmark-property combinations. Hence, we used the same program-property combinations that were found to be challenging for monitoring in the previous work [26, 61] and used JavaMOP as the monitoring tool for the evaluation of our technique.

2.5.2.2 Dynamic instrumentation based approaches

Dwyer et al. present optimization techniques [33, 36] that exploit the structure of typestate properties. The authors present a framework, *sofya*, [36, 53] for adaptive runtime monitoring. This framework can enable or disable instrumentation for obervables by performing dynamic rewriting. The observables that are selected for the optimization are the ones that only self-loop on a state of a property FSA, i.e. they do not change the state, when the FSA is in that state. Formally, for an entry state s, the technique calculates the set O of observable symbols, o, for which $\delta(s, o) = s$. Hence, once the FSA is moves to the state s, the framework disables the instrumentation for all the observables in O, provided no other monitors are interested in that observable symbol. It re-enables the instrumentation for those observables after one of the monitors moves to another state for which the condition is no longer valid. The framework evaluation on a few applications shows reduction in the runtime overhead that makes their monitoring feasible. However, dynamic rewriting of programs to enable and disable instrumentation and the infrastructure that the framework needs, makes efficient monitoring of larger programs challenging. The technique can be seen to be partially addressing the problem that stutter-equivalent loop transformation is addressing. An advantage of our loop transformation technique is that it is static. No

dynamic checks are required to see if instrumentation can be dropped. Moreover, it can handle loops that have more complex loop phrases that transition through multiple states. For example, in case of property `HasNext`, the technique can detect eligible loops with the phrase $hasNext_t, next$ when it transitions through two states in every iteration. An advantage of adaptive online monitoring is that it can handle program fragments that may or may not include loops, or include sequence of loops, provided that these fragments have all self-looping observables. In addition, its condition regarding self-looping behavior is less stringent than loop transformation analysis's condition, where it needs to hold for all states. However, it needs all the monitors to be in self-looping states with respect to the symbol that is to be disabled, which makes the instrumentation disabling condition severe. In general, being a completely static transformation that needs no support during runtime, we expect loop transformation technique to score over adaptive online analysis technique when it comes to performance.

2.5.2.3 Sampling based approaches

Sampling based approaches work by selecting only a subset of certain attributes for observation to decrease the runtime overhead. However, they compromise soundness for this efficiency.

Arnold et al. present *Quality Virtual Machine* (QVM) that has an ability to perform typestate, assertion and heap property checking [5]. The biggest advantage of QVM is that it is implemented in JVM, which gives it direct access to the information that resides in VM. However, this advantage comes with a cost of non-portability. Unique features of QVM include its overhead manager that allows user to specify acceptable monitoring overhead and then ensures that the overhead remains under this limit. At the same time, it extracts as much information as possible by using techniques such as property-guided sampling and object-centric sampling. Property-guided sampling ensures that sampling is

not done randomly and ensures no false positives. An object-centric sampling is performed by marking objects for tracking at their creation times using a bit in the object header. A typestate history, which is a log of sequence of events incurred by a receiver object, is maintained for a few objects created at an allocation site that is identified as creation site for objects violating a property. The evaluation performed using a variety of benchmarks including DaCapo, showed that even for small overhead budget of 10%, QVM could track more than 80% objects for 9 out of 15 benchmarks. This evaluation shows that QVM can effectively monitor large applications. However, the current approach does not consider multi-object typestate properties, in which case, its use may be restricted to checking uni-object typestate properties. The approach is unsound as it cannot detect errors for the objects not selected for monitoring or after they are dropped.

Dwyer et al. present an approach based on subalphabet sampling [33]. The properties being monitored are first composed to form a single large property. This creates a lattice in which the integrated property sits at the top of the lattice. Several program versions are made and then instrumented for the properties sampled systematically by decomposing the lattice, so that each version monitors only a subset of properties lowering the overall cost of monitoring. This technique can be effectively used in software deployment where each version would be executed by several users and no single user will have to take the entire burden of monitoring. The cost of reduced monitoring would be missed violations for the observables that do not appear in the chosen set of properties. However, a violation detected is guaranteed to violate the top property. The technique was evaluated using real open source applications that use Hibernate [48] to understand the impact of subalphabet size on detecting property violations and understanding effectiveness of the technique in comparison with the original properties. The authors observe that increasing subalphabet size increases the effectiveness of the technique. However, it also increases the monitoring cost although not necessarily in the same proportion. For low overhead, the sampling of

lattice properties is found to be more effective than the original property. One potential problem with this technique is that there may be *hot* regions in a program generating only one or two symbols. In that case, a user executing these regions with instrumentation would have to take larger monitoring burden compared to other users monitoring other properties that do not involve these symbols. Hence, it is unclear if subalphabet property sampling would always result in monitoring overhead that is fairly distributed among all users.

Bodden et al. address a problem similar to the one addressed by Dwyer et al. [33] by using a different approach for collaborative runtime verification [18]. Instead of sampling on subalphabet, the authors propose partial instrumentation of the program so that users executing different copies of the program can share the burden of monitoring. The authors propose two techniques for partitioning instrumentation, spatial in which different users observe different program points for violation and temporal in which monitoring is switched on and off depending on the cost of monitoring in terms of time, users are willing to pay. For spatial partitioning, a static analysis is first performed to identify consistent *shadow* groups. These are identified in such a way that the monitor observes all related events missing which may cause a false positive or negative. However, this approach may suffer due to instrumentation embedded inside hot program regions that gets executed more often than other instrumentation, say in loops. Moreover, overlapping points-to sets of *skip* shadow could add extra probes. Temporal partitioning would improve on such high overheads with the cost of missing violations. The evaluation on DaCapo benchmarks shows that with a few exceptions, distribution of probes results in reduced overheads, although not necessarily in the same proportion.

2.5.3 Hybrid Approaches

Most of the hybrid approaches discussed here are similar at a higher level in the sense that they perform static analysis first to identify program points for which instrumentations can be safely dropped, and then analyze the program during runtime with the help of the remaining instrumentation. However, they differ considerably in the details of underlying static analysis. Among the research that is closely related to our work and also uses a hybrid approach, major part of it has been performed by Eric Bodden, Laurie Hendren, Patrick Lam, Ondřej Lhoták, Nomair Naeem and their colleagues. Their contributions, along with ours, are among the first ones in this field. It should be noted that our approaches for both, residual analysis as well as loop transformation, are considerably different. Residual analysis, may combine the instrumentation points using summary transitions, whereas loop transformation is the only approach that uses program transformation technique for optimization. Here we give an outline of their research along with some other notable work.

Bodden et al. present a multi-staged analysis [19] similar to the one presented by Fink et al. [40], with a few differences. Unlike Fink et al.'s approach, Bodden et al.'s approach is hybrid. So for the PPFs that the analysis failed to show safe, the instrumentation is retained for runtime monitoring. Fink et al.'s analysis is in the context of uni-object typestate properties, whereas Bodden et al.'s analysis targets both uni-object as well as more general, multi-object properties. The first stage is a simple and efficient flow-insensitive check on observables to eliminate infeasible matches. The second stage is a inter-procedural context-sensitive, but flow-insensitive analysis to eliminate infeasible matches due to inconsistent bindings. For every shadow, the analysis checks if there exists a consistent group of which the shadow is a member. If not, instrumentation is dropped for the shadow. The third stage is more precise and expensive whole-program flow-sensitive analysis to check the order of

execution of the shadows. The evaluation by the authors on DaCapo benchmarks reveals that this stage is least effective as first two stages are responsible for dropping most of the shadows. The evaluation, however, shows the effectiveness of the technique as it could analyze most program-property combinations with an overhead of $< 5\%$. However, some combinations still incur a very large overhead. One problem with this approach is that once found to be a member of a consistent shadow group, a shadow will be retained irrespective of whether it can really contribute to an error. For example, the analysis does not check if a shadow is capable of causing a state change. The residual analysis [37] can deal with these cases effectively. Bodden addresses this issue in his recent work [17].

In our work [68], we have used a staged approach to static analysis which is somewhat similar to the above approach. We first perform an efficient flow-insensitive check to identify methods that have observables. We then use AST structures generated by Dava [64, 65] to identify methods with loop carrying observables. Our final stage is a more precise flow-sensitive, method-summary-based points-to and side-effects analysis that ensures the eligibility of loop for transformation. Our experience along with the work presented by Bodden et al. [19] and Fink et al. [40] have shown that multi-staged approach can scale to real applications.

Naeem and Lhoták present a precise whole-program context-sensitive and flow-sensitive typestate analysis for Tracematches [66]. The higher precision of analysis is at least partly due to the typestate and points-to information calculated together at every step. This is different from our analysis as well as Bodden et al.'s analyses that calculate this information in separate passes. In addition, the authors use a novel lattice-based representation to model points-to relationship between variables and objects. This representation can simultaneously hold *positive* as well as *negative bindings*, that correspont to *must* and *must-not aliasing* respectively. Author's lattice-based model addresses the problem related to unbounded number of tracematch automata, one each for every possibly related set of objects

associated with a tracematch, that creates difficulties in static analysis. The lattice used by the authors stored Tracematches parameter bindings with abstract objects instead of concrete objects which limits its size. The size is further controlled by not applying the *focus* operation to an abstract object once it is bound by a tracematch state. This results in strong updates. The focus operation splits an object into two resulting in the exponential growth of abstract objects [40]. As shown by the evaluation on DaCapo benchmarks, the extra precision in the points-to analysis resulted in ruling out several PPFs that Bodden et al.'s analysis [19] failed to show as safe. However, there were a few cases where this analysis was not as effective as the one presented by Bodden et al. The flow-sensitive analysis presented by Bodden et al. in the final stage performs only weak updates in the context of skip-shadows due to lack of must-alias analysis. This results no extra precision over the previous phase and in retaining a lot of shadows that could have been ruled out otherwise. For these reasons, Naeem and Lhoták's analysis score over the analysis in [19], particularly in the handling of `HasNext` tracematch where calls to `hasNext` and `next` often occur in succession in loops.

Bodden et al. [20] tried to remove some of the deficiencies in the approach presented in their earlier work [19]. The authors made a major change in the design by replacing the whole-program analysis by more precise, but less expensive summary-based flow-sensitive intra-procedural analysis. The approach makes conservative assumptions particularly about the feasible states at the entry of a method by assuming that a monitor could be in any state. This makes analysis less precise but preserves its soundness. The more precise intra-procedural analysis generates information that helps drop instrumentation from more shadows at the same time improve completeness of the analysis. For example, for property `HasNext`, a method $m1$ called by another method m may not impact the analysis of m, if it uses an iterator object not related to the one used in m. The intra-procedural analysis that we developed in our work [68], is similar in nature and is based on similar sound obser-

vations. For example, in case of property `FailSafeIter`, if method m has a loop that has observables corresponding to symbol `update` and calls method $m1$ that also has observables corresponding to only `update` where the receive object relationship is unclear, $m1$ does not impact the analysis of the loop as in either case, the loop still satisfies the necessary conditions mentioned in Section 5.5.2 and execution of method statement corresponding to `update` in $m1$ or absence of it, does not compromise with the correctness of the analysis. Of course, the condition may not hold if $m1$ can potentially execute an observable statement corresponding to other symbols such as `next`.

When evaluated using DaCapo benchmarks, Bodden et al.'s analysis could rule out majority of PPFs for majority of benchmarks except when applied on benchmarks `bloat` and `pmd` to check iterator properties. Another novelty of this work is presentation of machine learning algorithm and PPF ranking system that helps ruling out more PPFs and prioritize inspection respectively. However, this algorithm is unsound. When applied on the results of the analysis, it removed majority of PPFs that were false positives, but in one case also removed actual point of failure. It would be interesting to see how it works on applications that have real bugs.

Bodden presents *Nop-shadows* analysis [17], a more precise flow-sensitive analysis. The analysis improves on the previous analyses by identifying shadows that cannot contribute to the property violation and then disabling their instrumentation. In addition to the flow-sensitive forward pass that calculates for every statement s, the states that may reach s, the analysis also has a backward pass that calculates possible future states that would lead to an error by the program execution that follows s. This backward pass not only helps identify redundant shadows that need no instrumentation, but also deals effectively with a soundness problem with the analyses presented in the previous work. Previous analyses are based on calculation of *shadow history* of statement s, i.e. the shadows that can reach s. The soundness problem arises due to accidental disabling of conditionally executed shad-

ows that ensure that a tracematch is not matched, that otherwise would be matched. The backward pass corrects this situation by calculating future states for every shadow.

Since a nop shadow is a nop shadow only if all other shadows are enabled, the analysis drops only one shadow at a time to preserve the soundness. This means for a method with n nop shadows the analysis performs n iterations. This is similar to the process of calculating safe regions inside methods in residual analysis as explained in Section 4.4.3 in Chapter 4. An advantage of residual analysis algorithm is that it starts with a window with a maximum feasible size and then reduces it, if required, depending on the boundaries of an available safe region, the algorithm can deal with more observables at a time and terminate faster unless the method has as many nondeterministic regions as observable statements which is unlikely in practice. In addition, residual analysis makes use of summary transitions that results in larger safe regions, which in turn, results in even faster termination.

CLARA [16, 21] is a framework for hybrid typestate analysis that not only provides a suite of three static analyses, but also provides a facility for users to add their own analyses. Users provide a property of interest directly as an annotated aspect that specifies a nondeterministic FSA. Alternatively, they can use a high level specification compiler such as JavaMOP or Tracematches to translate these specifications to aspects. The compiler tool can generate necessary annotations, and if not then the user provides those manually. CLARA first weaves the aspects in the base program and then with the help of its static typestate analyses disables the shadows that are not relevant to the property. Two of the three analyses are similar to the analyses that formed the first two stages in Bodden et al.'s previous work [19]. The third analysis is the Nop-shadows analysis. We believe that our analyses would be complementary to the analyses developed by Bodden et al. and if combined should produce even better results. We believe that this would be an interesting area for future work.

Martin et al. present PQL, a *Programming Query Language* [60] that checks if a given program conforms to design rules. It follows a hybrid approach of using static analysis first to eliminate unnecessary instrumentation and then using dynamic analysis on the leftover. Its implementation is based on `bddbddb` program database for storing information generated by static analysis and `Datalog` for running queries on that data. The design rules can be specified as a pattern along with the actions that need to be performed if the pattern is matched. A support for subqueries allows PQL to match context-free grammars. This means PQL is more expressive than Tracematches. However, this feature normally comes with a loss in the precision and PQL is no exception. As shown by the studies [6, 26], PQL incurs higher overhead than both Tracematches and JavaMOP.

Goldsmith et al. [45] present PTQL, a Program Trace Query Language, which is a very expressive SQL-like query language over program traces. The authors provide a compiler, PARTIQLE for online analysis that generates instrumentation to keep query-related data in runtime tables that is used to feel empty slots in a query. It also generates query evaluation code. The author's approach is similar to the one taken by Martin et al. [60], but Martin et al.'s approach performs deeper static analysis. The records are time-stamped and this information is used in timing analysis that builds graphs that hold timing relationship between *start* and *end* events for every identifier in `FROM` clause of the query. The *timing graphs* are optimized for efficiency. PARTIQLE uses static filtering to ensure that instrumentation is not applied to sites that violate a predicate. In addition it also performs dynamic filtering at the instrumentation site by checking predicates involving fields from only one record. The query evaluation is triggered at points that generate potentially last event in the query. It is unclear if this strategy would result in extra overhead as runtime tables will be joined and searched for the results at these points which are all expensive operations. PTQL has been evaluated on benchmark and property combinations that are different than the ones used

by other tools such as JavaMOP and Tracematches, and hence it is difficult to compare its performance with either JavaMOP or Tracematches.

2.5.4 Specification Mining

It may not be always easy for a programmer to express the properties of interest for monitoring. Moreover, APIs can be complex and their documentation is often ambiguous and incomplete. This may result in a programmer generating an incorrect specification. At the same time, she would still be interested in understanding the program behavior with respect to several properties, and ensure that those properties are satisfied during runtime. The specifications would help her understand and debug programs. To address this problem, researchers have proposed several promising approaches in the last decade to extract specifications automatically based on static or dynamic analysis of programs. Many of the dynamic approaches work on the assumption that the program behavior would mostly be correct, so that an exceptional or unobserved behavior may correspond to an error.

Researchers have developed tools such as Daikon [39] and DIDUCE [46] that generate program invariants based on observed program behavior. The invariants are then checked during runtime and violations, if any, are reported. Daikon uses statistical techniques to avoid over-reporting of errors, whereas DIDUCE has a *learning* mode in which it gradually relaxes an invariant for every violation and reports a warning for the first occurrence of every violation during its *checking* mode. The program invariants in both the cases correspond to state properties, whereas our research is focussed on checking typestate properties that are path properties. Hence, in this section, we mainly discuss the research that is about checking path properties. Many of these approaches that generate specifications for temporal properties can be viewed as approaches that can or can be extended to generate typestate

property specifications. However, most of these approaches do not target properties that have associated parameters such as objects.

2.5.4.1 Machine learning based approaches

Ammons et al. [3] propose a mining technique that begins with isolating traces that are unrelated and then uses a machine learning approach based on an off-the-shelf probabilistic FSA model to generate temporal specifications. The infrequently traversed edges in the resultant FSA are pruned. The validation of the FSA is performed by human inspection. This step may create a bottleneck in the process of specification mining as a miner may generate an incorrect specification based on erroneous traces, and a programmer may have to check the generated specification for potentially very large number of unordered traces to discard the erroneous ones. Ammons et al. address this problem later [4] by presenting a technique *concept analysis*, that orders and clusters similar traces, so that the programmer has to check only a small subset of the traces.

Lee et al. [56] present a technique and its implementation; a tool named JMiner. Their approach targets parametric specifications and multi-object properties. It is based on trace slicing that differentiates traces based on the related objects and then passing the isolated traces to machine learning phase. This phase consists of an off-the-shelf probabilistic FSA learner and then refining its potentially over-generalized results using a FSA learner that authors have built. Most of the other tools work on the traces provided to them, however, JMiner can generate traces.

Lo et al. [58] present a technique that improves a precision of the *kTail* algorithm [13] which , in turn, results in improving the precision of FSA inferred from traces. It is the state merging phase of *kTail* that introduces this imprecision as it performs a lookup over next k outgoing transitions from the two states under consideration to check for their equivalence before making a decision about their merging. This operation occasionally merges the

states erroneously if their near future is same, but distant future is different. The authors present a statistical technique for mining temporal rules and then using these rules to *steer* the kTail algorithm so that it can make a better decision about merging states. If the states violate any of the temporal rules, the states are not merged. This improves the precision in the extracted FSA specification. The evaluation of the technique shows that it improves the specification precision without adding much time overhead.

Most of the specification mining techniques generate either data predicates that are state properties, or FSA that are path properties. However, they do not generate FSA that are parametrized with data. Lorenzoli et al. [59] bridge this gap by presenting an algorithm called *GK-Tail* that generates extended finite state machines (EFSM) from the program traces. GK-Tail is a variant of k-Tail algorithm and has its own rules for finding equivalent states that can be merged. Its evaluation indicates that it can extract several EFSM specifications that cannot be meaningfully described by FSM specifications.

2.5.4.2 Pattern template based approaches

Yang et al. present an approach [73] that can deal with imperfect traces caused either by the presence of bugs or due to their incompleteness, using statistical techniques. The authors target temporal specifications that have alternating pattern `(a b)*`. They consider traces that have a context as well as those that do not. Traces with context provide richer information, but can be made useful only when appropriately generalized based on object identity. Irrelevant or less interesting traces are pruned using *method reachability* within a call-graph and exploiting *naming conventions*. Results are simplified and generalized using *chaining* that combines multiple specifications in a meaningful way. The technique is scalable, but explores only one kind of pattern and its generalization. There are several other interesting patterns investigated by other researchers including a few that are associated with only two variables.

Gabel and Su present a similar approach [42] that is based on using a pattern template, but in addition to the pattern `(a b)*`, the authors also consider a pattern `(a b* c)*`. Authors show that the composition of these two patterns can yield complex patterns of practical interest. The authors show that the mining problem is NP-hard and then make an improvement [43] over the algorithm provided by Yang et al. [73] for solving mining problem for input automaton with alphabet of size 2. Gabel and Su's algorithm is based on binary decision diagrams, where as Yang et al.'s algorithm is matrix-based. The authors do not mention if extracted specifications are parametric. In the absence of parameters, the extracted specifications may be noisy.

2.5.4.3 Static approaches

All of the approaches mentioned so far are dynamic; that is they run a program and then analyze its runtime behavior. One limitation with these approaches is that the specifications extracted are limited to the observed behavior. In order to capture the program behavior especially when multiple APIs are used across multiple procedures, Acharya et al. [1] propose a static analysis technique implemented using a model checker. The traces are modeled as partial orders and fed to a miner for extracting specification. A weakness of this technique is that partial orders cannot describe looping and hence the technique cannot extract property specifications that have loops.

Chapter 3

A Cost Model for Runtime Monitoring

3.1 Introduction

Over the past decade, researchers have developed a number of monitoring techniques and implemented those techniques in tools that can be used to analyze large complex applications [5, 8, 17, 25, 26, 27, 47, 52]. In spite of this progress, there exist programs for which monitoring with respect to certain properties incurs significant runtime overhead which hinders the application of monitoring in practice. When monitoring programs, the property FSA must be *bound* to data values. For example, monitoring a property that expresses the legal sequencing of calls on a Java class requires that the value of the receiver object be used to correlate calls – a call on one instance of the class is independent of calls on other instances of the class. For each collection of objects that is related to a property an instance of a *monitor* is created and all subsequent calls on those objects generate *events* that are routed to track the state of the monitor, based on the FSA, and detect property violations.

It seems sensible that the number of monitors that are created and the number of events generated by the program execution influence the cost of monitoring. For example, for the DaCapo benchmark [14] `pmd`, Tracematches [2] observes over 33 million and over

25 million events, respectively, for properties `HasNext` and `SafeIterator` [62]. This leads to runtime overheads of 52% in the former case and 175% in the latter case. Similarly, the `bloat` and `pmd` benchmarks create nearly the same number of monitors, in excess of a few million, for property `SafeIterator` [62] for JavaMOP [26]. Despite the similarity in number of monitors `bloat` incurs an overhead of 769% whereas `pmd`'s overhead is only 19%. While the number of monitors and events are significant contributors to overhead, it seems clear from just these examples that there are other important factors in play.

There are a number of different algorithmic approaches to monitoring of typestate properties. We distinguish two basic categories: *object-based*, which maintain a set of monitors associated with each class instance relevant to the property, and *state-based*, which maintain a set of monitors associated with each state of the property FSA. Both of the approaches have been discussed in detail in Section 3.2.3. The JavaMOP tool implements object-based monitoring and Tracematches realizes the state-based approach. These tools can incur contrasting monitoring overheads for the same benchmark-property combinations. For example, Tracematches incurs an overhead of 2452% compared to an overhead of 1112% for JavaMOP when monitoring property `HasNext` on the `bloat` benchmark [62]. However, for the same property, Tracematches incurs an overhead of only 52% for benchmark `pmd` compared to JavaMOP's 191% overhead [62]. In this chapter, we explore the sources of these differences in performance.

We adopt an analytic approach, rather than an empirical one, since we seek to explain performance differences in order to gain insights that might drive the development of improvements in runtime monitoring. Towards that end, we abstract and generalize the state and object-based approaches of JavaMOP and Tracematches and construct models that define and relate the key components of cost in runtime monitoring. These models attempt to characterize the relative contribution of different algorithmic variants of the two monitoring approaches in order to permit comparisons. With the cost models in hand, we revisit

several previously published results, including some of our own, in order to better explain reported data on monitoring overhead, the conclusions drawn from those data, and to identify opportunities for further performance improvement.

3.2 Motivation and Background

Several efficient runtime monitoring tools have been developed in the past decade. While they perform well for many combinations of programs and properties, they have also been found to generate very high overhead occasionally which may restrict their usage. To date, the reasons for variation in the cost of runtime monitoring have not been studied or well understood.

3.2.1 Unexplained variation

To understand the variation in runtime overhead that may be observed when varying different parameters we performed a small study on programs and properties which are known to exhibit non-trivial monitoring overhead. We used two widely studied programs, the DaCapo benchmarks `bloat` and `pmd`, and typestate properties, `HasNext` and `FailSafeIter`. The programs can be run with two different inputs *small* and *default*. We used JavaMOP, a tool implementing object-based monitoring, and Tracematches, a tool implementing state-based monitoring. For each tool, we considered several options.

Object indexing provides support for efficiently locating the set of monitors that need to be updated when a program event is generated; events include information on a set of related program objects and they are used to "lookup" the set of monitors. JavaMOP includes support for indexing all of the objects involved in an event, whereas Tracematches permits just a single object to be indexed. The indexing options can be *enabled* or *disabled* in the tools.

No.	BM	Property	Input	Matching	Indexing	Tool	% Overhead
1	bloat	FailSafeIter	default	complete	enabled	JM	702.1
2	bloat	FailSafeIter	small	complete	enabled	JM	73.7
3	bloat	FailSafeIter	default	suffix	enabled	JM	3291.7
4	bloat	FailSafeIter	small	suffix	enabled	JM	443.6
5	pmd	FailSafeIter	default	suffix	enabled	JM	87.0
6	pmd	FailSafeIter	default	suffix	enabled	TM	9119.6
7	pmd	FailSafeIter	default	suffix	disabled	TM	41806.5
8	pmd	HasNext	default	suffix	enabled	JM	73.9
9	bloat	FailSafeIter	default	suffix	enabled	TM	>149900
10	bloat	HasNext	default	suffix	enabled	JM	2533.33
11	bloat	HasNext	default	complete	enabled	JM	172.9
12	bloat	HasNext	default	suffix	enabled	TM	1168.8
13	pmd	FailSafeIter	default	complete	enabled	JM	56.5
14	pmd	HasNext	default	complete	enabled	JM	15.2
15	pmd	HasNext	default	suffix	enabled	TM	91.3

Table 3.1: Variation in the runtime overheads caused by different combinations program, property, input and tool configurations. * Indicates that the process was stopped after 7200 seconds. JM = JavaMOP and TM = Tracematches.

The trace of events generated by a program can be matched against the FSA specification using two different approaches. In *complete* matching the trace is tested for membership in the language of the FSA; when a trace prefix cannot be extended to an accepting trace then violations can be detected before the program terminates. FSA properties can also be specified that are supposed to hold beginning at any point in the program trace—one can think of these as invariants across the trace. Checking such properties is performed using *suffix* matching; conceptually this involves creating a new monitor after every event to check the property against the suffix beginning at that event. JavaMOP supports both complete and suffix matching, whereas Tracematches only supports suffix matching.

The monitoring overhead for a non-exhaustive set of combinations of these parameters is provided in Table 3.1. The data show that even for a limited choice for individual parameters the combinations can be numerous and the overhead ranges significant.

Intuitively, we expect that the nature of the program, property, and input to strongly influence overhead. Comparing rows 1 and 2 and then 3 and 4, illustrates that, as expected, overhead is sensitive to program input; overhead drops by about an order of magnitude in both cases. The 3rd and 5th rows, with overheads of 3291% to 87%, demonstrate the influence of the program and the 1st and 11th, with overheads of 702% and 172%, demonstrate the influence of the property on overhead.

Algorithmic techniques for runtime monitoring should also be expected to influence overhead. Rows 1 and 3 illustrate the influence that varying the matching type can have on overhead; overhead increases by more than a factor of 4 to 3291% across this pair. Rows 6 and 7 indicate that indexing can significantly reduce overhead from 41806% to 9119.6%.

Even when holding the benchmark, property, input, checking mode and indexing scheme fixed the choice of checking tool can matter. Rows 3 and 9 show that JavaMOP slows the program down by a factor of more than 32, whereas Tracematches timed out so its overhead is at least 149900% on the `bloat` benchmark and `FailSafeIter` property when running in suffix mode. Switching just the property, to `HasNext`, however, reveals that the overhead of Tracematches can be less than half that of JavaMOP; row 12 shows 1168.8% for the former and row 10 shows 2533.33% for the latter.

Our intention here is not to provide a comprehensive explanation of the sources of variation in monitoring overhead, but rather to point out that predicting the overhead of runtime monitoring is difficult since it depends on many factors. While not illustrated here, there are additional implementation factors that can strongly affect monitoring cost. For example, while we used the same machine, OS, and JDK, for all runs reported in Table 3.1, we did observe that changing the platform may either increase or decrease monitoring overhead depending on how it interacts with the monitoring implementation; more specifically, we observed the effect of differences in garbage collection implementations. It is also known

that the choice of data structures to use for indexing can also dramatically affect performance [26].

3.2.2 Our Motivation

While tools like JavaMOP and Tracematches represent state-of-the-art runtime monitoring implementations, our goal is not to characterize or compare implementations. We believe that advancing research on runtime monitoring requires the ability to look beyond implementations to understand the performance of the underlying algorithmic techniques. Such insights will help researchers draw more generalizable conclusions about proposed techniques and to identify new opportunities for optimizing the performance of runtime monitoring.

We believe that a key tool for enabling such insights are *detailed cost models* for runtime monitoring techniques. These models should support reasoning about the relative costs of variants of a technique. Consequently, they should account for the detailed operation of techniques and their variants, e.g., indexing and matching types. We emphasize that our goal is not to provide accurate predictions of the overhead of a particular implementation, but rather to support analytical reasoning about the potential benefits of new algorithmic techniques.

We expect for the cost models for runtime monitoring to play a similar role to the detailed cost models used in selecting compiler optimization strategies. Modern instruction set architectures contain a number of complex interacting features, e.g., pipelines, caches, multiple functional units, and being able to reason about the expected performance of a code fragment in the presence of those features is needed to choose among optimization and code generation strategies. Similarly runtime monitoring cost models could be used to select the best technique for a particular program and property. In addition, we believe

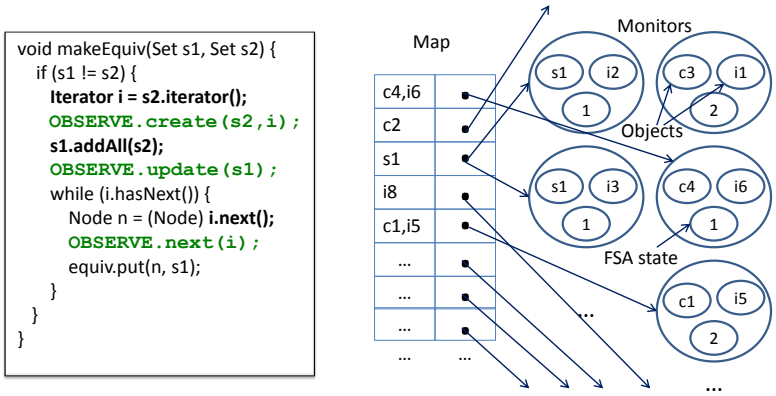
that insights gleaned from studying such models can reveal opportunities for beneficial new monitoring techniques.

In order to develop detailed cost models, one must first develop a deep understanding of the costs associated with existing runtime monitoring algorithms.

3.2.3 Monitoring approaches

Conceptually a *monitor* for an FSA property is a relation between a set of related objects created during program execution and a state of the FSA. The state reflects a sequence of FSA transitions which correspond to the execution of a sequence of program statements that are related to both the property and the set of related objects. In studying two state-of-the-art runtime monitoring implementations, JavaMOP and Tracematches, we have identified two fundamentally distinct algorithms for encoding the information associated with a monitor.

JavaMOP uses an *object-based monitoring* approach which defines a monitor as a set of related objects and a state of the FSA. The ovals on the right side of Figure 3.1 illustrate a set of monitors with objects on the top, e.g., `s1`, and the state on the bottom, e.g., `1`. Tracematches uses a *state-based monitoring* approach which defines for each state of the property FSA a set of monitors which each record a set of related objects. The rectangles on the lower right of the parts of Figure 3.4 illustrate a set of monitors, depicted as ovals, each consisting of a set of objects, e.g., `s1, i7`. In state-based monitoring the state is factored out of the monitor and defined by the state with which the monitor is associated. In essence these approaches are duals differing only in the element of the object-state relation they use as the key for organizing monitoring data. We illustrate and sketch these algorithms in the remainder of this section.



(a) Before the generation of event `Create`

(b) After the generation of event `Create`

(c) After the generation of event `Update`

Figure 3.1: Simple object-based monitoring scheme for complete matching.

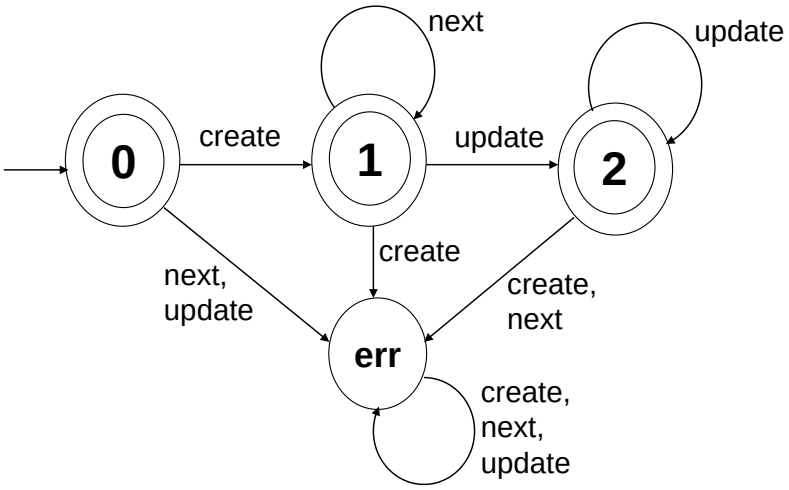


Figure 3.2: Property FailSafeIter

Object-based Monitoring Figure 3.1 illustrates a simple object-based monitoring scheme for complete matching. Figure 3.2 shows the FSA for the `FailSafeIter` property used in the example, which is same as Figure 1.2, but is reproduced here for easy reference.

The relevant statements for this property are calls to methods `iterator()`, `addAll()` and `next()` on appropriate receiver objects. The figure also shows a fragment of code being monitored where the relevant statements are instrumented with extra code that is used to perform monitoring, i.e., the `OBSERVE...` calls.

Figure 3.1(a) shows an example of the monitoring data structures before the execution of an `iterator()` call. The call generates a monitoring *event* which consists of a tuple (l, b) where l is a set of associated objects, in this case $\{s2, i\}$, and b is an FSA symbol, which in this case is `create`. The scheme provides a map, keyed by sets of related objects that have been involved in previous events. For simplicity, we have shown a single

```
ObjectBasedMonitoring(φ = (S, Σ, δ, s0, A), e = (l, b))
 1  let L be the set of sets of objects that receive events
 2  let Σc ∈ Σ be the set of creation symbols
 3  let MS be the set of sets of monitors
 4  let ObjsMons : L → MS be a map
 5  let ObjsSym be a binary relation over L and Σ
 6  if b ∈ Σc ∨ ObjsMons(l) = null ∨ suffix then
 7      m ← new monitor(l)
 8      m.cur ← s0
 9      for l' ⊆ l do
10          if ∃σ ∈ Σ : (l', σ) ∈ ObjsSym then
11              ObjsMons(l') ← ObjsMons(l') ∪ {m}
12  ms ← ObjsMons(l)
13  for m ∈ ms do
14      m.cur ← δ(m.cur, b)
15      if (m.cur = err ∧ ¬suffix) ∨ (m.cur ∈ A ∧ suffix) then
16          report error
```

Figure 3.3: Simple object-based monitoring algorithm.

map, but in practice for efficiency reasons, multiple maps may be provided. The values corresponding to the keys are sets of monitors that are associated with the objects.

Figure 3.1(b) shows the situation after the scheme handles the `create` event. Since no monitors are associated with collection `s2` and iterator `i`, a new new monitor is created and references to it are associated with the new keys `s2`, `i`, and `s2,s1`.

Figure 3.1(c) shows how a subsequent `update(s1)` event, which is triggered by a call to `s1.addAll()`, is handled. The monitors associated with `s1` in the map are retrieved. For each such monitor, an FSA transition is simulated to update the state. In this example, both the monitors associated with `s1` were previously in state 1 so they are updated to $\delta(1, \texttt{update}) = 2$ based on the FSA in Figure 3.2.

A simple algorithm for object-based monitoring is sketched in Figure 3.3. Lines 6–8 handle the creation of new monitors which are initialized to the FSA start state. This is only performed if either the symbol corresponds to a creation event, no map entries exist for the

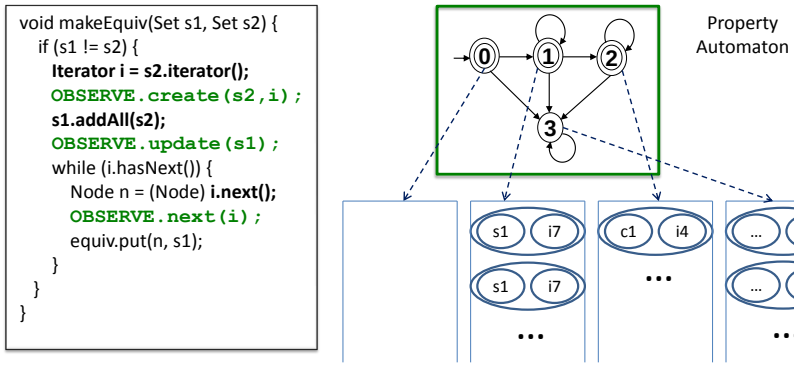
associated objects, or suffix matching is enabled[1]. Lines 9–11 add the new monitor to the sets of monitors that are associated with each subset of l that may witness a future event. Finally, lines 12–14 simulate FSA transitions for the states in every monitor associated with the objects involved in the event. If any of the monitors goes to the error state (or an accept state in the case of suffix matching) an error is reported as shown by lines 15–16.

State-based Monitoring Figure 3.4 illustrates a simple state-based monitoring scheme for complete matching for the same property and program fragment that was shown in the earlier example. Monitoring is performed over a single copy of the property FSA. A set of monitors are associated with each state and during monitoring those monitors are moved between states based on the events encountered and δ.

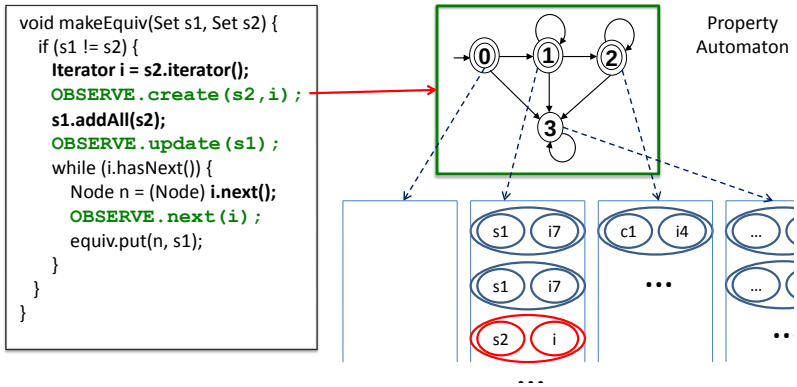
Figure 3.4(a) shows the situation just before the execution of a call to `iterator()`. The monitoring system searches for monitors associated with both `s2` and `i` by querying the sets associated with each FSA state. It fails to find any monitor, so it creates a new monitor, which consists of just the objects for state-based monitoring, and adds it to the set associated with state $\delta(0, \texttt{create}) = 1$, where state 0 is the FSA start state. The result is shown in Figure 3.4(b). Lines 4–6 in Figure 3.5, which sketches a basic state-based monitoring algorithm, describe the monitor creation steps in detail.

Figure 3.4(b) shows how the subsequent `update(s1)` event is handled. The monitor sets for each state are searched, lines 7 – 10 in the algorithm, to find monitors involving `s1`. This results in the two instances $\{s1, i7\}$ at state 1 being found. All of those monitors are then moved to appropriate states depending on the encountered symbol as shown in lines 11–12. In the example, this causes the monitors to be removed from state 1's set and added to the set for $\delta(1, \texttt{update}) = 2$. Line 8 skips the move for self-loop transitions. Lines

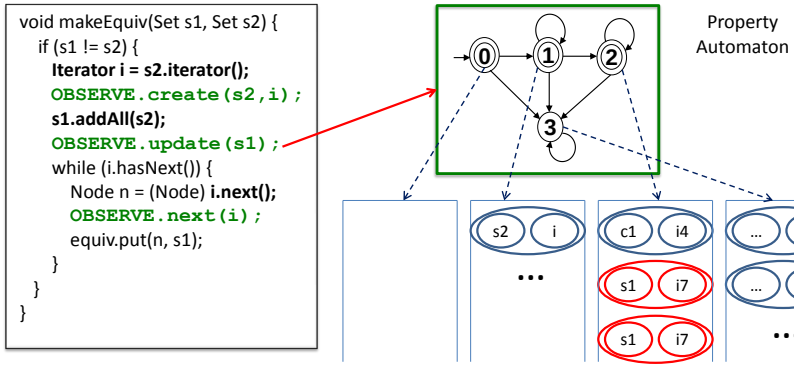
[1]For simplicity, when performing suffix matching, this algorithm creates new monitor for every event, but in practice for efficiency a monitor may maintain a set of states and simply add a start state to perform suffix matching. Additionally, to control the number of unnecessary monitors, implementations will also ensure that duplicate monitors with identical current states will be deleted from the pool.



(a) Before the generation of event `Create`



(b) After the generation of event `Create`



(c) After the generation of event `Update`

Figure 3.4: Simple state-based monitoring scheme for complete matching.

```
StateBasedMonitoring(φ = (S, Σ, δ, s_0, A), e = (l, b))
 1 let Σ_c ∈ Σ be the set of creation symbols
 2 let MS be the set of sets of monitors
 3 let StMons : S → MS be a map
 4 if b ∈ Σ_c ∨ (∀s ∈ S  StMons(s) = null) ∨ suffix then
 5     m ← new monitor(l)
 6     StMons(s_o) ← StMons(s_o) ∪ {m}
 7 for s ∈ S do
 8   if s ≠ δ(s) then
 9     for m ∈ StMons(s) do
10       if m.l = l then
11         StMons(s) ← StMons(s) − {m}
12         StMons(δ(s)) ← StMons(δ(s)) ∪ {m}
13         if (δ(s) = err ∧ ¬suffix) ∨ (δ(s) ∈ A ∧ suffix) then
14           report  error
```

Figure 3.5: Basic state-based monitoring algorithm.

13–14 show that if the target state is the error state (or an accept state in the case of suffix matching) an error is reported.

While relatively simple, these algorithm sketches illustrate the operations that must be performed during monitoring. For example, the creation of monitors, updating states, and searching and updating sets of monitors all contribute to monitor cost. These costs may become non-trivial especially for a sophisticated implementation of a monitoring system that may involve complex indexing and other data structures. The cost models detailed in the next section provide a more thorough explanation of the costs incurred in various monitor operations.

Note that both algorithms presented here can perform either complete or suffix matching. For suffix matching the tool keeps a set of current states instead of a single current state. The main difference between the two approaches is that for suffix matching, a match will always be started from the beginning in addition to other current states. This is highlighted by the separate algorithms for complete and suffix matching presented in Sections

2.3.1 and 2.3.2. However, those algorithms are highly generalized and do not consider any specific tool building approach.

The algorithms presented here do not maintain a set of current states. Our choice here is mainly influenced by the implementation of JavaMOP that keeps a separate monitor (although under a common wrapper corresponding to the related set of objects) each for every current state it is in. The purpose of the algorithms here is to highlight the effect of underlying tool building approach on monitoring algorithm and the usage of completely different data structures for monitor organization.

3.2.3.1 Basic Definitions

Let $\phi = (S, \Sigma, \delta, s_o, A)$ be the FSA for the property being monitored and π the trace of events related to ϕ that is generated by a program execution. The i^{th} event in the trace, e_i, is a pair (b_i, l_i) where $b_i \in \Sigma$ and l_i is the set of objects associated with e_i. Note that, $(i \neq j) \Longrightarrow (b_i \neq b_j)$ and $(i \neq j) \Longrightarrow (l_i \neq l_j)$.

Let E be the set of events and $E_c \subseteq E$ be the set of *monitor creation events* or simply *creation events*. Note that, the members of E_c may or may not have any special symbols associated with them. Any event can be a creation event especially for a uni-object property. For example, while monitoring for the `HasNext` property, either of the two events, generated by calls to methods `hasNext` or `next` can be a creation event if that is the first method call observed by the corresponding receiver object. If there is no monitor corresponding to a set of objects that received the event, a monitor will be created corresponding to the set of objects. In the case of multi-object properties, a creation event would typically correspond to an event in which two or more property-related objects are involved. It is possible that in a creation event, objects involved are only bound together and no object is created in the process. In either case, a creation event corresponds to an event that creates a monitor.

Let $E_b \subseteq E$ be the set of *binding* events, that bind new objects to the old ones that already have associated monitors. These events may create new monitors by combining new objects with the clones of existing monitors associated with the old objects. For example, if a property involves three objects o_1, o_2 and o_3, where o_1 and o_2 are involved in a creation event, and o_3 is created by calling method b on o_2 that binds o_3 to o_2 and then, in turn, to o_1. For convenience, we divide the set of objects l involved in a binding event into two partitions l_b and $l_{\bar{b}}$. Here, the call to b is a binding event and $l_b = \{o_2\}$ and $l_{\bar{b}} = \{o_3\}$. We assume that binding process would involve creating a new monitor by cloning the existing monitor associated with o_1 and o_2 and combining that with o_3.

Let $\Sigma_c \subseteq \Sigma$ be the set of symbols that are associated with monitor creation events. We define a predicate ρ that tests if all of the objects associated with a monitor for property ϕ are also involved in the monitor's creation event.

Let $\pi_i = e_1, \ldots, e_i$ be a prefix of π where $1 \leq i \leq |\pi|$.

We define τ_b as the set of types of objects that may be associated with symbol b. Let L be the set of sets of types of objects associated with all symbols. That means $L = \{\tau_b : b \in \Sigma\}$. For example, for the property `FailSafeIter`, $L = \{\{Collection, Iterator\}, \{Collection\}, \{Iterator\}\}$ and for `HasNext`, $L = \{\{Iterator\}\}$. Therefore, we have $g : \Sigma \twoheadrightarrow L$, where g is a surjection.

We make following assumptions.

1. The monitors are never reclaimed. Monitors can be reclaimed if they are not required in the future which may depend on whether the associated objects are reachable and can incur an event. The terms defined in the cost models use a prefix of a trace to compute values for the factors that appear in the definition. In order to decide whether a monitor can be reclaimed a model would need an additional information

about whether the associated objects are alive and may incur an event. This is beyond the scope of the current model.

2. A monitoring tool may support indexing, as explained in Section 3.2.3.2, to provide faster access to monitors. We assume that if indexing is provided, the tool would use a centralized scheme in which all objects share the index trees. In addition, the scheme would multi-leveled. That means it would use maps at multiple levels, each of which can be accessed using an object reference as a key. In other words, an index tree can be accessed using a set of related objects as a key.

3. For suffix matching, a new monitor is created for every new symbol. The primary reason for our choice of model is that both JavaMOP and Tracematches support it. Alternatively, a monitor may keep a set of states and the start state may be added to that set every time a symbol is encountered.

Before presenting the cost component definitions, we discuss the critical cross-cutting issue of monitor indexing.

3.2.3.2 Monitor Indexing

Since the monitored properties are often defined over multiple objects, their handling must be part of a cost model. In defining ϕ, each symbol is associated with objects of a given type. For the example in Section 3.2.3, a call to `iterator()` is associated with the collection being iterated and the iterator that is created. Thus, for an instance of the iterator creation event we know that l will be a pair of objects with types `Node` and `Iterator`. Similarly a call to `addAll()` will produce an update event where l is a single object of type `Node`.

The goal of a monitor indexing scheme is to allow efficient access to the set of monitor structures that are associated with a given event. For each subset of Σ that has the same

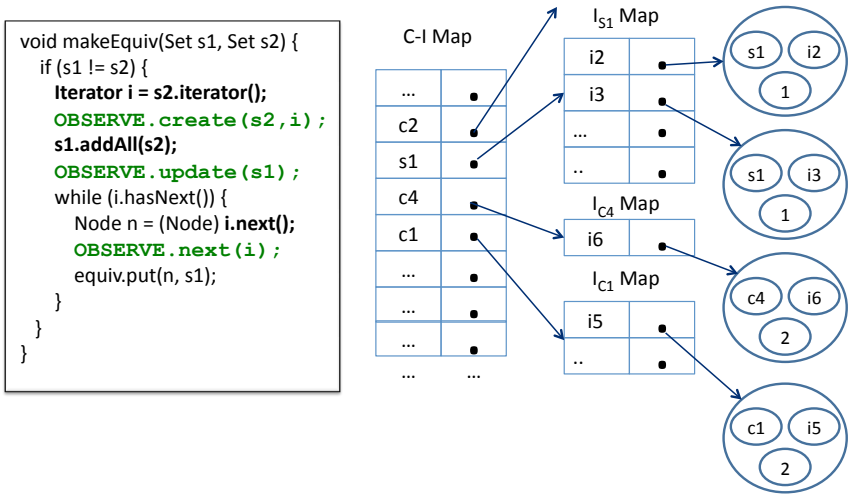


Figure 3.6: Multi-level indexing corresponding to Figure 3.1(a)

object types an *index map* is created; the set of all such maps for a property ϕ is denoted Ψ_ϕ. When an event (l, b) arrives $\Psi_\phi(b)$ is evaluated to produce an *object map*. l is used to index into the object map, i.e., $\Psi_\phi(b)(l)$, to produce the set of monitors to be updated based on b.

It has been shown that the hierarchical organization of the object map is quite efficient [26]. In such a scheme the structure of $l = (l_1, \ldots, l_n)$ is exploited to efficiently access the set of monitors. Any prefix of l gives rise to an object map, e.g., $\Psi_\phi(b)(l_1)$, and using all components of l produces a set of monitors, e.g., $\Psi_\phi(b)(l_1) \ldots (l_n)$.

Figure 3.6 illustrates multi-level indexing for the object-based monitoring example in Section 3.2.3. For pairs of objects associated with an event we see a second-level object map referenced from the entry of the index map; in general a complete indexing approach would have a number of levels equal to the size of the largest object set associated with an event. As described above the levels are indexed by using each element of l in sequence.

3.3 Cost Models

In this section, we present detailed cost models that account for the key components of the cost of runtime monitoring. These models were developed by studying multiple runtime monitoring implementations [2, 26, 53] and abstracting the operations performed to allow comparison of the underlying algorithmic techniques. The monitoring operations that incur a cost include creating and maintaining *pools* of monitors and keeping track of their states. A monitor pool could be implemented either as a set or a list of monitors.

The cost of monitoring $\mathcal{C}_o$ for ϕ is the cumulative cost of processing each event in the generated trace.

$$\mathcal{C}_o = \sum_{i=1}^{|\pi|} f(\pi_i)$$

As discussed in Section 3.2.3 the processing of each event may vary, e.g., depending on if it is a creation or update event, but we can identify five distinct components of the cost of processing an event.

1. C_C : Cost of creating monitors.
2. C_R : Cost of accessing and manipulating index trees.
3. C_I : Cost of inserting monitors into *pools*.
4. C_V : Cost of traversing monitor pools.
5. C_T : Cost of performing transitions.

Thus, the cost of processing the i^{th} event is given by

$$f(\pi_i) = C_C(\pi_i) + C_R(\pi_i) + C_I(\pi_i) + C_V(\pi_i) + C_T(\pi_i)$$

$$C_C = \begin{cases} |l_i| * c_{c_m} & e_i \in E_c \vee (sm \wedge \neg E_b) \\ \sum_{m \in \theta(l_{i_b}, \pi_i)} |P(m, \pi_i) \cup l_i| * c_{c_m} & e_i \in E_b \\ 0 & \text{otherwise} \end{cases}$$

$$C_R = \begin{cases} H(e_i, \Psi_\phi) * c_r + \sum_{v \in H(e_i, \Psi)} \kappa(v) & e_i \in E_c \cup E_b \\ H(e_i, \Psi_\phi) * c_r & \text{otherwise} \end{cases}$$

$$C_I = \begin{cases} |\Psi_\phi| * c_{a_m} & e_i \in E_c \vee (sm \wedge \neg E_b) \\ |G(l_i, \Psi_\phi)| * c_{a_m} & e_i \in E_b \\ 0 & \text{otherwise} \end{cases}$$

$$C_V = \begin{cases} 0 & e_i \in E_c \wedge \neg sm \\ \theta(l_i, \pi_i) * (c_v + c_e) & sm \\ \theta(l_i, \pi_i) * c_v & \text{otherwise} \end{cases}$$

$$C_T = \begin{cases} 0 & e_i \in E_c \\ \theta(l_i, \pi_i) * c_t & \text{otherwise} \end{cases}$$

Figure 3.7: Cost components for object-based monitoring

Note here that we define this cost in terms of the prefix of the trace up to and including the i^{th} event since some cost components are dependent on trace history.

Figures 3.7 and 3.8 define the five cost components.

3.3.1 C_C: Creating Monitors

The cost of creating monitors is the same for both object and state-based monitoring. In Figure 2.1, this cost is incurred in performing step 2. As mentioned earlier, a subset of events, $E_c \subseteq E$, is distinguished as *monitor creation events* which indicate the beginning of monitoring of ϕ for a set of related objects.

$$C_C = \begin{cases} |l_i| * c_{c_m} & e_i \in E_c \vee (sm \wedge \neg E_b) \\ \sum_{m \in \theta(l_{i_b}, \pi_i)} |P(m, \pi_i) \cup l_i| * c_{c_m} & e_i \in E_b \\ 0 & \text{otherwise} \end{cases}$$

$$C_R = \begin{cases} 0 & \neg dx \\ \sum_{s \in S} K(l_i, \alpha(s)) * c_r & \text{otherwise} \end{cases}$$

$$C_I = \begin{cases} c_{a_m} & e_i \in E_c \vee (sm \wedge \neg E_b) \\ \sum_{s \in S} J(l_{i_b}, s, \pi_i) * c_{a_m} & e_i \in E_b \\ 0 & \text{otherwise} \end{cases}$$

$$C_V = \begin{cases} 0 & b_i \in \Sigma_c \\ \sum_{s \in S} \lambda(s, b_i) * \eta(s, l_i, \pi_i) * & dx \wedge\ K(l_i, \alpha(s) \geq 0) \\ \quad ((|l_i| - K(l_i, \alpha(s))) * c_e + c_v) & \\ \sum_{s \in S} (\lambda(s, b_i) * \beta(s, \sigma_i) * (|l_i| * c_e + c_v) + & dx \wedge\ K(l_i, \alpha(s)) = 0 \\ |keys(\delta(s, b_i), \pi_i)| * c_v) & \\ \sum_{s \in S} \lambda(s, b_i) * \beta(s, \sigma_i) * (|l_i| * c_e + c_v) & \text{otherwise} \end{cases}$$

$$C_T = \begin{cases} 0 & e_i \in E_c \\ \sum_{s \in S} (\eta(s, l_i, \pi_i) * \zeta(s, e_i)) & \text{otherwise} \end{cases}$$

Figure 3.8: Cost components for state-based monitoring

For creation events, or when suffix matching, sm, is enabled, a monitor must be created and then inserted into the appropriate indexing structures. $|l_i| * c_{c_m}$ reflects the cost of creating a monitor component, c_{c_m}, for each of the objects in the event, l_i. For simplicity, as mentioned earlier, we assume that we always create a new monitor for a symbol observed for suffix matching, rather than adding an extra state to an existing monitor. This assumption is consistent with the implementation of JavaMOP. However, there is no reason

$$C_C = \begin{cases} |l_i| * c_{c_m} & e_i \in E_c \vee (sm \wedge \neg E_b) \\ \sum_{m \in \theta(l_{i_b}, \pi_i)} |P(m, \pi_i) \cup l_i| * c_{c_m} & e_i \in E_b \\ 0 & \text{otherwise} \end{cases}$$

Figure 3.9: C_C: Cost of creating monitors.

why a monitor implementation that has multiple current states should not be supported for suffix matching. In fact, this implementation would be more efficient than the alternative implementation that creates monitors as maintaining a set of states should have a smaller overhead.

For binding events, the situation is more complicated. The monitors that are already associated with l_b are cloned and then the clones are associated with $l_{\bar{b}}$. We define $P : M \times \{\sigma_i\} \rightarrow 2^S$, a function that takes as arguments a monitor and a prefix of the trace σ_i and outputs the set of monitor components (associated objects) at a time when the monitored program generated σ_i. A newly created monitor will have components corresponding to the binding event and the old monitor.

3.3.2 C_R: Index Trees

$$C_R = \begin{cases} H(e_i, \Psi_\phi) * c_r + \sum_{v \in H(e_i, \Psi)} \kappa(v) & e_i \in E_c \cup E_b \\ H(e_i, \Psi_\phi) * c_r & \text{otherwise} \end{cases}$$

$$C_R = \begin{cases} 0 & \neg dx \\ \sum_{s \in S} K(l_i, \alpha(s)) * c_r & \text{otherwise} \end{cases}$$

Figure 3.10: C_R: Cost of accessing monitors. Object-based monitoring (Top) and State-based monitoring (Bottom).

Accessing and manipulating maps plays an important role in both object or state-based monitoring, but the associated costs arise in different ways. In Figure 2.1, this cost is incurred in performing steps 1 and 4.

Let $\psi \in \Psi_\phi$. Let υ be a map in some index tree ψ and Υ be the set of all maps each one of which belongs to some index tree in Ψ_ϕ.

Let Ψ'_ϕ be the set of index trees that (i.e. only their roots) would be created beforehand, for all related objects are involved in a creation event. All objects involved in transition events must have associated monitors, and hence, we must have $h : L \rightarrowtail\!\!\!\!\!\rightarrow \Psi'_\phi$, where h is a bijection. However, if all related objects are not involved in a creation event, a few additional index trees that provide access to the already existing monitors based on the binding objects, in addition to the trees that provide access to the non-binding objects that may have associated transition events would be required. In the previous example involving objects o_1, o_2 and o_3, apart from the symbols associated with o_1 and o_2, and o_2, o_3, let other symbols be associated only with either $o1$ or o_3, but not with both and not with $o2$. This would mean that we need to create four index trees, for o_1 and o_2, for o_2 and o_3, for o_1, and for o_3. However, for a binding event involving o_2 and o_3, in order to provide efficient access to previously created monitors associated with o_2, we need to have an extra index tree corresponding to only o_2.

Let Ψ''_ϕ be the additional index trees that will be created beforehand for objects that are involved in binding events. Here, $\Psi'_\phi \cap \Psi''_\phi = \emptyset$, and $\Psi_\phi = \Psi'_\phi \cup \Psi''_\phi$. In other words, Ψ_ϕ will be partitioned into Ψ'_ϕ and Ψ''_ϕ. Formally,

$$\Psi_\phi = \begin{cases} \Psi'_\phi, & \rho(\phi) \\ \Psi'_\phi \cup \Psi''_\phi, & \text{otherwise} \end{cases}$$

and,

$$\Psi'_\phi = \{\psi : \exists u : (u \neq \emptyset) \wedge (u \in L) \wedge (u \in T(\{\psi\})) \wedge (\neg\exists \psi_1 \in \Psi'_\phi : u \in T(\{\psi_1\})\}$$

Let $Type$ be the set of types of all objects involved in the property ϕ and let $T : 2^{\Psi_\phi} \rightarrow 2^{2^{Type}}$ be a function that takes a set of index trees and returns a set of sets of types of objects that may form keys for the trees. Then, Ψ''_ϕ can be defined as follows.

$$\Psi'' = \begin{cases} \emptyset & \text{no binding events} \\ \{\psi : \exists u \exists i \exists j : (u \neq \emptyset) \wedge (\tau_{b_j} \cap \tau_{b_i} \neq (\tau_{b_i} \vee \tau_{b_j})) & \text{otherwise} \\ \wedge (u \subseteq \tau_{b_i} \wedge u \subset \tau_{b_j}) \wedge (\neg \exists u' : u \subset u' \wedge u' \subseteq \tau_{b_i} \wedge u' \subset \tau_{b_j}) & \\ \wedge u \in T(\{\psi\}) \wedge (\neg \exists \psi_1 \in \Psi''_\phi : u \in T(\{\psi_1\}))\} & \end{cases}$$

Theorem 3.3.1 *Following inequality must hold.*
$|L| \leq |\Psi_\phi| \leq \sum_{i=1}^{|L|} 2^{|i|}$.

Proof 3.3.1 *We need minimum index trees when all related objects are involved in a creation event. That is $\Psi_\phi = \Psi'_\phi$ when we have $\rho(\phi)$. In this case, we already have shown a bijection $h : L \rightarrowtail\!\!\!\!\!\rightarrow \Psi_\phi$. This shows that, we need at least $|L|$ index trees.*

On the other hand, when we have binding events, we will need additional Ψ''_ϕ index trees as per the definition of Ψ''_ϕ.

In the worst case, every subset u of an element of L may have a corresponding index tree. Hence, this number cannot be larger than $\sum_{i=1}^{|L|} 2^{|i|}$. In practice, we expect $|\Psi''_\phi|$ to be much smaller than this worst case.

Regardless of the type of event, object-based monitoring incurs the cost of retrieving object maps, and eventually a set of monitors, given an event and a set of index maps; we denote the number of maps accessed in processing an event with a set of index maps with $H : E \times 2^{\Psi_\phi} \rightarrow \mathbb{N}$. The cost of retrieving a value, c_r, from each such map is incurred when processing each event. If a map that is to be accessed does not exist, it needs to be created. The cost of adding an entry to a map is c_{a_p} and that of creating a map is c_{c_p}. This additional

cost of map entry insertion is applicable to both object-based and state-based monitoring and is shown using function $\kappa : \Upsilon \rightarrow \mathbb{R}$ which is a function that takes a map and outputs a cost of adding an entry to it, if required. The map will be created if required. Formally,

$$\kappa(\upsilon) = \begin{cases} 0, & \text{map entry exists} \\ c_{a_p}, & \text{map exists} \\ c_{c_p} + c_{a_p}, & \text{otherwise} \end{cases}$$

In state-based monitoring, no cost is incurred if indexing, dx, is disabled. When indexing is used, the state-based monitoring incurs cost for each state in ϕ.

For each state, s, processing $e_i = (l_i, b_i)$ involves accessing the set of monitors associated with the l_i objects and moving them to the successor state, i.e., $\delta(s, b_i)$. For logical separation, we show this cost of moving monitors to the successor state as a part of cost incurred during performing a transition.

Unlike object-based monitoring, here each state has an index map, $\alpha : S \rightarrow \Psi_\phi$. For each object in e_i that can be used to perform lookups in the state's index map the cost of a map reference is incurred. Here $K : 2^O \times \Psi_\phi \rightarrow \mathbb{N}$ takes a set of objects and an index tree and outputs the number of those objects that are keys in the index tree. In addition to map reference costs, a cost is incurred for insertion into each map referenced in the successor state.

3.3.3 C_I: Inserting Monitors

In Figure 2.1, this cost is incurred in performing step 3. For object-based monitoring, processing a creation event, or any event when suffix matching is enabled, results in the new monitor being inserted into each of the index maps; the cost of the insertion is c_{a_m}.

$$C_I = \begin{cases} |\Psi_\phi| * c_{a_m} & e_i \in E_c \vee (sm \wedge \neg E_b) \\ |G(l_i, \Psi_\phi)| * c_{a_m} & e_i \in E_b \\ 0 & \text{otherwise} \end{cases}$$

$$C_I = \begin{cases} c_{a_m} & e_i \in E_c \vee (sm \wedge \neg E_b) \\ \sum_{s \in S} J(l_{i_b}, s, \pi_i) * c_{a_m} & e_i \in E_b \\ 0 & \text{otherwise} \end{cases}$$

Figure 3.11: C_I: Cost of inserting monitors into pools. Object-based monitoring (Top) and State-based monitoring (Bottom).

However, the cost changes for a binding event as monitors as many as the cloned ones need to be inserted in the pools of relevant object sets. For this, we define $G : 2^O \times 2^{\Psi_\phi} \rightarrow 2^{\Psi_\phi}$ a function that takes as arguments a set of objects l, a set of index trees Ψ, and outputs a subset of Ψ, for which a subset of objects l' of l that includes at least one element from $l_{\bar{b}}$, forms a valid key. Formally,

$G(l, \Psi_\phi) = \{\psi : \exists l' : (l' \subset l) \wedge (l' \cap l_{\bar{b}} \neq \emptyset) \wedge key(l', \Psi_\phi)\}$, where $key : 2^O \times \Psi \rightarrow 2^O$ be a function that takes as arguments a set of objects and an index tree and outputs the subset of the set of objects that may form a key to the index tree.

Function G would give us the number of index trees that will be accessed to retrieve the the monitor pools in which the new monitor will be inserted.

For state-driven monitoring, insertion is simple for the case of creation event or suffix matching since the monitor is simply inserted into the map corresponding to the FSA start state.

For the case of binding events, we define $J : 2^O \times S \times \{\pi_i\} \rightarrow 2^M$, a function that takes as arguments a set of objects, a state and a prefix of the trace π_i and outputs the set of monitors associated with the set of objects in that state at a time when the monitored program generated σ_i. In state-driven monitoring there could only be one index tree at a

state if indexing is provided. In either case, monitors as many as the cloned ones need to be inserted in a pool of monitors.

3.3.4 C_V: Traversing Monitors

$$C_V = \begin{cases} 0 & e_i \in E_c \wedge \neg sm \\ \theta(l_i, \pi_i) * (c_v + c_e) & sm \\ \theta(l_i, \pi_i) * c_v & \text{otherwise} \end{cases}$$

$$C_V = \begin{cases} 0 & b_i \in \Sigma_c \\ \sum_{s \in S} \lambda(s, b_i) * \eta(s, l_i, \pi_i) * & dx \wedge\ K(l_i, \alpha(s) \geq 0) \\ \quad ((|l_i| - K(l_i, \alpha(s))) * c_e + c_v) & \\ \sum_{s \in S} (\lambda(s, b_i) * \beta(s, \sigma_i) * (|l_i| * c_e + c_v) + & dx \wedge\ K(l_i, \alpha(s)) = 0 \\ |keys(\delta(s, b_i), \pi_i)| * c_v) & \\ \sum_{s \in S} \lambda(s, b_i) * \beta(s, \sigma_i) * (|l_i| * c_e + c_v) & \text{otherwise} \end{cases}$$

Figure 3.12: C_V: Cost of traversing monitor pools. Object-based monitoring (Top) and State-based monitoring (Bottom).

Accessing the collection of monitors associated with an event, through the index and object maps, is a fundamental component of processing an event. In Figure 2.1, this cost is incurred in performing step 4.

For object-based monitoring, when handling a creation event corresponding to a creation symbol, there are no pre-existing monitors related to the event and thus no traversal cost. For other events, we must account for the fact that the monitors associated with an event will change throughout the trace. Consequently, we define $\theta : 2^O \times \{\pi_i\} \rightarrow \mathbb{N}$ to be the number of monitors associated with a set of objects at a time when the monitored program generated π_i. Each monitor is accessed in turn where the cost of an individual access is c_v. When suffix matching is enabled, there is an extra cost for comparing the current

state of the newly created monitor with the current states of the existing ones and keeping the newly created monitor only if its current state is different than the current states of all others.

For state-based monitoring, creation events corresponding to creation symbols, also result in no monitor traversal. For other events and regardless of whether indexing is performed, as discussed above, each state is processed independently. Moreover, processing in each state is controlled by whether the event causes a state transition. If $s = \delta(s, b_i)$ then no monitors need to be moved; this is captured by the $\lambda(s, b_i) = (s = \delta(s, b_i)\ ?\ 0 : 1)$ multiplier. $\eta : S \times 2^O \times \{\pi_i\} \rightarrow \mathbb{N}$ is the state-specific analog of θ for object-based monitoring. It returns the number of monitors associated with the given objects in the given state at a point in the trace. With indexing enabled, equality tests on monitors, which cost c_e, are performed on each non-key object in the event; without indexing the tests are needed for all objects. In either case, a monitor access is performed, which costs c_v.

However, the cost is higher when indexing is not supported. In this case, all monitors are traversed at every state to see if any associated monitors exist. For this, we define function $\beta : S \times \{\sigma_i\} \rightarrow \mathbb{N}$ that takes as arguments a state and a prefix of the trace σ_i and outputs the number of monitors in that state at a time when the monitored program generated π_i.

The cost is highest, when indexing is enabled but is not applicable to any of the objects in l_i. In this case, monitor traversal is performed by accessing all *key-value* pairs to find the relevant monitors. Hence, there is an extra cost for traversal of all keys. This is expressed using the function $keys : \Psi \times \{\pi_i\} \rightarrow 2^O$ that returns all keys that belong to the index tree.

$$C_T = \begin{cases} 0 & e_i \in E_c \\ \theta(l_i, \pi_i) * c_t & \text{otherwise} \end{cases}$$

$$C_T = \begin{cases} 0 & e_i \in E_c \\ \sum_{s \in S} (\eta(s, l_i, \pi_i) * \zeta(s, e_i)) & \text{otherwise} \end{cases}$$

Figure 3.13: C_T: Cost of performing transitions. Object-based monitoring (Top) and State-based monitoring (Bottom).

3.3.5 C_T: Performing Transitions

In Figure 2.1, this cost is incurred in performing step 4. For both object and state-based monitoring, creation events are initialized to an appropriate state, so no transitions are needed.

For non-creation events, object-based monitoring accesses each monitor for the event and performs a transition on the stored FSA state, which costs c_t.

In state-based monitoring, for each monitor associated with the event in the current state the cost may vary, depending on whether the monitor is required to be moved to another state and whether indexing is provided. That variation is captured by

$$\zeta(s, e_i) = \begin{cases} 0 & s = \delta(s, b_i) \\ c_{d_p} + |l_i| * c_{c_m} + H(e_i, \Psi_\phi) * c_r + \sum_{\upsilon \in H(e_i, \Psi)} \kappa(\upsilon) & dx \wedge \neg pf \\ c_{d_p} & dx \wedge pf \\ c_{d_m} & \neg dx \wedge pf \\ c_{d_m} + |l_i| * c_{c_m} + c_{a_m} & \neg dx \wedge \neg pf \end{cases}$$

where c_{d_m} and c_{a_m} are the costs of deleting and adding a monitor to the pools, respectively and c_{d_p} is the cost of deleting an entry from a map. When indexing is provided and

transition is performed, a new monitor is created and is deleted from the index tree at the source state and is added to the index tree at the target state. In the absence of indexing the monitor is deleted from the pool at the source state and added at the target state. Note that the term corresponding to monitor creation would disappear from the definition, if a monitor is reused and not created. However, for efficient monitor retrieval, a monitor may be created every time a state is transitioned to implement certain optimizations. For example, in Tracematches, a disjunct that is equivalent to a monitor is arranged in the memory in a way that allows an optimization related to memory leaks. However, this support comes at an extra cost of recreating a disjunct every time it is moved to a different state.

For suffix matching, a monitor will be deleted from the pool if pattern fails, that is when pf is enabled.

3.3.6 Comparing operational costs

We compare the costs of various monitoring operations and summarize them in Table 3.2. This comparison should highlight the differences in the costs due to the basic operational differences in the two approaches. For simplicity, we do not consider the binding events for this comparison. The comparison is only with respect to the creation and the transition events.

3.4 Reexamining unexplained variation

Using the cost models, we revisit the results in Table 3.1 to explain some of the observed differences.

Clearly, changes in the program, input, or property may give rise to changes in the length of the trace and the number of monitors created during the trace. For example, reducing the size of the input between rows 1 and 2 drops the number of events from 80

Cost	Object-based Monitoring	State-based Monitoring
C_C	This cost is similar in both cases.	
C_R	This cost would normally be higher than the cost incurred by state-based monitoring as access to monitors would always be provided using indexing on related objects. So this operation would involve accessing an indexing tree using all related objects as a key. However, this relatively higher cost of manipulating index trees results in much higher saving in the cost of monitor pool traversal as it gives direct access to only associated.	This cost would be 0 if no indexing is supported, or would be equal to or lower than the cost incurred by the object-based monitoring depending on whether index keys are formed by all related objects or only a subset of them.
C_I	This cost would be proportional to the number of index trees.	This cost would normally be that of inserting a monitor in one index tree or in the pool.
C_V	This cost would be proportional to the number of associated monitors as only relevant monitors would be accessed due to indexing. For creation events, this cost is 0.	This cost would be very high if no indexing is provided as the entire monitor pool would be traversed. It would also be very high if indexing is provided for objects other than that involved in the event. The cost would be low to high depending on whether full or partial indexing support is provided. This cost would be 0 for creation events as well as self-looping transitions.
C_T	This cost would be 0 for creation events and proportional to the number of associated monitors for transition events. However, the cost of performing a transition on a monitor is cheap.	This cost would be 0 for creation events and self-looping transitions. However, this cost would be proportional to the number of associated monitors. However, the cost of performing a transition on a monitor would be high as a monitor is moved from the monitor pool of one state to another. The cost would be even higher if indexing is provided as that would mean manipulation of index keys.

Table 3.2: Comparing operational costs.

million to 1.8 million and the number of monitors from 4 million to 340 thousand. This leads to the ten-fold reduction in overhead. This reduction is mainly due to the reduction in the total number of events. The overall cost of monitoring is sum of the costs of handling all individual events.

The variation between rows 6 and 7 is due to the disabling of indexing that rapidly increases C_V. In this case, indexing is performed on an iterator object. The hierarchy of index maps is just a single layer, thus the number of map operations, $K()$ in C_R, is relatively small. More importantly, more than 90% of the events are `next` and these self-loop in state 1 of Figure 3.2. Thus, the cost of processing state 1 in the summation of the second term of C_V is low and for the other states $K()$ serves to drive down monitor traversal costs.

The relatively low overhead of JavaMOP in row 5 is also due to indexing. Unlike Tracematches, JavaMOP provides indexing for all symbols making traversal dependent only on the number of monitors and not on the intermediate object maps as shown in C_V and C_T. Row 3 varies just the program and the overhead jumps to 3291.7%. This is due to the fact that bloat incurs more than 80 million events and in suffix mode, creates over 160 million monitors, whereas pmd creates just over 4.4 million monitors and incurs about 2.2 million events.

Rows 8 and 15 highlight the effect of monitoring the property `HasNext` which tends to flip-flop monitor states. For both tools, the cost of C_C and C_I are dominant since both approaches create monitors in suffix mode for every symbol seen. Moreover, for this property Tracematches is able to fully index the single object involved which keeps C_V low. The difference between tools is due to the cost of C_T which is higher for Tracematches since states are changing continually, and the ζ term's first case is not active. Since a single monitor is associated with any iterator object at any time, $\eta()$ in the definition of C_T is low. Hence, the performance of the tools would mainly depend on how efficiently JavaMOP deals with the large numbers of monitors.

When the number of events grows large enough, as shown in rows 10 and 12, Tracematches can outperform JavaMOP. In suffix mode all events produce new monitors, bloat generates more than 80 million events leading to 160 million monitors. We believe their is room for improving the efficiency of suffix matching in JavaMOP since, as shown in row 11, the same monitoring using complete matching mode incurs a much smaller overhead of 172.9%.

Although not comprehensive, this examination of unexplained variations in the light of the cost models illustrates the different dimensions they capture and provide glimpse of their explanatory potential. We explore this potential further in the next section.

3.5 Partial Validation of the Cost Models

The cost models presented in Section 3.3 are based on our understanding of modern monitoring tools, mainly JavaMOP and Tracematches. The question that we would like address at this point is whether the key factors highlighted by the models indeed influence the cost of monitoring as expressed by the models. In order to answer this question convincingly, we will have to perform a complete validation of the models. However, a complete validation of both object-based as well as state-based monitoring tools would be a big effort and may take a few months. Nonetheless, it would help increase our confidence in the cost models and we plan to do that in future.

In this section, we present the results of our profiling studies performed in order to understand the influence of one of the key factors identified by the models on the cost of monitoring. Our inspection of the models shows that the number of monitors associated with the objects is a factor that may have a big influence on the monitoring overhead as it appears as a dominant term in the definitions of both, C_V and C_T as shown in Figures 3.7 and 3.8. The terms corresponding to θ, η and β reflect the number of monitors associated

with the objects. Clearly, they influence the costs of monitor traversals and transitions. This factor does not affect C_C, C_R and C_I. Hence, keeping other factors unchanged would keep the other costs including C_C, C_R and C_I unchanged. Hence, by controlling the number of associated monitors and the number of events, one at a time, we should be able to observe the effect of the number of monitors on the total cost of monitoring.

In order to validate the effect of the number of associated monitors, we performed two small profiling studies using JavaMOP and Tracematches. In the first study, we used JavaMOP to understand the effect of the number of associated monitors on object-based monitoring. We selected `FailSafeIter` property for monitoring because it is a multi-object property that could reveal the effect of the number of associated monitors on handling events.

We developed small Java programs [2] that create and perform set and iterator operations. The variables that we considered for this study were the number of monitors associated with receiver objects and the number of events. We analyzed the change in execution times by varying values of variables one at a time and keeping the remaining variables unchanged. In order to understand their effect in isolation, we divided the events into two types. The first type includes `update` events that would lead to multiple transitions, whereas the second includes mainly `next` events that may trigger only one transition. The second type also includes those `update` events that are guaranteed to cause only one transition. These events correspond to the update events on sets that do not have associated iterators, but have only a default monitor. We analyzed the results in the context of events of exactly one type at a time. The results of this study are presented in Table 3.3.

The first column in Table 3.3 corresponds to the variable, the effect of which we are analyzing. The column also provides the number of monitors created and the number of events generated during the execution. The second column specifies the event type that indicates

[2] All the programs used in this study have been provided in Appendixes A and B

Variable	Event Type	Mons/Evts	Scaling Factor (SF)				
			1	10	100	1000	10000
Monitors	uni	$100 \times SF, 10^8$	64	71.4	155.1	268	252.8
	multi	$10 \times SF, 10^6$	12.8	25.9	249.6	7275.7	83095.7
Events	uni	$10^4, 10^4 \times SF$	7.7	12.4	67.8	172.2	165.2
	multi	$10^4, 100 \times SF$	16.2	151.6	1045.7	5058.6	8566.7

Table 3.3: JavaMOP: Overhead due to events and monitors associated with receiver objects (Time as relative overhead).

whether the events would trigger multiple transitions. The type *multi* represents *multi-monitor* events that trigger multiple transitions and the type *uni* represents *uni-monitor* events that trigger exactly one transition. The scaling factor corresponds to the factor of increase in the value of the parameter that is varied. For example, scaling factor of 100 corresponding to events would mean that the events generated by the monitored program are 100 times that generated by the base case corresponding to the scaling factor 1. Similarly, scaling factor of 10 on monitors associated with receiver objects would mean that, the number of monitors associated with receiver objects is 10 times larger compared to the base case. The figures represent relative overhead, i.e., a factor by which the execution time reported by the instrumented version of the program compares with the execution time reported by the uninstrumented version of the program.

The row corresponding to the number of monitors associated with receiver objects and uni-monitor property events, indicates that the overhead factor increases logarithmically from the scaling factor 10 to 1000 and then does not increase much for the scaling factor 10000. However, the overhead factor increases very rapidly in the case of multi-monitor events, as indicated by the next row. This shows that with the increasing number of associated monitors, the time required to process each event starts increasing rapidly. Hence, the number of associated monitors has a great impact on the overhead when observed events are of multi-monitor type.

The next rows corresponding to the number of events show a similar pattern about the overhead factor. For uni-monitor type, the overhead factor increases but only steadily. However, for multi-monitor type, it increases rapidly. In other words, this indicates that compared to multi-monitor events, uni-monitor events are cheaper to process. It should be noted that the last two columns corresponding to scaling factors 1000 and 10000 show no increase in the overhead factor for uni-monitor properties in both cases.

These results show that the number of monitors associated with the objects influences the cost of monitoring for object-based monitoring tools. In order to understand its effect on state-based monitoring tools, we used Tracematches and two synthetic properties that, in addition to the effect of number of associated monitors, would also reveal the difference in the way Tracematches handles looping transitions and non-looping transitions. Remember that we used function λ and ζ, to distinguish between costs associated with looping and non-looping events. In order to isolate the effect of the number of associated monitors on the costs of monitor traversal and transitions, we used Tracematches annotations for indexing. We used the synthetic properties, because we wanted a multi-object property with looping as well as self-looping transitions. Property `HasNext` has a looping structure, but it is not a multi-object property, whereas property `FailSafeIter` is a multi-object property, but it does not have a looping structure. In addition, we wanted loops as well as self-loops to be performed by transitions that are associated with single and also with multiple monitors to show their effects in the case of state-based monitoring. The synthetic properties allow us to have a control over all of these attributes.

Note that the properties that we have built here do not capture real API rules, but are built for their structure, i.e. either looping or self-looping characteristics, and multi and uni-object nature that reveal interesting behavior of monitoring tools. In practice, the behavior of a monitoring tool would depend on these characteristics of a property as well as its

Variable	Event Type	Scaling Factor (SF)				
		1	10	100	1000	10000
Monitors						
($\#mon = 1 \times SF$	uni	26.32	23.48	23.24	24.6	24.25
$\#evt = 2 \times 10^6$)	multi	286.44	324.33	555.56	3931.31	87020.94
Events						
($\#mon = 10^4$	uni	25.5	23.0	35.33	19.22	23.8
$\#evt = 200 \times SF$)	multi	2243.0	2961.5	2773.88	17318.38	91858.13

Table 3.4: Tracematches (Property 1): Overhead due to events and monitors associated with receiver objects (Figures indicate relative overhead).

Variable	Event Type	Scaling Factor (SF)				
		1	10	100	1000	10000
Monitors						
($\#mon = 1 \times SF$	uni	7.05	6.24	6.43	6.85	6.55
$\#evt = 10^6$)	multi	272.69	284.47	269.19	229.88	266.44
Events						
($\#mon = 10^4$	uni	21	26.5	20.33	7.67	14.52
$\#evt = 100 \times SF$)	multi	2125.5	2140.5	1472.67	532.12	264.88

Table 3.5: Tracematches (Property 2): Overhead due to events and monitors associated with receiver objects (Figures indicate relative overhead).

interaction with a program. In our study we control program-property interaction to observe the monitoring tool behavior.

Table 3.4 presents the results of monitoring property `create ((update empty) | (hasNext next))+ empty`. We refer to this property as `Property 1`. Figure 3.14 shows an FSA corresponding to this property. Symbol `empty` corresponds to the method call `isEmpty` on a collection. This property has alternating states, and the programs being monitored are written in such a way that the monitors would continuously change their states. Moreover, we can control the loop that we want to execute. The loop formed by `(update empty)` can be executed with the help of a collection object that is associated with multiple monitors. On the other hand, the loop formed by `(hasNext next)` can be executed with the help of an iterator object that is associated with a single

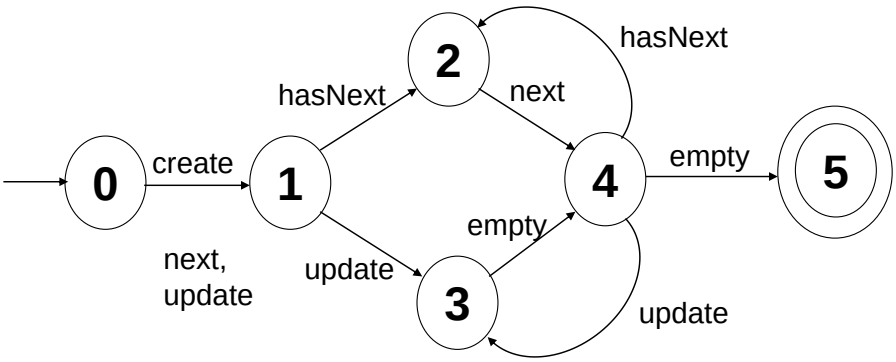


Figure 3.14: Property 1: `create ((update empty) | (hasNext next))+ empty`

monitor. Tracematches incurs heavy cost for traversing and transitioning numerous monitors as seen by the events that are of multi-monitor type. The uni-monitor type events are associated with only one object, hence although there is state-transitioning, the overhead is not too high as work performed in traversing a monitor list and performing a transition over one monitor is much smaller than the previous case. These costs are captured in the cost models by C_V and C_T, which are high when the number of associated monitors (β and η) is large. Note that, accessing the associated monitors directly is possible because of the indexing option that we used.

Table 3.5 presents the results of monitoring property `create (update | next)+ empty`. We refer to this property as `Property 2`. Figure 3.15 shows an FSA corresponding to this property. This property does not have continuously changing states. Instead it has a self-loop. As in the previous case, we can control the self-loop that we want to execute. The loop formed by `update` can be executed with the help of a collection object that is associated with multiple monitors. On the other hand, the loop formed by `next` can be executed with the help of an iterator object that is associated with

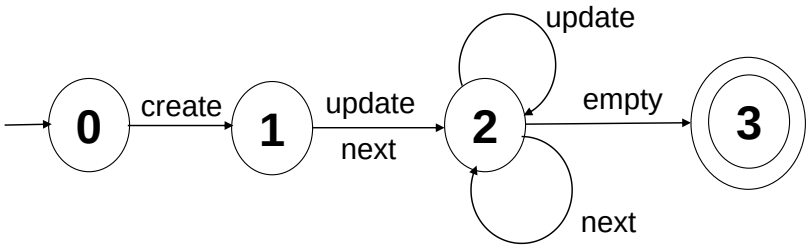


Figure 3.15: Property 2: `create (update | next)+ empty`

a single monitor. As indicated by the cost models, Tracematches efficiently handles the self-looping transitions and hence, the multi-monitor events are handled efficiently. The overhead figures are with respect to the execution time of the uninstrumented version of a program. The decrease in the overhead is because of increase in the execution time of the uninstrumented program version. The actual time taken by the instrumented version remains almost constant irrespective of the scaling factor. These costs are captured in the cost models by the function λ and the condition $s = \delta(s, b_i)$ for C_V and C_T. This shows that Tracematches is insensitive to the number of associated monitors (β and η) when it comes to handling self-loops.

Although these preliminary studies target only one key cost factor which is the number of associated monitors, they help increase our confidence in the cost models. More studies that target other factors would provide additional evidence for the claim that the cost models highlight influential factors and explain how they impact the runtime overhead.

Program	Complete (JavaMop)		Suffix (JavaMop)		Suffix (Tracematches) `next` indexed	
	Unopt.	Opt.	Unopt.	Opt.	Unopt.	Opt.
bloat	173	73	2533	340	1169	115
pmd	15	7	74	11	91	4

Table 3.6: Property `HastNext` - Overheads are expressed in percentages over the original uninstrumented versions (4.8s for bloat and 4.6s for pmd with a default input load). Optimization corresponds to the loop transformation to reduce the number of monitored events.

Program	Complete (JavaMop)		Suffix (JavaMop)		Suffix (Tracematches) `next` indexed		Suffix (Tracematches) `update` indexed	
	Unopt.	Opt.	Unopt.	Opt.	Unopt.	Opt.	Unopt.	Opt.
bloat	702	298	3292	410	2492217	2939060	*	*
pmd	57	9	87	11	9120	3754	28883	4759

Table 3.7: Property `FailSafeIter` - Overheads are expressed in percentages over the original uninstrumented versions (4.8s for bloat and 4.6s for pmd with a default input load). Optimization corresponds to the loop transformation to reduce the number of monitored events. * Indicates that the process was stopped after 172800 seconds and overhead of 3599900.

3.6 Revisiting Previous Findings

In this section, we look at some of the previously published techniques and studies from the fresh perspective provided by the cost models. We use the model to help us understand the scope and generalizability potential of one of our own techniques, to validate and investigate conjectures about sources of overhead, and to reveal missing optimization opportunities.

3.6.1 Techniques' scope and generalization

Recently we introduced a program transformation technique to reduce runtime overhead caused by events occurring in loops [68]. The key insight behind the technique is that a monitor must observe a few iterations of the loop in order to judge whether the program violates or satisfies the property, but it can effectively ignore the rest of the loop iterations. The program transformation then enables this discrimination between relevant and irrelevant iterations. The technique was very effective at reducing the number of generated events. Yet, the results were unsatisfying on two levels.

First, for several cases, the reduction in the number of events did not translate into significantly smaller overheads. When mapping this technique to the cost models it is evident that it attacks some terms but not others. In particular, the costs of monitor traversal and update, C_T and C_V, are not considered by this technique. As these costs become dominant, the overhead reduction from cutting down the number of observed events becomes inconsequential. For example, for property `FailSafeIter`, handling `update` is usually much more expensive than handling `next` events as the former events are associated with collection objects. This means that the per iteration cost that is saved by transforming loops that contain `next` events may be overcome by the cost of state traversal and changes caused by `update` events. This type of analysis would not be possible without the cost models which provide insights into the scope of effectiveness of our proposed loop optimization technique.

Second, our results were from an object-based approach and it is not obvious what would happen with a state-based monitoring approach. For a program where loops are dominant, when the proposed technique is used with an object-based approach, the cost of monitoring would depend primarily on the number of monitors associated with the observed objects, where more monitors imply a higher cost of C_V and C_T. For state-based

approaches, except for C_R and C_V, all the costs become negligible. If indexing is available, then C_R would be low and C_V would also be low since the technique would ensure that most of the monitors would move to self-looping states that generate no events after a few iterations. In the absence of indexing, however, the number of monitors at non-self-looping states would decide the cost of C_V. Overall, since the object-based approach would be independent of the monitors' states, we would expect for its performance to be more consistent when other variables change.

In order to validate the last conjecture hinted by mapping the technique to the cost models, we reproduced part of the study originally performed on this technique but considering both object-based (JavaMOP) and state-based (Tracematches) approaches. Tables 3.6 and 3.7 show the results of monitoring the `HastNext` and `FailSafeIter` properties on the bloat and pmd programs from the Dacapo benchmark using different types of matching and indexing to explore the variations in C_R and C_V. We see that the overhead differences between the unoptimized and optimized versions under JavaMOP range from about 54% to 88%, while for Tracematches they vary from about 15% to about 95%. This shows that the benefits for JavaMOP are more consistent than benefits for Tracematches. Comparatively smaller reduction for JavaMOP for property `HasNext` could be due to the dominant cost of C_C, C_I and C_R as our loop optimization technique only targets the number transitions and not the number of monitors which is extremely large in this case. Overall, the results confirm the intuition provided by the cost model: the proposed technique generalizes to state-based approaches, but the gains in performance will exhibit more variability depending on then number of monitors and their states.

3.6.2 Conjectures about sources of overhead

Meredith et al. [62] reported that PQL and Tracematches overheads are more susceptible to the complexity of the monitored property than JavaMOP. Although complexity is not explicitly defined in their work, we derived from the text that it means the property has more symbols and operators. When trying to map this conjecture to the cost models we find limited support for it, that is, there was not a way for the cost models to support this conjecture. This lead to the investigation of other potential sources of overhead.

We start by mapping the transition cost, C_T, from the cost model for the state-based approach. When compared with the object-based approach, this cost for state-based is high if it requires changing the state of the monitor, but low if the transition is a self-loop over the property automaton. So, based on analysis performed using the cost model it is conceivable that a large property would work better for state-based monitoring if there is an abundance of self-loops and scarce monitor transitions. Although there are examples of those properties, we also find at least one property with these attributes in the cited study, `FailSafeIter`, that incurs a high overhead in Tracematches compared to JavaMOP. So we set out to analyze another potential factor that may explain this result.

The second term we analyze is C_V. In the presence of a large number of monitors the cost of monitor traversal can be high, particularly if no indexing is provided. On the other hand, it would be prohibitively expensive for a state-based tool to provide full indexing support for all symbols (as monitors are moved after a transition, index maps at the source and target state must be updated). That is the reason for state-based tools to provide very limited indexing (e.g., for a subset of objects). Given this insight about the potential impact of C_V due to indexing, we briefly studied the overhead produced by state-based versus object-based monitoring with the indexing factor controlled. In order to do this, we developed an object-based monitor using JavaMOP, and then used a copy of it as

Property	bloat		pmd	
	Object	State	Object	State
FailSafeIter	702	577	43	20
HastNext	354	808	17	41

Table 3.8: Percentage of overhead across program-property pairs when running object- and state-based monitors in complete-matching mode.

a template from which we implemented a state-based monitor that kept the same indexing structure. We then run those implementations on bloat and pmd while monitoring the `FailSafeIter` and `HasNext` properties, considered complex and simple respectively in the cited study.

Table 3.8 shows the results of running the two monitors in complete-matching mode. When looking at `FailSafeIter` row, we see that the trend is not uniform. For the bloat and `FailSafeIter` combination, the state-based monitor performs better than object-based monitor, however, the trend does not hold for pmd. So, more complex properties in terms of size do not necessarily imply greater overhead. Furthermore, as we observe the second row of the table for the much simpler `HasNext` property, we note that object-based monitoring is consistently better. The reason is that this property causes a monitor to change state, which we have established as detrimental for the state-based approach. These results reveal that attributing overhead differences to property complexity in terms of size is insufficient. The reasons, as explained by the cost models, are much more subtle and cross-cutting.

3.6.3 Missed optimizations opportunities

To address the general indexing weakness of state-based approaches, Avgustinov et al. introduced an optimization technique for Tracematches that analyzes a property to decide what symbol should be indexed at a state [8]. Apart from a default scheme that identifies

objects for indexing based on some heuristics, the technique can be configured by the user to specify what symbols may occur more frequently and should be prioritized for indexing. In general, leveraging user domain knowledge seems like a good idea. However, as we have already seen, the factors causing the overhead may interact in subtle and unexpected ways that even an expert user may not predict. In this particular case, the frequency of a symbol clearly matters but its ability to change many states may have a significant influence as well. So, for example, if a highly frequent symbol self-loops over a state in which most of the monitors are to be found at any time, then choosing that symbol for indexing may not be so beneficial, as those monitors would anyway be processed efficiently. Similarly, choosing a symbol that self-loops in most of the states would not be beneficial. For such a symbol, the costs C_V and C_T would be zero for the self-looping states.

For illustration purposes, the last four columns of Table 3.7 show monitoring overheads for bloat and pmd for the property `FailSafeIter`, when indexing is performed on `next` and on `update`. Indexing `next` is clearly a better choice because `next` events are about 35 times more frequently than `update` events, and both events self-loop in most of the states. However, if a program generates a similar number of events for `update` and `next`, and if one of the two symbols was continuously state-changing (like `next` in the property `HasNext`), then identifying that symbol for indexing would be a better choice. Clearly, a heuristic based on the combination of these factors would lead to more complete optimizations. In general, we note that mapping algorithmic runtime monitoring techniques to cost models helps to explain how the technique works, when it may perform more effectively, and what other opportunities for optimization remain to be incorporated.

3.7 Evolution of Cost Models

We have discussed the most relevant related work on runtime monitoring in previous sections. Here we identify some broader related themes from the literature and how we plan to further evolve our cost modeling work to support prediction in addition to explanation.

Our cost models bear some resemblance to algorithmic complexity models except that our goal is to capture the effects of a number of variants of two well-established techniques for runtime monitoring of FSA properties. In general, algorithmic complexity characterizations focus exclusively on the dominant term that governs performance, but we believe that approach to be too coarse to allow for insights into how to make monitoring less costly for certain classes of properties.

One inspiration for our approach are the detailed complexity models for sorting algorithms on the CM-2 machine which capture a number of parameters of the machine and its interconnection network [15]. In this work, the authors were able to run a series of benchmarks to approximate their model parameters and then compare the models results with runtime data. We believe a similar approach would add value to our work in validating the fidelity of the model, however, several of the cost components we identify may vary significantly with the platform, e.g,, the JVM, OS, and architecture, so generalizing our findings across platforms presents a challenge. In addition, the functions in our cost models, e.g., H, θ, are very dependent on the program, input values, and property being monitored. It makes little sense to compute those values in one setting and using them for predictions in another setting, thus, we plan to focus on validating the explanatory power of our models.

One setting in which our models could be used for prediction is within a run of the program. Currently, the available runtime monitoring infrastructures configure monitoring offline. We believe that it is possible to build on the trend in modern JIT compilers [50] to perform "trace based" runtime monitoring. In such a setting, our models could be cal-

ibrated during the early parts of a program execution and then used to predict the relative performance benefits of using, for example, object-based versus state-based monitoring, or specific indexing or matching approaches, and switch to the most beneficial monitoring configuration. Continuing predictions would then lead to the dynamic adaptation of the monitoring strategy to fit the characteristics of the "hot" program paths. Inherent in such an approach is the simultaneous use of distinct monitoring strategies for different properties. We are working to realize this type of adaptive trace-based monitoring strategy within a modern JVM.

Chapter 4

Residual Analysis

4.1 Introduction and Motivation

Our optimization techniques are guided by the cost models that we presented in Chapter 3. The cost models indicate that the techniques that target the number of events should be developed first as those techniques would reduce the number of terms in the summation of the cost models and would result in indirectly targeting all the operational costs associated with the optimized events. In particular, considering that this technique would mainly target the transition event, this would save the costs of retrieving monitors (C_R), inserting monitors (C_I), traversing a monitor list (C_V) and performing transitions (C_T). This would obviously reduce the total cost of monitoring. For example, the techniques that allow us to simply skip some events without compromising error reporting would save the entire cost of handling the skipped events. As a result, we have prioritized our research so as to develop the techniques that target the number of events first and the other key factors later. In this chapter and in Chapter 5, we provide two such techniques that target the number of events. We leave the development of techniques that target other factors as future work.

Ideally, software systems would be statically checked for compliance with a typestate property. In practice, however, scaling to large software systems and providing precise analysis results in the presence of aliasing, data, thread and context-sensitive program behavior is very difficult. In recent years, researchers have combined multiple techniques to dramatically improve the cost-effectiveness of static typestate analyses [11, 28, 29, 32, 40]. However, as we discussed in Section 2.5.1, it is challenging to develop a static analysis that is both, scalable and precise. Hence, most static analysis techniques produce false positives. In this chapter[1], we present a technique for exploiting the intermediate results of *inconclusive* static typestate analyses to reduce the overhead of monitoring a program for typestate conformance at runtime.

Figure 4.1 sketches the high-level architecture of our approach. Conceptually, one runs an existing static typestate analysis to check conformance ($\tilde{}$) of the program with a set of typestate properties represented as finite-state automata (FSA). The properties, e.g., property_i, for which the analysis is inconclusive are the subject of *residual analysis*. Residual analysis uses the call graph, control flow graphs, and data flow analysis information leftover from the static typestate analysis to identify *safe* regions of program behavior that cannot lead to property violations. The analysis *reformulates* the typestate property by calculating FSA transitions that are only ever exercised in safe regions and deleting them. In order to monitor typestate conformance, the dynamic component of the analysis then only exercises the transitions that are generated by the remaining instrumentation such as inst_i from the regions that were not identified as safe.

The contributions of this work are (a) presentation and definition of *residual dynamic typestate analysis* a novel approach to reducing the cost of dynamic analysis of path-oriented properties by exploiting static analysis results, (b) an algorithm for calculating

[1]This work is an extended version of our paper that appeared in the proceedings of ASE, 2007 [37]. In particular, we have added Section4.3 on the soundness and the completeness of the analysis.

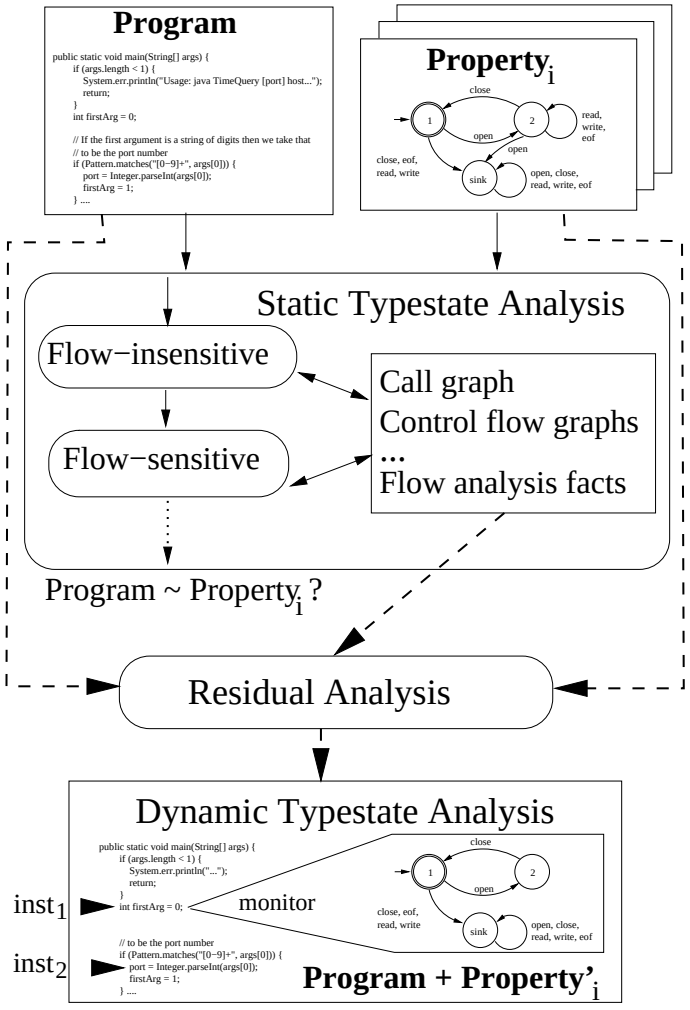


Figure 4.1: Residual Typestate Analysis Architecture.

regions of the program in which analysis instrumentation can be removed without impacting error reporting and using those regions to reformulate the typestate analysis problem, and (c) description of an implementation of the algorithm and experiences applying it to several applications that use non-trivial Java library APIs.

We begin with an overview of our technique in Section 4.2. Section 4.3 presents a discussion on the soundness and the completeness of the analysis. It identifies the conditions under which the analysis may produce unsound and incomplete results. Section 4.4

```
public class SocketChannel {
  static SocketChannel open() ...
  static SocketChannel open(SocketAddress ...
  SocketChannel connect(SocketAddress ...
  char read(ByteBuffer dst) ...
  int write(ByteBuffer src) ...
  final void close() ...
}

public void transformData() {
  SocketChannel sc;
  ByteBuffer buf;
  try { ...
    sc = SocketChannel.open();
    sc.connect(new InetSocketAddress(...));
    while (sc.read(buf) != -1) {
      ... transform buffer data ...
      sc.write(buf);
    }
  } catch (Exception e) { ...
  } finally {
     if (sc != null)
       sc.close();
  }
}
```

Figure 4.2: SocketChannel API Example

provides background and define new terms in order to set the stage for the presentation of the rationale for and details of our algorithm for calculating the residual dynamic analysis problem. Our experiences to date, which we describe in Section 4.5, provide promising evidence that residual dynamic analysis can significantly reduce the cost of monitoring for path properties at runtime.

4.2 Overview

We illustrate the principles of residual dynamic typestate analysis through an example. The top of Figure 4.2 sketches a portion of the `java.nio.channels.SocketChannel`

API; the `close` method is actually inherited, but we show it here for completeness. The documentation for this API defines a number of constraints on its proper usage. For example, calls to the `read` and `write` operations can only be performed on *connected* channels, channels should eventually be `close`d, and once closed, connections and I/O operations cannot be performed. Figure 4.3 illustrates an FSA that captures the typestate constraints for this API [2]. The states of this FSA capture abstract post-states of operations, for example, after a call to `open` (`connect`) returns normally the channel is considered *opened* (*connected*). Consequently, labels on transitions in this automaton represent the normal, i.e., non-exceptional, return from an API method call. We note that there are several operations in the `SocketChannel` API that are independent of typestate, e.g., `validOps`, which do not appear in the FSA. Furthermore, `open` is a factory method which generates a reference to an internal, hidden sub-type of `SocketChannel`, consequently each successfully returning call to `open` generates a new instance upon which successive calls are judged relative to the typestate FSA. Since opening and then connecting a channel is the common usage pattern, a convenience method `open(SocketAddress ...)` is provided.

The bottom of Figure 4.2 sketches a simple method that uses the `SocketChannel` API to read data, transform it and write it back to the sender. It `open`s and then `connect`s a channel, and then proceeds with a sequence of calls on the API to `read` the contents of the channel's data stream, transform the data, and `write` it back to the channel. Since calls on this API may throw a variety of exceptions they are embedded in `try-catch-finally` blocks. The finally block exhibits the common idiom of guarding calls to `close` a channel with a nullness test. Based on the typestate FSA, this code contains four points of potential failure (PPF) – the calls to `connect`, `read`, `write` and `close`.

[2] A complete set of typestate constraints would capture the interactions between `SocketChannel`s and other `java.nio` classes, such as `ServerSocketChannel`, and `java.net` classes, such as `Socket`. Our intention here is to capture the complexity of `SocketChannel` typestate, but we make no claim of completeness.

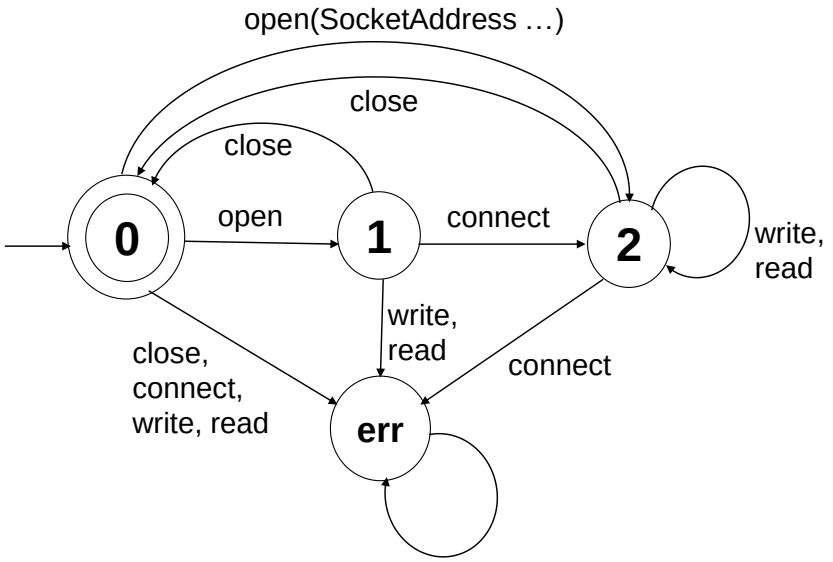


Figure 4.3: SocketChannel API Typestate FSA

To understand this code relative to the typestate constraints one can consider three equivalence classes of executions of this method. (1) The `open` call fails by throwing an exception. No instance of `SocketChannel` is created so no `close` call is needed, i.e., the typestate constraints are not active. (2) The `open` succeeds, but the `connect` call fails by throwing an exception. The channel will be `close`d in the finally block. (3) `open` and `connect` calls succeed and then the loop either terminates or a call to `read` or `write` throws an exception. The channel will be `close`d in the finally block.

From this analysis, it seems clear that even with the subtleties of exceptional return that are possible when using this API, the typestate constraints are obeyed by this code.

4.2.1 A Simple Static Typestate Analysis

A flow-sensitive static typestate analysis, for example, stages 2-4 in [40], propagates and updates information about the possible type states at different points along paths through

a program's source code. The left side of Figure 4.4 illustrates the control flow graph (CFG) for the example method; exceptional transfers of control are illustrated as dashed edges. In the Figure, edges in the CFG are annotated with sets of typestates that reflect the *cumulative effects* of the execution paths reaching those edges in terms of the typestate FSA. The analysis of the method begins in the start state, 1. The `open` call can either return normally, in which case the typestate FSA transitions to state 2, or exceptionally, in which case no state transition occurs. The `connect` call drives analogous state transitions based on normal or exceptional return. Once connected, the calls to `read` and `write` within the loop maintain a typestate of 3 regardless of normal or exceptional return. In the finally block, the exceptional and normal control flows merge at the entry of the `if (sc != null)` to form the set $\{1, 2, 3\}$. That set flows to the (implicit) return from the method, which is problematic since neither 2 nor 3 is an accept state, and to the `close` call which drives transitions to state 1, from 2 and 3, and to err, from 1.

This simple static typestate analysis is unable to confirm the satisfaction of the constraints encoded in the FSA. One way to resolve this is to integrate information about program data to form a *path-sensitive* typestate analysis. For the example, reasoning about the value of `sc != null` will improve the precision of typestate information accumulated along paths on which the `close` call will be executed. This could be achieved, for example, by propagating pairs of sets of typestates and predicate values. The right side of Figure 4.4 illustrates the results of such an analysis; some values on edges within the loop are elided. The `open` call would propagate the pair $(\{1\}, false)$ out of the exceptional control flow edge and $(\{2\}, true)$ out of the normal edge. Ultimately, this results in a flow into the finally block of $\{(\{1\}, false), (\{2, 3\}, true)\}$. At the conditional those pairs are filtered according to the predicate value and propagated along outgoing edges of the conditional. This results in the pair $(\{1\}, false)$ flowing along the false branch to the return and the value $(\{2, 3\}, true)$ flowing across the true branch to the `close` which transitions

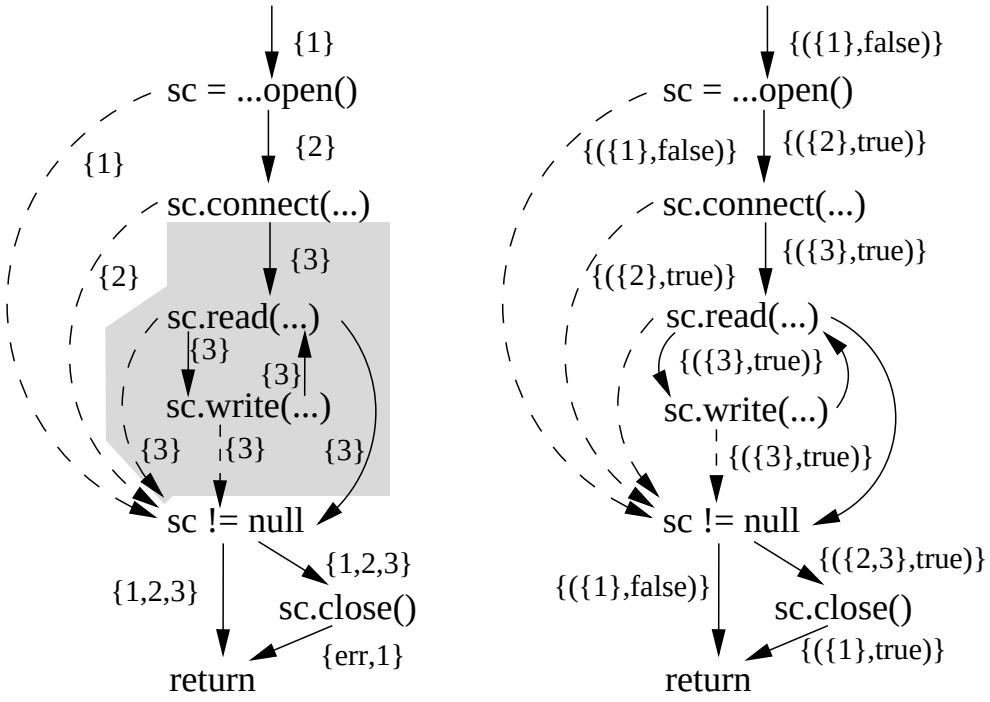


Figure 4.4: Example CFG and Static Typestate Analyses

both typestate values to $(\{1\}, true)$ on entry to the return. While incorporation of data context in this example is relatively simple, and effective in *conclusively* demonstrating typestate conformance, in general, enhancing a typestate analysis to achieve this kind of path-sensitivity can severely limit its scalability.

4.2.2 Leveraging Inconclusive Analysis Results

A traditional approach to dynamically analyzing typestate conformance would instrument the program to report the execution of each method call of the `SocketChannel` API that labels an edge in the FSA; essentially monitoring each PPF. For a program trace where the `open` and `connect` calls return normally and k buffers of data are `read` such an instrumented program will generate $3 + 2k$ *observations* to be processed by the dynamic analysis which steps through the FSA and reports an error whenever the err state is reached.

For programs that regularly call `transformData`, this may result in significant run-time overhead.

The simple static typestate on the left of Figure 4.4, while unable to confirm the typestate property, provides a rich source of information that can be used to reduce the cost of dynamic typestate analysis. Specifically, the gray region which begins with normal return from the `connect` call and ends with either the termination of the loop or an exception raised in either the `read` or `write` call has an interesting property. All program executions in the region have the same cumulative effect relative to the typestate automata; on entry to the region the typestate is 2 and on exit the typestate is 3. Furthermore, the `err` state is not reached within the region. A CFG region which is error-free and for which its cumulative effects can be summarized deterministically is termed *safe*.

The identification of safe regions lies at the heart of our proposed technique. We present an algorithm for calculating safe regions in Section 4.4. Based on the results of that algorithm, we can configure a dynamic analysis that elides all instrumentation within such regions and inserts a single *region observable event* that will trigger a typestate FSA transition to simulate the cumulative effects of the region. For the gray region in Figure 4.4, the automaton would be modified to introduce transitions from 2 to 3 on a new symbol, $region_{while}$, and transitions to the error state on that symbol from all other states. Then, every time the gray region is exited instrumentation will generate $region_{while}$, which will drive the new transition in the typestate automaton and thereby reflect the cumulative behavior of the path taken through the region. We note that in the special case where a region has no cumulative effect no such region event is needed; we call these *identity* regions.

The resulting dynamic typestate analysis is termed *residual* since it only reasons about the portions of the program that the static typestate analysis is unable to fully verify. A residual dynamic typestate analysis for the program trace described above will process 3 observable events : `open`, $region_{while}$, and `close`. Thus, even an imprecise static types-

tate analysis has the potential to significantly reduce dynamic analysis cost. The techniques described here are largely independent of the specifics of the static analysis. In general, a more precise static analysis, such as a path-sensitive analysis, will generate larger or more safe regions which will eliminate more instrumented events from the program. At present, we only consider relatively simple static typestate analysis techniques and leave the assessment of the benefit of more precise analyses to future work.

4.3 Soundness and Completeness

In general, our analysis produces correct results, which means that it is sound as well as complete with respect to the observed program behavior. However, the analysis may not produce correct results in the presence of unchecked exceptions if certain conditions are met. For example, Figure 4.5 shows a code snippet and let us assume that the property that is being monitored is `FailSafeIter`, which is shown in Figure 1.2.

For this property, statements 1, 2 and 3 are observables. Hence, they have been instrumented as shown using calls to corresponding methods in the class `OBSERVE`. The residual analysis may efficiently optimize this method by dropping the instrumentation from observables 2 and 3, and then adding an instrumented call to a summary method `summary_id` that performs summary transition. This summary transition will result in the monitoring tool pushing the monitor corresponding to the objects s and $iter$ that is in state 1 to state 2, saving many transitions that would have otherwise needlessly taken. The output of the analysis is shown in Figure 4.6.

The analysis results are correct in this case, provided method `process` does not throw any unchecked exception that is caught at some higher level during execution. If the method throws an exception, the control may not reach the instrumented call to method `summary_id` and the monitor will not get an opportunity to update its state to 2. If there

```
void foo(Set<Item> s) {
    int MAXELEM = 100;
    Iterator<Iter> iter = s.iterator();    // Statement 1
    OBSERVE.create(s, iter);

    while (iter.hasNext()) {
        Item item = iter.next();    // Statement 2
        OBSERVE.next(iter);
        item.process();
    }

    for(int i = 0; i < MAXELEM; i++){
        s.add(new Item());    // Statement 3
        OBSERVE.update(s);
    }

    ...

}
```

Figure 4.5: An instrumented code snippet.

are any events related to either the set object referenced by s or the iterator object referenced by $iter$, they will either not get monitored or the monitor monitoring them may not be in a consistent state. This problem happens because the methods that may throw unchecked exceptions may not be soundly approximated by the underlying static analysis, as these exceptions do not appear in the method signatures. Hence, the exceptional control flow graph may not correctly capture the control flow that is associated with these exceptions. Note that the results are of interest only if the program throwing these exceptions handles the exception and continues execution. Our observation of many open source benchmarks including DaCapo showed that unchecked exceptions are either seldom caught or if they are caught the program often terminates as soon as they are caught possibly by logging some information, as these exceptions are thrown in unexpected situations. For example, for benchmark `bloat`, out of the total 108 occurrences of catching an exception, we observed that only in 8 cases the thrown exception was unchecked and the program apparently

```
void foo(Set<Item> s) {
    int MAXELEM = 100;
    Iterator<Iter> iter = s.iterator();    // Statement 1
    OBSERVE.create(s, iter);

    while (iter.hasNext()) {
        Item item = iter.next();
        item.process();
    }

    for(int i = 0; i < MAXELEM; i++){
        s.add(new Item());
    }

    OBSERVE.summary_id(s, iter);     // Statement 2
    ...

}
```

Figure 4.6: Residual Analysis Output.

was allowed to continue its execution. Hence, program typically crashes or only does minimal logging work before it terminates instead of recovering from these exceptions and continuing its execution.

In general, a statement that lies inside an optimized safe region may impact soundness or completeness of the analysis only if (1) the statement throwing an unchecked exception is both, dominated as well as post-dominated, by observable statements within the region, and (2) the part of the region that is not executed due to the thrown exception may change the state of the monitor after the other part that has been executed, and (3) the thrown unchecked exception is caught and handled.

Note that the correctness in the presence of unchecked exceptions is a general issue that is applicable to most hybrid analysis techniques that are based on dropping instrumentation at some program points.

4.4 Residual Program Analysis

In this section, we provide a detailed account of how static typestate analysis results are processed to calculate safe regions that can be used to eliminate instrumentation. We begin by defining basic typestate concepts and terminology, then describe several forms of intermediate static typestate analysis results, and follow that by presenting the safe region identification algorithm and justifying its correctness.

4.4.1 Typestate Checking

Typestate properties, and more general safety properties, can be expressed as finite-state automata or regular expressions, e.g., [34]. Developers define properties in terms of *observations* of a program's run-time behavior. In general, an observation may be defined in terms of a change in the data state of a program, the execution of a statement or class of statements, or some combination of the two. For simplicity, in our presentation we only consider observations that correspond to the entry and exit of a designated program method; our implementations support richer sets of observations.

Static typestate checking was outlined in Section4.2. Such an analysis is typically formulated as a monotone flow-sensitive data flow analysis over a lattice of sets of FSA states, using set union to merge values at control flow join points, and using δ to define the space of transfer functions, e.g., [35]. The terms used in a monotone flow-sensitive data flow analysis have been formally defined in other literature, e.g., [67]. Much more sophisticated typestate analyses than this have been developed, such as [32, 40], but we will illustrate our approach in this simple setting.

The left-side of Figure 4.4 illustrates the data flow facts calculated for each control flow edge once the simple static typestate analysis converges to a fixpoint. These facts are sets of typestate FSA states so we refer to such an analysis as *set-based*. Analysis

```
public void doTransform(SocketChannel sc)
  throws Exception {
  ByteBuffer buf;
  while (sc.read(buf) != -1) {
    ... transform buffer data ...
    sc.write(buf);
  }
}
```

Figure 4.7: Refactored `transformData` Example

of real programs requires the ability to summarize the behavior of a method body and consume that summary at call sites of the method. A set-based analysis naturally produces a *set summary*. Consider a refactoring of the `transformData` method to isolate the while loop in the `doTransform` method shown in Figure 4.7. A set summary for this method would account for all possible behaviors of the method from all possible start states. Analyzing the structure of the automaton indicates that this method can end in only two states $\{3, err\}$ regardless of the start state. Unfortunately, such a set summary is generally too imprecise to be of value in an inter-procedural typestate analysis. For example, we know that the only typestate value that flows to the entry of the loop in `transformData` in Figure 4.4 is 3, so this value would flow to the input of `doTransform` in the refactored version. For that particular start state, the only possible end state is 3.

An alternative formulation of analysis summaries represents the behavior of a method relative to the typestate property as a function, $F \mid S \rightarrow \mathcal{P}(S)$, and we refer to these as *functional summaries*. The functional summary for `doTransform` would be: $F_{doT} = [1 \rightarrow \{err\}, 2 \rightarrow \{err\}, 3 \rightarrow \{3\}, err \rightarrow \{err\}]$. Each individual mapping in the function represents the cumulative effects of the set of possible executions of the method when starting from the given typestate. This summary allows the typestate analysis of a refactored `transformData` method to accurately calculate the typestate effects of paths through calls to `doTransform` by applying the functional summary to each typestate flowing

into the call-site and accumulating the output typestates. While more precise, calculation of functional summaries can be more expensive. In the worst-case, it requires $|S|$ flow analyses of each method, but in practice calculation of summaries is performed on demand, whenever a new start state is generated at a method call-site. This typically results in a very small number of repeated method body analyses – in our small example, the method summary need only be calculated once for typestate 3. A special case of F is the identity function which is used whenever it is determined that no observables are executable during any execution of a method which is quite common in practice. We define a *domain-restricted functional summary* for $Q \subseteq S$ as the partial function, $F\langle Q\rangle$, where $\forall_{s\in Q}\colon F\langle Q\rangle(s) = F(s)$ and $\forall_{s\in S-Q}\colon F\langle Q\rangle(s) \uparrow$. Here, $\downarrow$ indicates that a function is defined and $\uparrow$ indicates the lack of a definition.

4.4.2 Safe program regions

Central to our technique is the identification of regions of the program within which static typestate analysis identifies no erroneous PPFs and for which the cumulative effects of all paths through the region can be summarized deterministically. A *control flow graph region* is a connected single entry, single exit sub-graph of a control flow graph; in some cases, *dummy* nodes may be introduced to achieve a single entry or exit for the region – such a node would be introduced prior to the **finally** block in `transformData` of Figure 4.4 to merge the exceptional and normal flows out of the shaded region. We denote the, potentially unbounded, set of paths through a region, r, as $Paths(r)$. A path, p, gives rise to sequence $\sigma(p) \in \Sigma^*$ which encodes the program observations on the path.

Definition 4.4.1 (Safe Region) *A CFG region, r, is* safe *for an FSA, $(S, \Sigma, \delta, s_0, A)$, if*

$$\forall p, p' \in Paths(r) \forall s \in S - \{err\}\colon \quad \Delta(s, \sigma(p)) = \Delta(s, \sigma(p')) \wedge \Delta(s, \sigma(p)) \neq err$$

Intuitively, such a region will never force the typestate FSA to its error state and starting from a given typestate s, every path through the region drives the FSA to a common typestate $\Delta(s, \sigma(p))$. This is a strong condition since it demands path equivalence, relative to the FSA, from all region start states. The refactored `doTransform` method is not safe for the `SocketChannel` typestate FSA since it transitions to the err state. In practice, we relax the definition to consider safety of regions relative only to the sub-sets of the typestates that reach its entry.

Definition 4.4.2 (Reachably Safe Region) *A CFG region, r, is* reachably safe *if for the subset, $Q \subseteq S$, of typestate FSA states calculated at its entry*

$$\forall p, p' \in Paths(r) \forall s \in Q - \{err\}: \quad \Delta(s, \sigma(p)) = \Delta(s, \sigma(p')) \wedge \Delta(s, \sigma(p)) \neq err$$

Since the value 3 is the only one that reaches entry of `doTransform`, we can observe its domain-restricted summary $F_{doT}\langle\{3\}\rangle$ to see that it is a reachably safe region. In the rest of the section, we refer to both safe and reachably safe regions as *safe*; if the distinction is important we will use the more precise terminology.

Identity safe regions have no net effect on a typestate property and obey the constraint that

$$\forall p \in Paths(r) \forall s \in S: \Delta(s, \sigma(p)) = s$$

For our analysis, we distinguish three classes of regions that require support. Individual statements in the program are trivial regions, method bodies are regions, and *compound statements*, such as, conditionals, loops, or try blocks, are regions; note that a region may contain other regions. The safety of regions can be inferred by reasoning about *region functional summaries*. We overload the symbol Δ to denote summaries over the set of all program regions R as $\Delta \mid R \rightarrow S \rightarrow \mathcal{P}(S)$. For regions that are statements, t, $\forall s \in$

S: $\Delta(t)(s) = \{\delta(s, obs(t))\}$. For method, m, $\Delta(m)$ is simply the method's functional summary. For compound statements, we calculate a functional summary for the region corresponding to the statement. Conceptually, this can be thought of as refactoring the compound statement into a separate method and calculating the functional summary for that method; the refactoring of `doTransform` in the example illustrates exactly this concept.

Summarization of portions of the program in this way allows us to view a program's control flow graph as a sequence of *region summary nodes*. The successor (predecessor) relation for such summary nodes can be easily inferred from the underlying control flow graph – it amounts to the semantics realized by most debuggers when providing a *step over* functionality. We distinguish compound statements and method calls from simple statements with the predicate $isCompound$.

4.4.3 Calculating safe regions

Safe regions are calculated using a top-down traversal of the program's source text consuming the typestate sets produced by a preceding static analysis and expanding, on demand, imprecise typestate analysis results into more precise function based summaries.

The algorithm calculates information for sub-sequences of regions to assess their potential safety. Specifically it calculates an upper triangular matrix whose cells store a domain-restricted functional summary for each region sub-sequence that is specialized to the typestate values that reach the entry of the first region in the sequence. The *region sub-sequence matrix*, $D \mid \mathbb{Z}^+ \times \mathbb{Z}^+ \rightarrow (S \rightarrow \mathcal{P}(S))$, has the following structure. The diagonal elements, $D[i, i] = \Delta(r_i)\langle in(r_i)\rangle$, encode domain-restricted functional summaries based on the input values to r_i calculated by the typestate analysis. The interior elements, $D[i, j]$, encode the domain-restricted region functional summaries restricted to the composed application of preceding region summaries to the typestate value on input to r_i,

$D[i,j] = \Delta(r_j)\langle\Delta(r_{j-1})(...(\Delta(r_i)(in(r_i)))...)\rangle$. The approach potentially increases the precision of our safe region analysis since it forces the use of function summaries even if the preceding typestate analysis was set-based. We have observed that it is not uncommon for set-based analyses to produce sufficiently precise results that they can be trivially converted to functional summaries. For example, any control flow region with a singleton set flowing out of it can be converted to a functional summary with each of its input values mapping to the single output value.

Algorithm 4.8 uses D to identify an estimate of *candidate* safe regions and then performs a more focused analysis of those regions to determine the ones that are actually safe. The algorithm is called on a region, r, with a set of states, Q; initially the region is the main program and the initial set of states is the typestate FSA start state. If a region corresponding to a method has already been processed for all of the states, then it is not reprocessed (line 1). The algorithm operates in two phases: (1) lines 2-15 perform a high-level analysis of the sequence of regions that comprise r, and (2) lines 16-23 recursively process regions that are not contained within a safe region as determined by the first phase. In phase (1) the algorithm opens a window across r, bounded by the $start$ and cur regions, expanding it as long as possible (lines 6-12), then marks safe regions within the window (line 13), and finally advances the window (lines 13-15). Each iteration of loop 6-12 serves to test the region at the end of the window to determine whether it is a region boundary and, if not, it adds an additional column to D; we suppress the low-level details of memory management for D in this algorithm description.

Finding Safe Region Boundaries D is used to determine when a sequence of regions cannot be extended further. A region, r_j, is a *boundary* of a sequence $r_1 \ldots r_{j-1}$ if $\Delta(r_j)$

```
FIND_SAFE_REGIONS(r, Q)
 1  if ∀_{s∈Q} : Δ(r)⟨{s}⟩ ↓ then return
 2  start = cur = r_1
 3  cur.in = cur.prev.out = {s|s ∈ Q ∧ Δ(r)⟨{s}⟩ ↑}
 4  while start! = null do
 5     j = 1
 6     while ¬ IS_BOUNDARY(cur, D) do
 7        cur.out = ∅
 8        D[j, j] = Δ(cur)⟨cur.prev.out)⟩
 9        foreach i ∈ 1 . . . j − 1 do
10           D[i, j] = D[i, j − 1] ∘ Δ(cur)
11        cur.out = ⋃_{i∈1...j} ran(D[i, j])
12        cur = cur.next; j + +
13     MARK_SAFE_REGIONS(start, cur.prev, D)
14     start = cur = cur.next;
15     cur.in = cur.prev.out
16  foreach stmt ∈ r |!isMarked(stmt) do
17     if stmt is method call on m then
18        FIND_SAFE_REGIONS(m_body, stmt.in)
19     else if stmt is loop, l then
20        FIND_SAFE_REGIONS(loop_body, stmt.out)
21     else if stmt is conditional then
22        FIND_SAFE_REGIONS(then_body, stmt.in)
23        FIND_SAFE_REGIONS(else_body, stmt.in)
end FIND_SAFE_REGIONS()
```

Figure 4.8: Calculating Safe Regions.

can drive the analysis to an error state from a typestate reaching its input.

$$\text{IS\_BOUNDARY}(r_j, D) = err \in \bigcup_{s \in out(D[1,j])} \Delta(r_j)(s)$$

Marking Safe Regions Once a sequence of candidate safe regions is identified we must confirm that those regions satisfy Definition 4.4.2. Algorithm 4.10 does this and then marks the individual regions to indicate their safety. The algorithm attempts to find the longest region sequences that have deterministic functional summaries. It begins with $D[1, D.size]$,

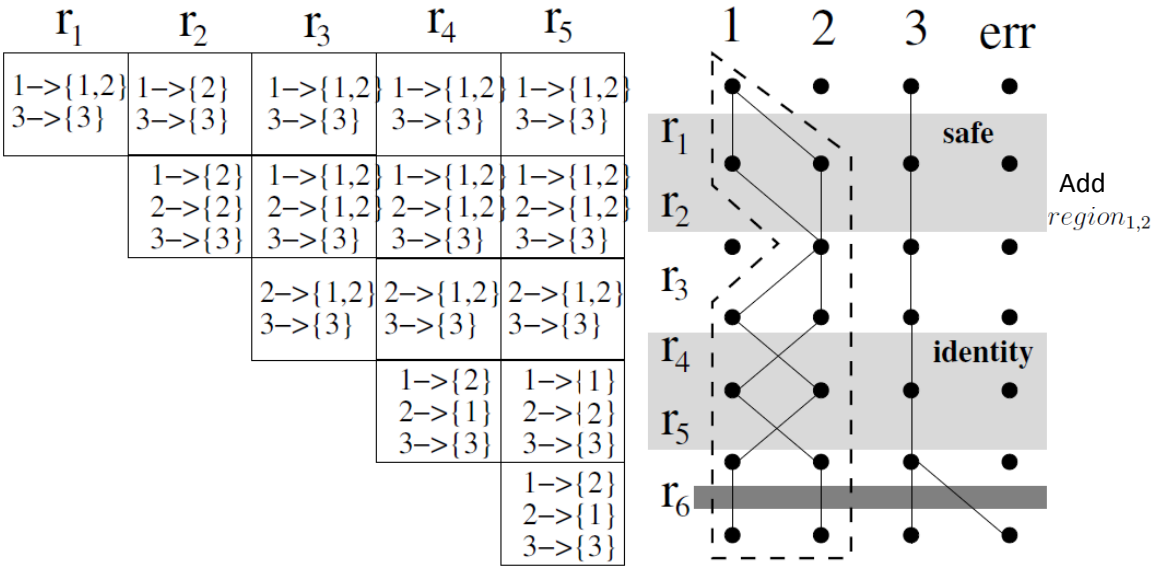


Figure 4.9: Region matrix and typestate flow example.

where $D.size$ is the number of columns (and rows) in the matrix. It tests (line 3) to see whether the image of the summary on all input values to r_1 are singletons – if so the summary is deterministic and the region is safe. If the test fails, the algorithm tests region sequence prefixes beginning with r_1 (line 2) and if that fails it then begins to test region sequences that begin at later regions (line 1).

Once a region subsequence is found to be safe, it is tested to determine whether it is an identity region (line 4). Identity regions are simply marked as safe, whereas non-identity safe regions are marked and the typestate automaton is customized. Specifically, for each safe region a new symbol is introduced into the automaton (line 8) and then the automaton transition function is defined to be the deterministic region summary $D[i, j]$ (line 9). Note

```
MARK_SAFE_REGIONS(D)
 1  foreach i = 1 ... D.size do
 2     foreach j = D.size ... i do
 3        if ∀_{s∈in(r_i)}: (|D[i,j](s)| == 1) then
 4           if ∀_{s∈in(r_i)}: D[i,j](s) == {s} then
 5              mark r_i ... r_j as safe (identity) region
 6           else
 7              mark r_i ... r_j as safe region
 8              Σ = Σ ∪ {region_{i,j}}
 9              ∀s : δ(s, region_{i,j}) = D[i,j](s)
10           i = j + 1
11           goto 1
```

Figure 4.10: Mark Safe Regions.

while $D[i, j]$ is domain-restricted it is safe to extend it to a total transition function by simply defining transitions for the undefined typestates that lead to err; those transitions are guaranteed to never be taken.

Since the algorithm works from the end of the region towards the beginning, whenever it identifies a safe region sequence, $r_i \ldots r_j$, it is known that any additional safe region sequences must begin after r_j. Consequently the algorithm skips all processing of regions that are already marked safe (line 10).

An Example Figure 4.9 illustrates the algorithm on a synthetic example. We choose a different example here because the example in Section 4.2 is relatively simple and does not illustrate the complexity of problem. The left-side shows D when assessing r_6. The image of $D[2, 2]$ on both 1 and 2 is 2. This *merging* of typestate values after r_2 is illustrated in the region typestate flow diagram on the right-side of the Figure; a column in the diagram corresponds to a typestate, each row corresponds to the in and out states for a region, and when read downward the edges encode functional summaries. We observe several interesting features of this typestate flow diagram.

(1) Region r_1 has a non-deterministic summary. This can occur when a region is a method or compound statement with internal behavior that drives the typestate automata into different states. Despite the presence of a non-deterministic summary within the region sequence r_1, r_2 the overall summary for the sequence, in $D[1, 2]$, is deterministic. This is an essential property of safe regions. This region is not an identity, consequently,
MARK_SAFE_REGIONS generates a new symbol, $region_{1,2}$, to instrument its exit and the state transition function for the typestate automaton is extended such that:

$\delta(1, region_{1,2}) = 2$, $\delta(2, region_{1,2}) = err$,

$\delta(3, region_{1,2}) = 3$, $\delta(err, region_{1,2}) = err$

(2) The region sequence r_4, r_5 is also safe, since it is clearly deterministic, $D[4, 5]$. Furthermore, despite the fact that it it contains multiple state changes along typestate flows within the region the net effect is an identity function summary, hence it is an identity region and needs no extra symbol.

(3) The typestate flow diagram shows that r_6 drives state 3 to the err state, consequently it will be regarded as a boundary. This is a function of the fact that the static typestate analysis calculated that state 3 reaches the input of region r_1. If that were not the case, then the domain-restricted summary would capture the typestate flow region in the dashed box. In that flow, there is no error transition so r_6 would be absorbed into the candidate region sequence, and ultimately into the identity region beginning at r_4. This illustrates why we use domain-restricted summaries in our analysis.

Correctness (proof sketch) Algorithm 4.8 will only mark a region as safe if it is guaranteed to satisfy Definition 4.4.2. It is clear that IS_BOUNDARY will not allow a transition to an error state to be subsumed within a safe region sequence – it will force a break in regions. It is also clear that even if IS_BOUNDARY allows for non-determinism to be included into a region, then MARK_SAFE_REGIONS will only judge a region sequence

safe if non-deterministic branching merges in the typestate flow *within* the sequence. If such branching were to persist at the end of a region sequence, then there would have to be a state that produces a summary image with more than one state. Such a summary would disqualify a region sequence from being marked safe by MARK_SAFE_REGIONS.

For the converse, we must be content with a relative notion of correctness. Our algorithm is limited in its precision, i.e., in its ability to detect maximal safe regions, by the static typestate analysis information it starts with. If there exists a (reachably) safe region that is detectable based on the static typestate analysis information, it will be marked by the algorithm.

FIND_SAFE_REGIONS expands and slides a region sequence window across the entire candidate region such that no safe region can span the window boundary. MARK_SAFE_-REGIONS then finds maximal length safe sub-sequences within window. The only possibility for missing a maximal safe region is if it overlaps two safe sub-sequences identified within MARK_SAFE_REGIONS, however, safe regions cannot overlap. Assume the existence of two overlapping safe regions $r_i \ldots r_j$ and $r_{j-c} \ldots r_k$ where $i < (j - c) < j < k$; note that region $r_i \ldots r_k$ cannot be safe, since it would have been marked by MARK_SAFE_REGIONS when $D[i, k]$ was considered. While safe regions allow branches in the typestate flow, those branches must merge prior to the end of the region. Consequently, there can be no un-merged typestate flow branches at r_j – it is the end of a safe region; the same holds for r_k. This means however that flow from r_j to r_k must be deterministic implying that $r_i \ldots r_k$ is safe which contradicts our initial assumption about the existence of overlapping safe regions.

Complexity While not optimized for complexity, this algorithm is clearly polynomial in the number of program statements. In practice, we believe that it is sub-quadratic due to

the on-demand nature of region expansion and the heuristic in IS_BOUNDARY that is used to identify boundary regions which helps to control the size of D.

4.4.4 Program analysis residue

Safe region marking can be used to reduce the cost of dynamic typestate analysis by eliminating instrumentation for all observables that are marked as safe or that lie within safe regions. No such observable can ever lead to an error and the effects of those observables in driving transitions in the typestate automaton are accounted for in MARK_SAFE_REGIONS. Specifically, if observables lie in an identity region, then even if they give rise to non-identity typestate transitions the region analysis is able to determine that there is a *compensating* transition later in the region that results in no net transition for the region. If observables lie in safe but non-identity regions, then their effects are captured by the region-specific symbols and transitions added to the typestate automaton.

The residual dynamic typestate analysis consists of instrumentation for all observables that lie outside safe regions and instrumentation at safe, non-identity region boundaries that generate the region-specific symbols. If all of the instrumentation associated with a given observable symbol is eliminated then that symbol can be eliminated from the alphabet of the typestate FSA thereby simplifying it. The reformulated typestate automaton is then used to monitor the execution of the instrumented program and reports property violations.

4.5 Evaluation

We have implemented prototype analysis components that realize the residual typestate analyses architecture in Figure 4.1 for Java programs. In the following subsections, we describe the tool support for the current implementation and its application to several Java programs that use standard Java libraries.

4.5.1 Implementation

There are three essential elements of our analysis architecture: (1) a static typestate analysis, (2) an implementation of the safe region identification algorithm, and (3) a configurable dynamic typestate analysis. We describe each of these in turn.

Developing a fully-featured, precise, scalable static typestate analysis for Java is a multi-person-year effort [40]; our analysis is much more modest. We implemented a flow-sensitive set-based typestate analysis, in the style of [35], as an inter-procedural summary-based data flow analysis in the Soot framework [72] using its type-based call graph support construction and building functional method summaries on-demand during call graph traversal.

Precise static analysis of typestate properties requires the correlation of receiver objects appearing in API calls. We implemented two techniques proposed in the literature to handle two special cases. We analyze the occurrence of `new` expressions for instances of the API under analysis to determine whether they occur in loops. (1) If not, we specialize the flow analysis facts on a per-allocation site basis [40]. (2) If so, we use a simple form of *must* value-flow analysis for reference expressions [32] to correlate receiver object expressions so that the typestate analysis can perform strong updates. We chose these approaches since they are relatively simple and inexpensive with the understanding that they are relatively imprecise. Despite the lack of precision, however, we have found that they are sufficient to reveal large safe regions in programs even when the analysis cannot conclusively prove typestate conformance.

We use Sofya and, specifically, its object-sensitive FSA (OSFSA) conformance checker application as our dynamic typestate analysis [54, 69, 36]. This implements the basic checking approach described in Section 4.4.1, i.e., δ is evaluated at run-time to maintain the current state, while automatically handling the correlation of observables to receiver

objects by capturing object ids in instrumentation. This allows the typestate analysis to precisely track sequences of API calls on the same receiver object.

Sofya is a good choice for our purposes since it allows for precise specification of the program scopes within which an observable should be instrumented. Specifically, it allows for instrumentation of observables to be enabled (disabled) within a specific static scope. For a residual analysis, where an observable o occurs at two locations in the program one of which is in a safe region and one which is not, this allows instrumentation for o to be disabled only in the safe region. In addition, the goal of our study is to understand the effectiveness of residual analysis relative to the unoptimized instrumented version of the program. We do not intend to compare the runtime data of the optimized version with any previous studies performed using any other monitoring tools. Hence, we do not expect the choice of a monitoring tool to influence our results as long as the tool provides necessary features.

Our implementation of the Algorithm in Figure 4.8 uses the data flow facts and function summaries that have been calculated by the flow-sensitive typestate analysis. This results in the reformulation of the original typestate analysis by (a) dropping instrumentation within safe regions, (b) eliminating automaton symbols for observables that are no longer instrumented, and (c) adding transitions for safe region summary symbols. In our experience, the alphabet of the residual typestate FSA often requires a smaller number of observable symbols and, more importantly, the analysis eliminates observables that can occur frequently during program execution, i.e., within loops. This residual typestate FSA is fed to Sofya's OSFSA conformance checker yielding a residual dynamic typestate monitor.

4.5.2 Artifacts

Informal typestate properties can be found in the documentation of many standard java libraries. We selected two *challenging* properties found in the header comments of the classes encoded as typestate FSA:

`java.nio.channels.SocketChannel`

henceforth named `SocketChannel`, and:

`javax.mail.Folder`

henceforth named `Folder`. Residual analysis has the potential to reduce the cost of challenging properties that may require an arbitrary number of occurrences of program observations to decide property conformance. For example, *response* [34], or *bounded liveness*, properties can be regarded as challenging and the typestate properties we selected have this structure.

For `SocketChannel` we chose the property that was presented in Figure 4.3. For `Folder` we chose the property presented in Figure 4.11. In this FSA we use wildcards "*" to denote sets of method names, e.g. `get*` denotes all methods beginning with "get". The property captures the behavior of a `READ_ONLY Folder` by only allowing `get*` and `fetch` calls when a folder instance is open. We note that a closely related property, which allows `set*` calls in state 3, encodes `READ_WRITE Folder` behavior. Due to the presence of cyclic behavior in these FSA these properties may require monitoring of arbitrarily long sequences of program observations to determine conformance.

We performed a limited search for programs that used `Folder` and `SocketChannel`. Our goal was to find programs that use the APIs in ways that are related to the two selected typestate properties. Furthermore, we wanted programs in which sequences of API calls span method boundaries, are dependent on program input values, and occur in both normal

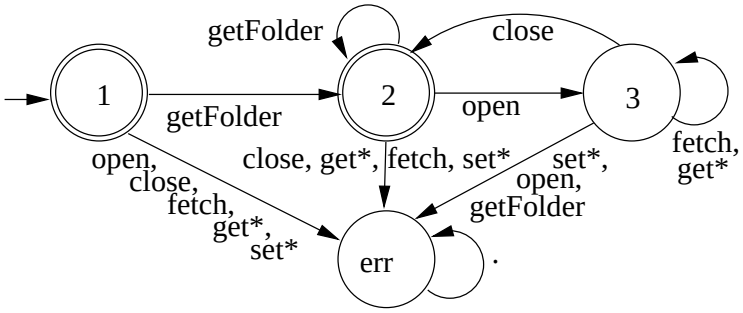


Figure 4.11: Folder API Typestate FSA.

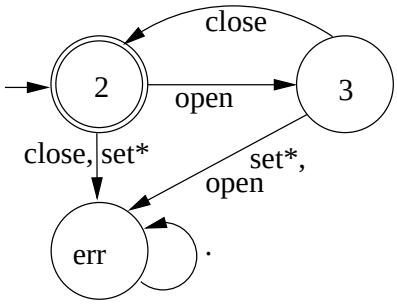


Figure 4.12: Residual Folder API Typestate FSA

and exceptional program code blocks. We selected three relatively small programs, that consisted of 2-3 classes, 6-17 methods and 177-346 lines of source code.

We studied two variations of `TimeQuery.java` a small program distributed with the programmer's guide to the Java NIO library. These programs use `SocketChannels` to connect to NTP time servers and then print out the time values. We checked the typestate property from Figure 4.3.

The first version we considered (v1) is the code from the Java NIO guide that we embellished with functionality to calculate the maximum and minimum time accessed from the set of queried time servers. This code includes the *guarding* of the `close` call in a finally block with a test for `SocketChannel` nullness as in Figure 4.3. Consequently, path-insensitive static typestate analysis cannot prove conformance. The second version

```
SocketChannel sc;
try {
  sc = open();
  sc.connect();
  ...
} catch (...) {
} finally {
  if (sc != null)
    close();
}
```

```
SocketChannel sc;
try {
  sc = open();
} catch (..) {
  return;
}
try {
  sc.connect();
  ...
} catch (...) {
} finally {
  close();
}
```

Figure 4.13: Conditional Close Refactoring.

(v2) refactors v1 to process primary NTP servers differently than secondary servers and introduces a typestate property error when contacting secondary servers. The distinction between servers is determined by comparison with a table of known primary NTP servers. Figure 4.13 illustrates a property preserving refactoring that eliminates the need for the nullness test in the `finally` block. This allows the static typestate analysis, and our safe region algorithm, to determine that the code that interacts with primary servers is an identity safe region.

We studied a program that implements a command-line interface to access Gmail via POP3 using the `GmailUtilities` library. The application code uses the `javax.mail` package indirectly through the library. Portions of this application write to `Folders` by executing calls to `set*`, thus it is impossible for even a perfect static typestate analysis to judge the entire program a safe region relative to the `READ_ONLY Folder` property. Nevertheless the typestate analysis determines that much of the program is safe with respect to the `READ_ONLY` property. Figure 4.12 shows the reformulation of the FSA in Figure 4.11 where the associated reduction in instrumentation is apparent from the reduced FSA alphabet.

Program	Input Size	Time(s) No Inst.	Result	OSFSA			Residual OSFSA		
				Time	Over.	#Obs.	Time	Over.	#Obs.
`TQ v1 (SC)`	100	2.41	pass	4.08	69	400	3.31	37	200
`TQ v2 (SC)`	100	2.37	fail	4.11	73	420	2.86	21	100
`TQ v2 (SC)`	200	4.70	pass	6.23	33	800	4.73	1	0
`POP3 (FR)`	40	2.96	pass	4.32	46	43	3.24	9	3
`POP3 (FR)`	200	3.76	pass	5.68	51	203	4.00	6	3

Table 4.1: Sample programs and analysis times. TQ = `TimeQuery`, POP3 = `Gmail POP3`. SC = `Socketchannel`, FR = `Folder`. Times in seconds and Overhead in percentage.

4.5.3 Results and Discussion

Table 4.1 presents data on several dynamic typestate analysis runs. The table reports for pairs of programs and input sizes, the time for the program to execute without any analysis instrumentation; all reported run-times are the average of three program runs of seconds of user and system time on an Athlon XP 2600+ workstation running SUSE 10.1 linux. The variation in run times was small and not significant relative to the overhead. For example, for `TimeQuery v1` it was less than 4%, where the runtime overhead for the unoptimized version was 69% and the optimized version reported an overhead of 37%.

We ran two dynamic typestate checkers: the OSFSA checker on the original typestate FSA and the OSFSA checker on the residual typestate FSA. Upon program exit the checkers indicate the result of each analysis as pass or fail. For each of these analyses, we report the run-times, the percentage overhead relative to the un-instrumented program, and the number of observations processed at runtime.

The inputs sizes were chosen to make the programs perform moderate amount of work that would be monitored. Inputs for `TimeQuery` consist of a list of NTP server URLs. We used the same server lists for uninstrumented, FSA checker, and residual versions so they would have incurred the same execution time related to network latency. The list of length 100 consists of 80 primary and 20 secondary servers; the presence of secondary servers will cause the inserted error in `TimeQuery v2` to be revealed by both dynamic analyses. The list of length 200 consists only of primary servers. Residual analysis is able to determine that the code which processes primary servers is safe and hence all observables in that code are removed. Inputs for `Gmail` consist of a list of mail commands – specifically queries to print headers and sets of specific messages. The two inputs differ only in their size; the input of size 200 is simply 5 copies of the smaller input.

Static typestate and residual analysis and code instrumentation incur a cost. The typestate analysis time and safe region calculation times ranged, on these small programs, from 4 to 6 seconds and the instrumentation time was always less than 2 seconds. Most of the analysis time was spent in call graph construction. These can be regarded as *compile time* costs. Our data show that those costs are easily compensated for by the reduction in *run time* of programs deployed with residual dynamic typestate monitors. In fact, the investment pays off very quickly – in as few as five program runs.

We report both, time and the number of observations, for our dynamic analyses. We believe that these data together would give a better idea about the benefits that may be expected by using other monitoring tools such as JavaMOP or Tracematches. We believe that the benefit of residual analysis can be seen clearly through comparison of observation counts per analysis run as well as through comparison of time per analysis run. For some properties and inputs the number of observations are halved, suggesting a halving of analysis overhead, whereas for other properties the number of observations is independent of changes in program input, suggesting that the runtime of residual dynamic typestate

analysis will grow at the same rate as that of the un-instrumented program. We can also see, in the case of `TimeQuery v2` with an input of size 200, that residual analysis can completely eliminate the need to monitor the program on certain inputs.

There is some variation in the effectiveness of residual analysis across this set of programs, properties and inputs, but residual analysis always reduces overhead significantly. In total the number of observations processed is reduced by a factor of 6 over unoptimized OSFSA and the analysis overhead is reduced on average from 54% to 14%. It is noteworthy that these reductions are achieved by applying residual analysis to the results of a relatively simple static typestate analysis. Further advances in static analysis promise to produce even larger safe regions and thereby further drive down the cost of dynamic typestate analysis.

4.6 Lessons Learned from the Evaluation

The residual analysis is an effective optimization technique as shown by our study. However, it performs a whole-program analysis which is very expensive. Moreover, the analysis requires that a region needs to have a completely deterministic summary to be identified as safe. However, it does not perform a precise analysis of path conditions and hence, in general, cannot predict infeasible program paths. Moreover, an imprecision in the analysis sometimes prevents it from accurately identifying a monitor state that would reach the entry of a region. The result is that a region may get identified as unsafe, whereas in practice, it may always behave like a safe region. One way would be to identify such regions as safe conditionally and then instrument the program with those conditions that may be evaluated dynamically. We plan to explore this idea as future work.

The cost of a program region not getting identified as safe could be high if it the region is a loop. This is because a loop may contain observables that are frequently executed generating a large number of events. A loop may not get identified as safe either because

of the imprecision in the static analysis or because the loop region indeed may have a nondeterministic summary. In either case, the analysis would have no other option but to retain the instrumentation on the observable statements. In Chapter 5, we present a technique that targets such loops for optimization.

4.7 Acknowledgements

This work has been performed in collaboration with my adviser Dr. Matthew Dwyer and a paper based on this work has appeared in the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007) [37]. Dr. Dwyer proposed the concept, and both of us developed it further. I played a major role in the design of the analysis and both of us implemented the analysis together. Dr. Dwyer played a major role in the evaluation. Both of us contributed equally in the writing of this work. Overall, both of the co-authors contributed equally in this work.

Chapter 5

Stutter-Equivalent Loop Transformation

5.1 Introduction and Motivation

In this chapter[1], we present a technique that targets program loops for monitor optimization. We leverage the observation that path properties are typically insensitive to the number of times that a loop iterates. In other words, a monitor must observe a few iterations of the loop in order to judge whether the program violates or satisfies the property, but it can effectively ignore the rest of the loop iterations.

The motivation for this work comes from the cost models as explained in Section 4.1 and also from the weaknesses of the residual analysis technique that we discussed in Section 4.6. In order to counter the weaknesses of the residual analysis, we develop a static analysis that is flow-sensitive at intra-procedural level and context-insensitive and summary-based at inter-procedural level that reduces the cost of it. Secondly (and more importantly), using program transformation techniques we reduce the cost of monitoring loops, that residual analysis might fail to optimize.

[1]This work is an extended version of our paper that appeared in the proceedings of OOPSLA, 2010 [68]. In particular, we have added two algorithms, one each for checking a loop for stuttering distance 1 and for stuttering distance 2 in Section 5.5.4.

Our major contributions for this chapter are as follows:

1) We build on the theory of stutter-invariance [55] to formalize a general framework for analyzing and transforming loops that offers the potential to significantly reduce the overhead of runtime monitoring; (2) We define an instance of that framework that realizes a common special case arising in Java programs and implement a property-driven loop optimization; and (3) We evaluate the effectiveness of that optimization on a collection of properties and programs that are drawn from previous runtime monitoring work and that exhibit high monitoring overhead. The evaluation results provide strong preliminary evidence of the potential of the technique for reducing cost of monitoring for path property conformance.

The next section provides an overview of our approach and highlights each of these contributions.

5.2 Overview

We illustrate the principles of our approach by way of of an example. Figure 5.1 shows an excerpt of a method from the *bloat* Dacapo benchmark [14]. This is one of the 424 call sites within the bloat code base that creates an `Iterator` to process the contents of a collection.

A well-known, widely studied [8, 26, 16] and previously seen in Chapter 2, `HasNext` property of the `Iterator` interface is that a call to the `hasNext()` method should precede each call to `next()`, which returns the next element. We reproduce the FSA for this property in Figure 5.2 for a quick reference, and denote the property as ϕ_{it}.

The code in Figure 5.1 creates the iterator by a call to the `iterator()` method and the while loop condition guards calls to `next()` in the loop body to ensure that they always follow `hasNext()`. This example is simple enough that by studying the code one can

```
public void visitPhiStmt(
        final PhiStmt stmt) {
  ...
  final Iterator it =
    stmt.operands().iterator();
  while (it.hasNext()) {
  //instrumentation: hasNext(it)
    final Expr op =
            (Expr) it.next();
    //instrumentation: next(it)
    if (op instanceof VarExpr)
      if (op.def() != null)
        phiRelatedUnion(
            op.def(),
                stmt.target());
  }
}
```

```
public void visitPhiStmt(
        final PhiStmt stmt) {
  ...
  final Iterator it =
        stmt.operands().iterator();
  outer : {
    inner : {
      while (it.hasNext()) {
      //instrumentation: hasNext(it)
        final Expr op =
            (Expr) it.next();
        //instrumentation: next(it)
        if (op instanceof VarExpr)
          if (op.def() != null)
            phiRelatedUnion(op.def(),
                stmt.target());
        break inner;
      }
      break outer;
    }
    // uninstrumented remainder
    // of loop
    while (it.hasNext()) {
      final Expr op =
        (Expr) it.next();
      if (op instanceof VarExpr)
        if (op.def() != null)
          phiRelatedUnion(op.def(),
                stmt.target());
    }
  }
}
```

Figure 5.1: Example from class `SSAPRE`: original (left) and transformed (right)

see that all possible sequences of calls on the iterator are consistent with the `HasNext` property. When calls occur in multiple methods or in complex control flow constructs it becomes much more difficult to determine that a program is consistent with a property.

In recent years, a number of researchers [5, 8, 26, 52] have proposed the use of runtime monitors to check program executions for conformance with such properties. A simple such analysis would create an instance of the property FSA on each call to `iterator()`, which would be defined as the *property creation event* [26], and then for each relevant call

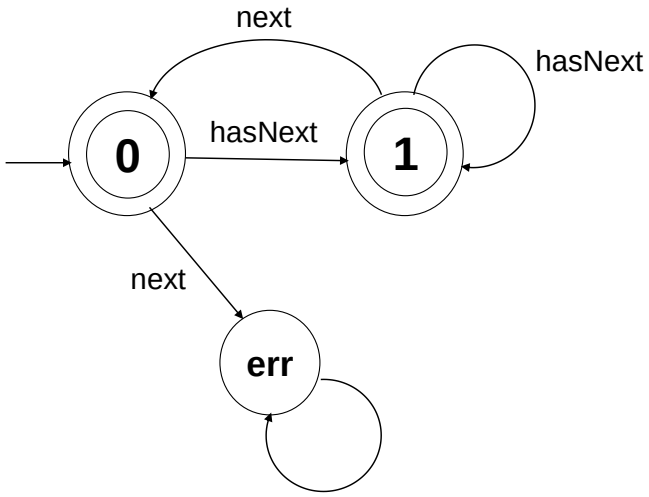


Figure 5.2: `HasNext` property, ϕ_{it}, as an FSA

on the `Iterator` instance the monitor would update the current FSA state based on the FSA's state transition function. This requires, of course, that all potentially relevant calls be instrumented to generate *symbols* in the language of the FSA.[2]

There are two dominant components of the runtime overhead introduced by monitoring: (1) the number of FSA instances needed and (2) the number of transitions simulated for each instance. Except in rare circumstances, whose details are elided in Figure 5.1, every time method `visitPhiStmt()` is called an instance of ϕ_{it} is created and if the `Collection` iterates over has k elements, then $2k$ symbols are generated. The overhead adds up quickly. When using the default Dacapo input load for *bloat*, monitoring for this property generates more than 157 million symbols generated for nearly two million FSA instances.

[2]In our presentation, we use the terms symbol when discussing automata-theoretic concepts and observable when discussing program statements that are related to a property; the term *event* is used in [26] to describe the same concept.

To reduce monitoring overhead while preserving the semantics of the original monitors, techniques such as Tracematches [8] and JavaMOP [26] heavily optimize the first overhead component, e.g., by eliminating FSA instances when they are no longer needed. The second overhead component has been addressed using static analysis to remove instrumentation of statements that do not contribute to the monitoring problem [37, 20, 17]. Unfortunately, these techniques do not help much with `bloat` because most of its monitoring overhead is incurred in loops that generate millions of events that change the FSA state, like the one in Figure 5.1, and consequently the best reported results in the literature still have unacceptable overhead – 154% [26] and 258% [16].

An alternative dynamic approach to reduce the second source of overhead consists of inserting and removing instrumentation on the fly during program execution [36, 5]. Such optimization, however, requires potentially frequent dynamic re-writing of program code to insert and remove instrumentation, and it only applies when cyclic patterns of behavior are comprised of symbols that do not cause the property state to change. Again, the code in Figure 5.1 would cause such techniques to perform very poorly, since on every iteration the loop body would be re-written twice and no occurrences of the statements whose instrumentation was removed will ever be executed.

5.3 Our Approach

Independent of whether the approach is static or dynamic, a common objective across the existing body of work on property monitoring is the determination of whether instrumentation for generating a symbol can be *completely removed* without impacting the monitoring results. We believe that aiming for complete removal has limited optimization opportunities that might be applied to monitoring instrumentation where a majority, but not all, of its executions can be removed.

Our technique leverages this realization by performing a semantics preserving program transformation to reduce the frequency of executing instrumentation in a loop. Our key insight is to calculate the loop iteration after which the behavior of any subsequent iteration is guaranteed to preserve the state of the monitored property – at that iteration the remaining loop iterations are said to `stutter` relative to the property [55].

The technique has several desirable properties: (*i*) it does not require dynamic rewriting of the program; (*ii*) it accommodates complex behaviors within loop bodies; (*iii*) it considers each loop independently and thus scales well; and (*iv*) it preserves the semantics of runtime property monitoring.

Consider the loop in Figure 5.1 and the `HasNext` property. Our analysis calculates all possible sequences of symbols in the property alphabet that can be executed by a loop iteration on the same object – in this case it is simply the sequence $hasNext; next$.

Table 5.1 illustrates the analysis of the first two loop iterations for each possible state of ϕ_{it} on entry to the loop. When entering the loop in state 1, an execution of the loop body assures that monitor transitions to state 2 (after $hasNext$) and then back to 1 (after $next$). Since subsequent loop iterations are guaranteed to preserve the state of this property, we can assert that from state 1 the loop stutters immediately on its first iteration; state identifiers written on a black background indicate the point at which stuttering begins. Whenever the monitor is in the err state the remainder of the program is trivially stuttering.

When entering the loop in state 2, the first iteration transitions the monitor to state 2 (via the self-loop on $hasNext$) and then to 1. Since state 1 was determined to immediately stutter, we can conclude that when initially entering the loop in state 2 the loop stutters on the second iteration. In fact, this is the longest iteration distance at which the loop stutters for any entry state.

For the given code and property, the analysis has determined that only the first loop iteration that generates the sequence $hasNext; next$ needs to be monitored since subsequent

Entry State	Iteration 1		Iteration 2	
1	2	1	2	1
2	2	1	2	1
err	err	err	err	err

Table 5.1: Stutter distance for ϕ_{it} on $hasNext; next$

iterations cannot reveal anything new about that property. Hence, only the first iteration of the loop needs to be instrumented and only that iteration will introduce runtime overhead. Figure 5.1 illustrates, on the right, how the loop is transformed based on the analysis results. Note that in this case it appears as if we have simply unrolled the loop one time, but in general, the transformation must account for the fact that multiple loop iterations may be performed that bypass the instrumented statements in the original loop. The correctness of our transformation requires that instrumented statements be executed a specific number of times before falling through to the uninstrumented loop.

Clearly, this technique offers significant potential for overhead reduction for runtime monitoring. The transformed loop in Figure 5.1 will always require processing of 2 symbols, rather than $2k$, regardless of the size of the collection. In the context of the bloat benchmark nearly all of the more than 900 thousand dynamic instances of `Iterators` are processed in similar loops, consequently, monitoring using our loop optimization will result in fewer than 2.5 million, rather than 211 million, symbols that need processing – a significant overhead reduction.

In the next Section we formalize the key concepts behind our transformation and establish a sufficient condition on loop transformation that if met ensures the preservation of runtime property checking. Following that we detail, in Section 5.5, the design and implementation of our stutter-equivalent loop transformation for optimizing runtime moni-

toring of Java programs. We then present, in Section 5.7, the results of an evaluation of the effectiveness of that optimization.

5.4 Terminology and Definitions

In this section, we define the requirements on program transformations that ensure that they preserve soundness and completeness of runtime monitoring of path properties.[3]

5.4.1 Runtime Path Property Monitoring

Path properties can be expressed in a variety of formalisms [34]. Regardless of the formalism, developers define properties in terms of *observations* of a program's behavior. In general, an observation is defined in terms of a program statement, such as a method call or return, coupled optionally with a predicate defined over program variables. In our presentation, we abstract away from such details by assuming the definition of a property *alphabet*, Σ, which is a set of symbols that encode observations of program behavior that are relevant to a property.

We have presented monitoring fundamentals in Section 2.3. For convenience, we reproduce the relevant part of that discussion here. For run-time monitoring, the most common form of path property specification used is a deterministic *finite state automaton* (FSA) [49]. An FSA is a tuple $\phi = (S, \Sigma, \delta, s_0, A)$ where: S is a set of states, Σ is the alphabet of symbols, $s_0 \in S$ is the initial state, $A \subseteq S$ are the accepting states and $\delta\colon S \times \Sigma \rightarrow S$ is the state transition function. We use $\Delta\colon S \times \Sigma^+ \rightarrow S$ to define the composite state transition for a sequence of symbols from Σ; we refer to such a sequence as a *trace* and denote it π. We lift the transition function from traces to sets of traces, Π, and define

[3]The transformation can be unsound and incomplete in the presence of unchecked exceptions if certain conditions described in Section 5.6 are met. Henceforth, in this section, wherever we use the terms sound and complete, we mean sound and complete in the absence of unchecked exceptions.

$\Delta(s, \Pi) = \{s' | \exists \pi \in \Pi : \Delta(s, \pi) = s'\}$[4], i.e., the set of states reached from s via any trace in Π. We define an *error* state as $\mathrm{err} \in S$ such that $\neg\exists \pi \in \Sigma^* : \Delta(\mathrm{err}, \pi) \in A$. A property defines a *language* $L(\phi) = \{\pi \mid \pi \in \Sigma^* \wedge \Delta(s_0, \pi) \in A\}$; for convenience we overload L so that $L(r)$ denotes the language defined by regular expression r.

FSA monitoring involves instrumenting a program to detect each occurrence of an observation, $a \in \Sigma$. A *simple* runtime monitor stores the current state, $s_c \in S$, which is initially s_0, and at each occurrence of an observation a, it updates the state to $s_c = \delta(s_c, a)$ to track the progress of the FSA in recognizing the trace of the program execution. We say that a program execution *violates* a property, ϕ, if the generated trace, π, ends in a non-accepting state, i.e., $\Delta(s_0, \pi) \notin A$; violations can be detected as soon as the monitor enters an error state, i.e., $s_c = err$.

Based on the Definition 2.4.1, the simple runtime monitoring approach described above is both sound and complete, provided that certain conditions in the presence of unchecked exceptions are not met. This is explained in detail in Section 5.6.

5.4.2 Stutter Equivalence

Our objective is to reduce monitoring overhead by transforming a program that would generate a trace, π, to generate a shorter trace, π', such that the two traces are equivalent relative to the property, i.e., $\pi \in L(\phi) \Leftrightarrow \pi' \in L(\phi)$. We achieve this by building on Lamport's notion of *stuttering equivalence* [55], which he introduced as a means of describing desirable limitations on the expressive power of temporal logics and which has been leveraged as a basis for partial order reductions in model checking [9].

A program execution defines a sequence of concrete program states, $\sigma = c_0 \ldots$, where c_i records the values of program variables, call stacks, execution locations, etc. Reasoning

[4] Δ inside the set comprehension corresponds to the composite state transition for a sequence of symbols from Σ.

about sequences of states at runtime is prohibitively expensive, so most existing research [52, 8, 26] instead reasons about a trace of observations, $\pi = a_0 \ldots$, where $a_i \in \Sigma$, generated by the program execution. We note that such an approach abstracts a concrete state sequence to a sequence of property states, i.e., states of ϕ, such that $\sigma_\phi = s_0, \ldots$, where $s_i \in S$ and $\delta(s_i, \pi[i]) = s_{i+1}$; we refer to the sequence of property states for a trace as $states(\pi)$.

An adjacent pair of states in $states(\pi)$, i.e., s_i, s_{i+1}, is a *stutter step* if $s_i = s_{i+1}$; one can also define stutter steps in terms of the observation in the trace that transitions from s_i, i.e., $\delta(s_i, \pi[i]) = s_{i+1} = s_i$. Two traces are *stutter equivalent* if their state sequences differ only in stutter steps. Since we are concerned with runtime monitoring, we restrict our attention to stuttering equivalence of finite traces by adapting the definition of Baier and Katoen [9, Definition 7.86].

Definition 5.4.1 (Stutter Equivalent Traces) *Traces π and π' are stutter equivalent for a given property ϕ, denoted $\pi \overset{\phi}{=} \pi'$, if there exists a finite sequence $s_0 s_1 \ldots s_n$, where $s_i \in S$, such that $states(\pi) \in L(s_0^+ s_1^+ \ldots s_n^+)$ and $states(\pi') \in L(s_0^+ s_1^+ \ldots s_n^+)$, where s_i^+ is a sequence of one or more occurrences of state s_i.*

In this definition, we judge trace equivalence by relating sequences of automaton states reached throughout the trace. We characterize such sequences as regular expressions over state values, s_i. Such an expression gives rise to a language, e.g., $L(s_0^+ s_1^+)$, defining a set of state sequences, e.g., $\{[s_0, s_1], [s_0, s_0, s_1], [s_0, s_1, s_1], \ldots\}$, where each sequence satisfies the constraints of the expression, e.g., that one or more s_0 precede one or more s_1. Judging trace equivalence based on states is powerful as it permits traces involving different symbols to be judged equivalent, e.g., if ϕ is defined such that $\delta(s_i, a) = \delta(s_i, b)$ then two equivalent traces reaching state s_i may extend the trace by a and b, respectively.

For runtime monitoring, this focus on states is appropriate since judgement about property violations is based solely on property states. We note that property states, S, play the role of atomic propositions in our adaptation of traditional stuttering frameworks, e.g., [9, Chapter 7].

5.4.3 Phrase Stuttering

A stutter step reflects the fact that an observation is irrelevant with respect to a property since it does not cause a state change. We extend this notion to *phrases*, or sequences, of symbols, i.e., elements of Σ^+, that do not cause a state change. A phrase, denoted α, is defined as a regular language over the observation alphabet.

A sequence of states $s_i, \ldots, s_j$ is a *stutter step for phrase* α if $\forall \pi \in L(\alpha) : (\Delta(s_i, \pi) = s_j) \wedge (s_i = s_j)$. Phrase stuttering trace equivalence is concerned only with the differences among non-stuttering steps in the trace.

Definition 5.4.2 (Phrase-Stutter Equivalent Traces) *Traces π and π' are stutter equivalent for phrase α and property ϕ, denoted $\pi \overset{\alpha:\phi}{=} \pi'$, if there exists a finite sequence $\sigma_0\sigma_1 \ldots \sigma_n$, where σ_i is a sequence of states in S, such that the following three conditions hold*

$$\forall \pi_\alpha \in L(\alpha) : \Delta(\sigma_i[1], \pi_\alpha) = \sigma_i[|\sigma_i|] \tag{5.1}$$

$$states(\pi) \in L(\sigma_0^+ \sigma_1^+ \ldots \sigma_n^+) \tag{5.2}$$

$$states(\pi') \in L(\sigma_0^+ \sigma_1^+ \ldots \sigma_n^+) \tag{5.3}$$

In the remainder of the section, we only consider phrase stuttering since it generalizes the simpler stuttering definitions in which the phrases are individual symbols in Σ.

5.4.4 Stutter Equivalent Loops and Programs

Since a significant source of overhead in monitoring arises due to observations processed repeatedly in loops, we start by focusing on defining loops that are guaranteed to be phrase stutter equivalent. A loop is an iterative structure in a program, i.e., every *while*, *do* and *for* statement in a Java program. A loop, l, may perform a wide a variety of different computations on any given iteration and consequently it may generate a variety of different phrases. We define the phrases that a loop body may generate as $\Pi(l)$ which can be encoded as a regular expression or an FSA. We note that in general it is possible that $\epsilon \in L(\Pi(l))$, i.e., a loop with observables may not execute them.

A loop, l, stutters for property ϕ if, regardless of the state on entry to the loop and the instance of the phrase generated by the loop iteration, the iteration can be viewed as a stutter step, i.e., $\forall s \in S : \forall \pi \in \Pi(l) : \Delta(s, \pi) = s$.

In our experience, it is uncommon for a loop to stutter immediately on the first iteration for a non-trivial property. As we have observed in *bloat*, however, it is frequently the case that after the execution of several iterations involving observations the possible states of the monitor will converge to a set of states from which the remainder of the loop iterations stutter.

Definition 5.4.3 (Loop Stutter Distance) *A loop, l, stutters at distance d for property ϕ if*

$$\forall s \in S : \forall s' \in \Delta(s, (\Pi(l) - \{\epsilon\})^d) : \Delta(s', \Pi(l)) = \{s'\}$$

Note that this definition does not "count" iterations of the loop that do not execute any observations, i.e., we remove ϵ from the set of loop phrases then take the d closure of the remaining phrases; we refer to these as *non-trivial* iterations.

We are interested in the *minimum* loop stutter distance which defines the number of non-trivial loop iterations after which the loop will stutter regardless of the property state on entry to the loop; henceforth when we refer to stutter distance we always mean the minimum such distance.

$\Pi(l)$ defines the set of traces that a loop body may execute. To reason about the equivalence of loops, we must relate possible execution paths through the loop to the elements of $\Pi(l)$ in more detail.

Definition 5.4.4 (Path Traces) *Given a program fragment f, let $paths(f)$ define the set of input constraints that force execution along each path through f. Given a constraint, $c \in paths(f)$, the execution of f on any input satisfying c generates a trace $\pi(f, c)$ for a given property ϕ.*

We first consider the equivalence of fragments comprised just of loops, then extend our notion of equivalence to programs.

Proposition 5.4.1 (Stutter Equivalent Loops) *Given a loop l that stutters at distance d for property ϕ. Loop l and a sequence of loops u are phrase stutter equivalent if they perform identical computations on the first d non-trivial iterations, and on all subsequent iterations u performs the same computation as l except that it generates no observations.*

Proof 5.4.1 *The fact that l and u perform the same computation, other than the generation of observations, means that $paths(l) = paths(u)$, yet for some constraints $c \in paths(l)$ $\pi(l, c)$ may not be identical to $\pi(u, c)$.*

By Definition 5.4.3, we can write $\pi(l, c) = \pi_d \pi_{rest}$, where $\pi_d \in L((\Pi(l) - \{\epsilon\})^d)$. In this case, $\pi(u, c) = \pi_d$. Moreover from any state, s, of ϕ on entry to the loop, $\Delta(s, \pi_d)$ is a state from which any additional instances of a phrase of l is a stutter step; note that the

phrase ϵ produces a (trivial) stutter step. Thus, $\Delta(\Delta(s, \pi_d), \pi_{rest}) = \Delta(\Delta(s, \pi_d), \epsilon)$ and we conclude that $\forall c \in paths(l) : \pi(l, c) \overset{\Pi(l):\phi}{=} \pi(u, c)$. $\square$

When a loop that iterates n times can be replaced by a stutter-equivalent sequence of loops, the cost of monitoring the loop is reduced from $O(n)$ to $O(d)$. In many cases d is very small and the cost of monitoring a loop is significantly reduced and independent of the number of iterations of the loop.

While we focus only on analyzing and transforming loops in this work, stutter equivalence naturally extends to entire programs. We define a *loop partition* of a trace π as a sequence of traces $[\pi_1, \ldots, \pi_n]$ such that $\pi_1 \ldots \pi_n = \pi$ and where each π_i consists of symbols that are either all generated within a single execution of a loop or are generated outside of any loop; note that a loop execution may be comprised of multiple iterations. Let $l(\pi)$ be a function that maps π to the loop that generated all of the symbols in π or $\perp$ if π was not generated by a loop.

Definition 5.4.5 (Stutter Equivalent Programs) *Two programs p and p' are stutter equivalent for a given property ϕ, if for every execution path, c, each program generates a trace that can be loop partitioned into*

$\sigma = [\pi(p, c)_1, \ldots, \pi(p, c)_n]$ and $\sigma' = [\pi(p', c)_1, \ldots, \pi(p', c)_n]$ such that

$\forall 1 \leq i \leq n : ((l(\sigma[i]) \neq \perp) \implies \sigma[i] \overset{\Pi(l(\sigma[i])):\phi}{=} \sigma'[i]) \quad \wedge \quad (l(\sigma[i]) = \perp) \implies \sigma'[i] = \sigma'[i])$

Intuitively, this definition requires an exact equivalence of corresponding traces that do not involve loops and phrase stuttering equivalence for corresponding traces from loops. In the next section, we present an algorithm that transforms a program to a phrase stutter equivalent by only transforming observables that lie within loops that satisfy Definition 5.4.3.

Our focus on loops reflects our insights about the current dominant source of monitoring overhead. As monitoring becomes more prevalent we expect that other program structures, e.g., recursion, may lead to significant overhead. We leave such extensions to future work.

5.5 Analysis and Algorithms

The framework presented in the previous section is quite general, but we have found that focusing on even the simplest instance of the framework, where $d = 1$, can yield significant reductions in the cost of monitoring real programs. We present an analysis to detect whether a loop enjoys this property, which we term *unit stuttering*, and prove that it calculates a sufficient condition for Definition 5.4.3. We then present a program transformation that eliminates instrumentation from all but the first iteration of such loops thereby assuring that Proposition 5.4.1 holds and that the resulting program meets Definition 5.4.5.

5.5.1 Static Analysis

We begin with a set of static analyses that calculate three properties that are necessary to ensure the soundness and completeness of the transformation process. These analyses assure that (1) all observables that can be executed within a loop have been identified, (2) those observables are related to a common set of objects, and (3) all paths involving those observables can be safely and efficiently approximated.

The first of these is assured by scanning the program for syntactic occurrences of observables. Standard techniques are used to construct a program call-graph. Methods are visited in reverse topological order for further analysis. For each method, an intra-procedural flow-insensitive analysis checks for the existence of observables inside the method; the methods that do not have any observables are not analyzed further. For methods with observables, an abstract syntax tree like representation is constructed and it is analyzed to

determine if any observables lie within loops. This analysis is an efficient flow-insensitive check within each loop body, since the analysis tool that we use directly provides an AST-like data structure. When loops with observables are found in a method, an exceptional control flow graph is built and additional analyses are run to assure the second and third properties. First, a must points-to analysis is calculated within the scope of the method to determine whether all observable method calls within a loop, including the ones that occur in the loop condition, involve the same object; for multi-object properties [66] we perform the points-to analysis for all of the object references related to the property observable. Second, a side-effects analysis is performed within loop bodies to determine that they are free of statements that reference any object involved in the property being checked.

While restrictive, these conditions help to ensure the preservation of monitor correctness in our transformation process. Several of these limitations can be relaxed, e.g., at the end of this section we describe how nested loops that contain multiple receiver objects that satisfy certain conditions can be transformed. In spite of these restrictions that help to ensure the correctness of the transformation process, the analysis can still yield property monitors that produce false positives and false negatives results for programs that use unchecked exceptions in certain ways – we explain this in detail in Section 5.6.

5.5.2 Checking for Unit Stuttering

We assume at this point that a loop l containing observables has been detected and determined to meet the conditions described above. The first step in our process is to summarize the phrases that l may generate. We do this by constructing an NFA from the control-flow graph (CFG) of l's body. CFG nodes whose statements contain an observable are mapped to an NFA transition labeled with a symbol for that observable and all other nodes generate ϵ-labelled transitions.

```
public void visitPhiStmt(
                final PhiStmt stmt) {
  ...
  final Iterator it =
          stmt.operands().iterator();
  while (ctr++ < buf.len)) {
    if(it.hasNext()){
        // instrumentation: hasNext(it):t
        final Expr op = (Expr) it.next();
        // instrumentation: next(it)
        if (op instanceof VarExpr)
        if (op.def() != null)
            phiRelatedUnion(op.def(),
                        stmt.target());
    }
  }
}
```

Figure 5.3: Example from class `SSAPRE`: Modified to show the effect of m.

In general, a loop's observables may be executed conditionally on an iteration. Definition 5.4.3 requires that we only consider iterations that execute an observable. To distinguish such iterations, we extend the NFA with an additional initial symbol, m, that *marks* the beginning of each loop iteration. The NFA is determinized and minimized to produce ϕ_l which compactly encodes the possible loop phrases. We note that since this process ignores the semantics of branch conditions within the loop body, ϕ_l overapproximates the actual phrases that can be generated by loop executions.

The top of Figure 5.4 illustrates this construction for a loop that is similar to the loop on the left of Figure 5.1, except that we have added an extra condition inside the `while` loop to show how presence of symbol m alters the structure of this automaton. Notice that the start state itself would have been an accept state in the absence of m. The changed code is shown in Figure 5.3.

The algorithm in Figure 5.5 checks whether a loop capable of generating phrases ϕ_l stutters at unit distance for a property ϕ. The algorithm operates by relating ϕ_l to a series

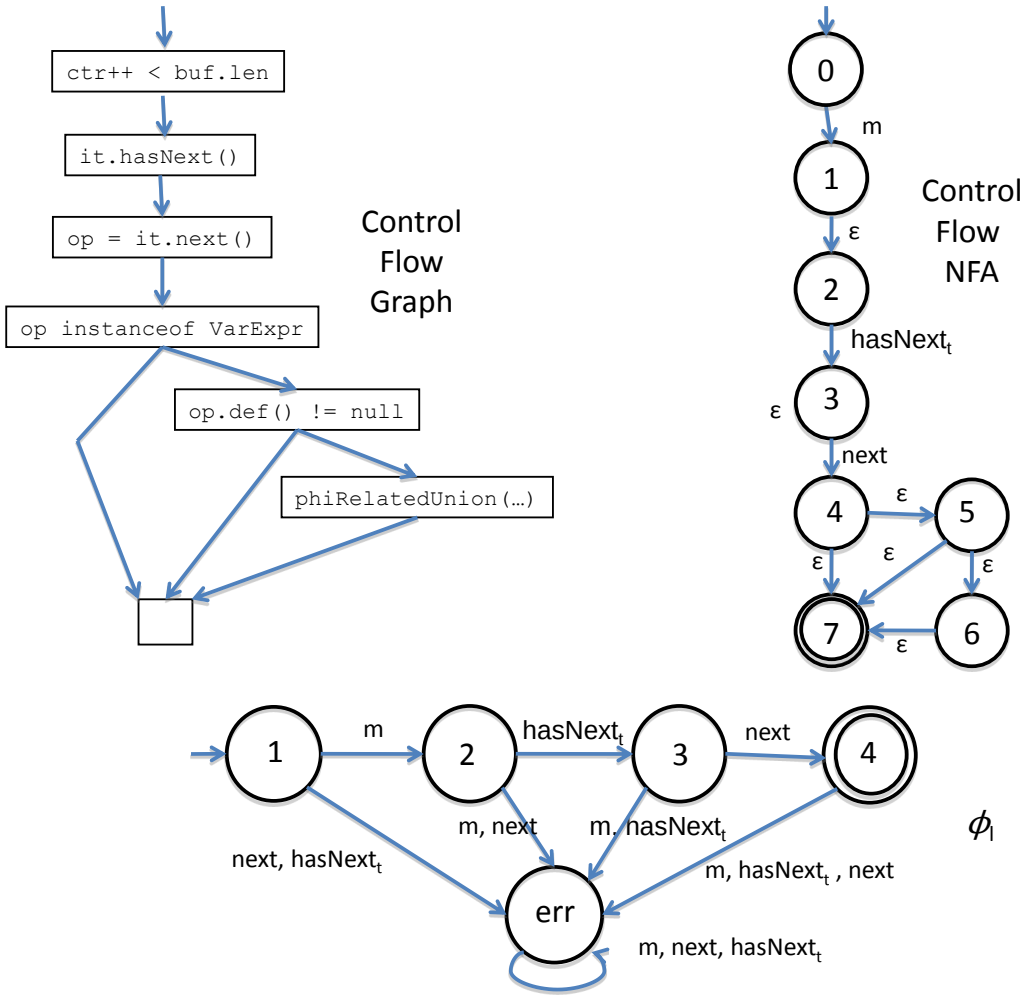


Figure 5.4: Loop phrase automaton construction

of closely related variants of ϕ. Those variants are constructed in lines 1–8, with their common elements defined in lines 1–6. Lines 1–3, define a set of shadow states for each non-error state of ϕ. Lines 4–5 extend the alphabet of ϕ with the marker symbol, m, from ϕ_l and define all states of ϕ as accept. Line 6 defines the transition function for the variants by extending ϕ's function with transitions from each state of ϕ to its shadow on m and self-loop transitions on m for all shadows. For each state in ϕ, its transitions are mirrored

$UnitStutteringCheck(\phi_l, \phi = (S, \Sigma, \delta, s_0, A))$
1 **let** S' **be a set of states, such that** $S' \cap S = \emptyset$
2 **let** $R|S - \{err\} \rightarrow S'$ **be a bijective map**
3 **let** $S_s = S \cup S'$
4 **let** $\Sigma_s = \Sigma \cup \{m\}$ **where** $m \in \Sigma_l$
5 **let** $A_s = S$
6 **let** $\delta_s = \delta \cup$
$\bigcup_{s \in S - \{err\}} \{((s, m), R(s)), ((R(s), m), R(s))\} \cup$
$\bigcup_{a \in \Sigma, s \in S - \{err\}} ((R(s), a), \delta(s, a))$
7 **for** $s \in S$ **do**
8 **let** $\phi_s = (S_s, \Sigma_s, \delta_s, s, A_s)$
9 **let** $\phi_p = \phi_l \times \phi_s$
10 **minimize**(ϕ_p)
11 **if** $|A_p| \neq 1$ **then return false**
12 **return true**

Figure 5.5: Checking a Loop for Unit Stuttering Distance.

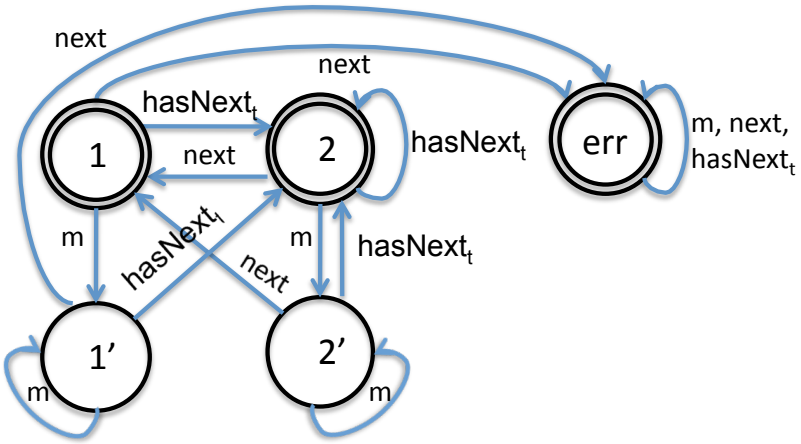


Figure 5.6: Shadow property automaton

by its shadow state. Figure 5.6 illustrates the shadow property automaton constructed for the property in Figure 5.2.

The loop on lines 7–11 considers each state of ϕ and constructs a variant of the automaton with that state as the start state. It forms the product of that automaton with ϕ_l

and minimizes the product. Intuitively, the accept states of the product define sequences of symbols that are common to both the loop and a sub-sequence of a string of the property that begins in state s. If there is exactly one such state, then all of the strings in $L(\phi_l)$ drive ϕ_s to a single accept state. If that property holds when starting in all states of ϕ, then the loop stutters after its first non-trivial iteration.

To illustrate, consider the loop and property automata in Figures 5.4 and 5.6, respectively. There are two non-error states in ϕ that may correspond to monitor states on entry to the loop; the error state is trivially stuttering. Tracing the product $\phi_p = \phi_l \times \phi_s$, starting with $s_0 = (1, 1)$ on the sequence of symbols $m, hasNext, next$ results in the sequence of states $(1, 1), (2, 1'), (3, 2), (1, 1)$ where $(1, 1) \in A_p$; all other traces fail to reach an accept state. Tracing ϕ_p, starting with $s_0 = (1, 2)$ on the sequence of symbols $m, hasNext, next$ results in the sequence of states $(1, 2), (2, 2'), (3, 2), (1, 1)$; all other traces fail to reach an accept state. Thus, the resulting products confirm that a single accept state is present in the minimized products for each start state of ϕ.

Theorem 5.5.1 (Sufficient Unit Stutter Distance Check) *If the algorithm in Figure 5.5 returns true for loop l and property ϕ, then l stutters at distance 1 for ϕ.*

Proof 5.5.1 *When $d = 1$ Definition 5.4.3 requires that $\forall s \in S : \forall s' \in \Delta(s, \Pi(l) - \{\epsilon\}) : \Delta(s', \Pi(l)) = \{s'\}$.*

Consider a single state $s \in S$. The algorithm calculates $\Delta(s, \Pi(l) - \{\epsilon\})$ by forming the product of ϕ_l, which overapproximates the phrases $m\Pi(l)$, and ϕ_s, with start state s. That product accepts every string that is accepted by both ϕ_s and ϕ_l.

Accepting strings of ϕ_l correspond to a sequence of iterations of the loop. An initial sequence of k trivial iterations generates the string m^k, which drives ϕ_s into the shadow state of s–a non-accepting state. The first non-trivial iteration produces a string $m\alpha$, where

$\alpha \neq \epsilon$, which drives ϕ_s to state $\Delta(s, m^k m\alpha)$ which by definition is an accepting state. Consequently, the product is guaranteed to have at least one accept state $(s_0, \Delta(s, m^k m\alpha))$.

A subsequent non-trivial iteration, perhaps preceded by k' trivial iterations, may produce a string $m\beta$. If

$$(s_0, \Delta(s, m^k m\alpha)) \neq (s_0, \Delta(s, m^k m\alpha m^{k'} m\beta)$$

then the product has at least two accept states. Clearly, if this is the case then the property reaches one state after the first non-trivial iteration and another state after the second, thus it does not stutter at distance one. If the two states are equal, then there is a single accept state and the loop stutters at distance one.

The product construction considers all possible loop iterations, i.e., all possible α and β. Moreover, the loop at line 7 considers each state in turn varying ϕ_s appropriately, thereby enforcing the single product accept state test for all states of ϕ. $\square$

We note the algorithm in Figure 5.5 may return false for loops that stutter at distance one. We discuss the generalization of the algorithm in Section 5.5.4.

5.5.3 Transforming Unit Stuttering Loops

A loop that satisfies the unit stuttering condition defined above may be transformed to eliminate instrumentation for any iteration after the first non-trivial iteration. To achieve this we must (1) insert a dynamic test to determine when a non-trivial iteration has been executed and (2) when that test is true we must transfer control to an uninstrumented version of the loop. These must be achieved in a manner that preserves the semantics of the loop.

It is easy to determine when a non-trivial iteration has been executed – simply detect when some observable in the loop has been executed and exit the instrumented loop after the iteration completes. However, such an approach would incur overhead even on iterations of the instrumented loop that do not execute any instrumentation. We have developed

```
Transform_Loop(l)
 1  l_inst = l.clone()
 2  block_inner = block(l_inst, break_outer)
 3  block_outer = block(block_inner, l.clone())
 4  for o ∈ obs(l_inst) do
 5    instrument(o)
 6  for o ∈ postDom ∧ o ∈ obs(l_inst) do
 7    replace o, n with o, trailer(o), n
 8    for s ∈ trailer(o) do
 9      if (s = continue) ∨ (s = goto))∧
            target(s) = entry_{l_inst} then
10        replace s with break  block_inner
11      else if s = break then
12        replace s with break  block_outer
```

Figure 5.7: Unit Stuttering Loop Transformation

a transformation that completely avoids overhead on trivial loop iterations. The key to this approach is the calculation of a loop *trailer* for an observable. The trailer for an observable statement, $trailer(o)$, is defined by the control flow subgraph rooted at the statement and ending at a loop backedge, continue, or break statement. We clone the trailer for each observable statement, re-target selected branches within the trailer, and splice the trailer into the loop immediately after the observable.

All `continue` statements that lie within a trailer are replaced with `break` statements that transfer control to the succeeding uninstrumented loop. A loop backedge, i.e., the fall through case at the end of a loop body, is also replaced with a similar `break` statement. Explicit `break` statements are retargeted to skip the succeeding uninstrumented loop.

There may be many observables in a loop, but only a subset need to have trailers created for them. Specifically, for any pair of observables o and o', if o post-dominates o' then only a trailer for o is needed. We collect all observables that are not post-dominated by another into the set $postDom$.

Figure 5.7 defines the steps in transforming a unit stuttering loop l. Lines 1-3 create and populate a pair of nested blocks. Within the inner block a clone of l, l_{inst}, is inserted followed by a `break` statement– this skips execution of the successor loop when the first l_{inst} exits. The inner block and a clone of l are inserted into the outer block.

Lines 4-5 instrument all observables within l_{inst}; no instrumentation is inserted into the other loop clone.

The loop in lines 6-10 handles each of the post-dominating observables in l_{inst}. The trailer for observable statement o is calculated and spliced into l_{inst} between o and the next statement n. The trailer is then processed to replace all `continue` statements and backedges with `break` statements targeted at the inner block, and all `break` statements are retargeted to exit the outer loop.

Figure 5.8 shows the pseudo-code for a simple loop and its transformed version. The calls to methods a and b are observables. The $*$ indicates that the call is instrumented. The trailer consisting of S_4, S_5 and the loop backedge is inserted after $o.b()$; it post-dominates $o.a()$. The backedge has been converted to a `break`. Note that this transformed loop only incurs instrumentation overhead when it executes an observable and upon completion of that iteration it transitions to the uninstrumented copy of the loop.

5.5.4 Generalizations

The approach described in the preceding section handles a limited, but valuable set of special cases. In this section, we describe several extensions that we have developed to generalize the set of loops that are amenable to stutter-equivalent optimization.

Supporting nested loops In certain cases, loop nesting can make stutter-equivalent transformation very complex. We identify two commonly occurring cases that can be supported safely; in all other cases we do not transform the loop nest.

```
1  while Cond_1 do
2    S_1
3    if Cond_2 then
4      S_2
5      o.a()*
6      S_3
7      o.b()*
8      S_4
9    S_5
```

```
1  outer : {
2    inner : {
3      while Cond_1 do
4        S_1
5        if Cond_2 then
6          S_2
7          o.a()*
8          S_3
9          o.b()*
10         S_4
11         S_5
12         break inner // if ends
13       S_5 // while ends
14     break outer } // inner ends
15   while Cond_1 do
16     S_1
17     if Cond_2 then
18       S_2
19       o.a()
20       S_3
21       o.b()
22       S_4
23     S_5 } // while and outer end
```

Figure 5.8: Original loop (left) and Transformed loop (right).

We can distinguish each loop in the nest by its depth. When observable statements only occur at one depth in the loop nest we can transform that loop to a stutter-equivalent sequence of loops. Any loops that are deeper in the nest are treated as any other code in our transformation, i.e., trailers are computed branches are retargeted. Any loops that are shallower remain unchanged and simply enclose the now transformed loop. When observables can occur at multiple depths, it must be the case that the objects involved at each depth are provably distinct. If this condition is met, transformation can proceed from the inner-most loop with observables outwards.

Stuttering at $d > 1$ The algorithm in Figure 5.5 only treats a stutter distance of one. The condition on line 11 is stronger than required. It can be relaxed and the algorithm can be made more precise as shown in Figure5.9.

$UnitStutteringCheck(\phi_l, \phi = (S, \Sigma, \delta, s_0, A))$
1 **let** S' **be a set of states, such that** $S' \cap S = \emptyset$
2 **let** $R|S - \{err\} \rightarrow S'$ **be a bijective map**
3 **let** $S_s = S \cup S'$
4 **let** $\Sigma_s = \Sigma \cup \{m\}$ **where** $m \in \Sigma_l$
5 **let** $A_s = S$
6 **let** $\delta_s = \delta \cup$
$\bigcup_{s \in S - \{err\}} \{((s, m), R(s)), ((R(s), m), R(s))\} \cup$
$\bigcup_{a \in \Sigma, s \in S - \{err\}} ((R(s), a), \delta(s, a))$
7 **for** $s \in S$ **do**
8 **let** $\phi_s = (S_s, \Sigma_s, \delta_s, s, A_s)$
9 **let** $\phi_p = \phi_l \times \phi_s$
10 **minimize**(ϕ_p)
11 **for** $s' \in A_p$ **do**
12 **let** $\phi_{s'} = (S_{s'}, \Sigma_{s'}, \delta_{s'}, s', A_{s'})$
13 **let** $\phi_{p'} = \phi_l \times \phi_{s'}$
14 **minimize**$(\phi_{p'})$
15 **if** $\{s'\} \neq A_{p'}$ **then return false**
16 **return true**

Figure 5.9: More Precise Algorithm for Checking a Loop for Unit Stuttering Distance.

The difference between this algorithm and the one in Figure 5.5 is the nested loop that allows more than one end state for a start state after the first nontrivial iteration of the loop. The nested loop in lines 11–15 checks that after the first nontrivial iteration the end state does not change. In general, the loops starting with the outermost loop find the end states reachable from every start state for the iteration that corresponds to the loop's level of nesting.

Generalizing the loop transformation process to distance d requires the introduction of d copies of the loop where those loops are nested within blocks in a cascading fashion. For example, for loop l with $d = 2$ one would produce:

```
d0 : { d1 : { d2 : { l2 } l1 } l }
```

where the trailers in the $l2$ copy of l break using label $d2$ and transition to $l1$. The staging of loop copies serves to perform a "count down" of the non-trivial loop iterations, until the original uninstrumented loop l is reached.

This shows that the transformed code grows linearly with the stuttering distance. However, for the algorithm in Figure 5.9 the time grows exponentially in terms of the number of states with respect to the stuttering distance. This is because increasing the stuttering distance by a unit length would result in adding another level of nesting to the nested loops in lines 7–15, and every loop would iterate as many times as $|A_p|$ which in the worst case would be equal to $|A|$.

This algorithm can be made more efficient if we keep a set of stuttering states at every level, so that the state which has already been found to be stuttering at that distance need not be recomputed. This means that the total loop iterations at every level would be bound by $|S|$. In other words, for this algorithm, the time grows linearly in terms of the number of states with respect to the stuttering distance. This algorithm is presented in Figure 5.10 for stuttering distance 2.

The algorithm has a newly added lines 12 and an extra condition in line 11 in addition to the original corresponding to set $stutter_1$. This set ensures that the nested loops are executed only if the new start state at that level has not seen by it previously. We have not shown a similar set for the outermost loop at line 7. Since the outermost loop iterates for all states, keeping such a set would be needless. We plan to implement this efficient algorithm in the future. Note that this algorithm is also as precise as the one presented in Figure 5.9.

Supporting distinct loop traces The algorithm in Figure 5.5 is too restrictive in that it requires all loop phrases, $\Pi(l)$, to lead to the same state, i.e., the second component of the

$UnitStutteringCheck(\phi_l, \phi = (S, \Sigma, \delta, s_0, A))$
1 **let** S' **be a set of states, such that** $S' \cap S = \emptyset$
2 **let** $R|S - \{err\} \rightarrow S'$ **be a bijective map**
3 **let** $S_s = S \cup S'$
4 **let** $\Sigma_s = \Sigma \cup \{m\}$ **where** $m \in \Sigma_l$
5 **let** $A_s = S$
6 **let** $\delta_s = \delta \cup$
$\bigcup_{s \in S - \{err\}} \{((s, m), R(s)), ((R(s), m), R(s))\} \cup$
$\bigcup_{a \in \Sigma, s \in S - \{err\}} ((R(s), a), \delta(s, a))$
7 **for** $s \in S$ **do**
8 **let** $\phi_s = (S_s, \Sigma_s, \delta_s, s, A_s)$
9 **let** $\phi_p = \phi_l \times \phi_s$
10 **minimize**(ϕ_p)
11 **for** $s' \in A_p \wedge s'\neg \in stutter_1$ **do**
12 **let** $stutter_1 = stutter_1 \cup \{s'\}$
13 **let** $\phi_{s'} = (S_{s'}, \Sigma_{s'}, \delta_{s'}, s', A_{s'})$
14 **let** $\phi_{p'} = \phi_l \times \phi_{s'}$
15 **minimize**$(\phi_{p'})$
16 **if** $\{s'\} \neq A_{p'}$ **then return false**
17 **return true**

Figure 5.10: Efficient and Precise Algorithm for Checking a Loop for Stuttering Distance 2.

single product accept state. This can be relaxed by using the iterated product construction approach described above and, particularly, the A_p fixed point test.

This approach for unit stutter distance checking requires two products to be constructed, the first to generate an A_p and the second to confirm that it is a fixed point. In addition, we observed a high frequency of cases with unit distance. Hence, we favored the algorithm in Figure 5.5 which is more efficient for the unit stuttering case.

5.6 Soundness and Completeness

Java provides checked as well as unchecked exceptions. Although the problems associated with both types of exceptions are similar, the analysis handles them differently because

methods are not required to declare unchecked exceptions, and such exceptions may be thrown by many different statements within a program.

For the purpose of the analysis, it only matters if a statement that may throw an exception lies inside a loop that contains at least one observable. Any other statement will not be considered in the loop analysis and transformation process, and hence cannot affect the soundness or completeness of the resulting property monitor. The occurrence of such a statement inside a loop may introduce a new loop phrase, but that new phrase impacts the loop transformation process only if it causes the failure of the unit stuttering distance check. This depends on both the loop phrase and the property being monitored.

Formally speaking, a statement that lies inside a loop that is processed may impact soundness or completeness only if (1) the statement throwing an unchecked exception is both, dominated as well as post-dominated, by observable statements within the loop, and (2) the suffix of the loop phrase that is not executed due to the thrown exception may change the state of the monitor after the prefix has been executed, and (3) the thrown unchecked exception is caught and handled.

This problem with the handling unchecked exceptions also impacts the residual analysis. We have discussed it in detail in Section4.3.

We illustrate how exceptions can impact the operation of our monitor optimization approach through examples in the remainder of this subsection, and conclude with a discussion of the limitations of our current analysis implementation.

5.6.1 Checked exceptions

Our analyses operate on exceptional CFGs for each method in the program that includes a loop containing property observables. These CFGs, in addition to edges that represent normal control flow, include an edge to represent the exceptional control flow introduced

```
try {
    while (it.hasNext()){
        x.throwsException();
        it.next().doSomething();
        ...
    }
} catch (CheckedException e) {
}
```

Figure 5.11: Stutter distance checking in the presence of exceptions.

by statements that are declared to throw an exception.[5] A throw of a checked exception must be explicitly declared in Java.

The loop phrase automaton construction process overapproximates the execution paths based on this CFG which ensures that the loop phrases corresponding to the loop executions that throw exceptions are taken into account. Consequently, the algorithm in Figure 5.5 that checks for unit stuttering distance handles checked exceptions directly.

Figure 5.11 shows a simple loop inside a `try` block that has a statement which invokes method `throwsException()` and catches `CheckedException`, which is a subtype of `Exception` but not of `RuntimeException` – in other words it is a checked exception.

Here we consider the case where the method may throw a checked exception. Suppose the property being monitored is the `HasNext` property in Figure 5.2 for which both $hasNext$ and $next$ are observables. The invocation of `throwsException()` lies on the path that connects the two observables and the exceptional control flow makes the loop phrase $hasNext$ possible; since the call to `next()` may be bypassed.

The loop phrase automaton construction process ensures that all possible loop phrases, i.e., the exception related phrase $hasNext$ and the non-exceptional phrase $hasNext; next$, are taken into account. The unit stuttering check calculates that the loop does not have a unit

[5]Our discussion in this section is based on the exceptional CFG provided by the Soot Java analysis framework [70] which we use in our implementation as discussed in Section 5.7

stuttering distance, due to the fact that the two phrases drive the the property automaton into distinct states, and hence this loop would not be transformed. Note that whether method `doSomething()` might throw an exception has no impact on the loop transformation process since it does not lie on a back-edge free path connecting two observables.

5.6.2 Unchecked exceptions

In Java, methods are not required to declare that they throw unchecked exceptions. This makes construction of an exceptional CFG a challenge. Many different Java statements, in addition to method invocation, may potentially throw unchecked exceptions. An extremely conservative analysis may overapproximate exceptional control flow by including an exceptional edge at every such statement. This approach, however, would make for a very imprecise CFG and significantly degrade the precision of any analysis performed using it.

We adopt the approach used in recent work [20, 17] which handles many common uses of unchecked exceptions. The exceptional CFG we use includes an exceptional edge whenever the program includes a `throw` statement regardless of whether the exception thrown is checked or unchecked. In addition, it conservatively adds an exceptional flow edge for every statement that may throw an unchecked exception, if an appropriate `catch` target is specified in the same method body. If neither of these cases hold, then our exceptional CFG may not model all possible exceptional transfers of control, since it does not determine whether method invocations may throw an unchecked exception.

In general, if a transformed loop contains more than one observable and a statement that lies on a path between any two observables can throw an unchecked exception that is not declared and is not caught in containing method, then the property monitor may be in an incorrect state as a result of our transformation. In this situation, our analysis is not guaranteed to produce property monitors that are sound and complete.

We illustrate using the `HasNext` property and the example in Figure 5.11, but this time we consider the case where `throwsException()` may throw an unchecked exception. By not modeling this unchecked exception, our approach would transform the loop. During any iteration after the first one, if `throwsException()` actually throws an unchecked exception, then that exception would not be caught by the `catch` of `CheckedException`. In this case, the loop exits and the program continues execution with the property monitor in state 2, since the $hasNext$ observable has been processed. This is potentially problematic because our unit stutter distance check, which was misinformed about exceptional control flow, calculated that the loop is always exited in state 1.

An incorrect property monitor state can give rise to either false or missing reports of property violations. Thus the treatment of undeclared unchecked exceptions means that our analysis and transformation approach may produce monitors that are unsound and incomplete.

In future work, we plan to investigate the use of additional analyses that focus on the set of unchecked exceptions that are explicitly caught in the program, but that are not declared as thrown by methods in the program. Unchecked exceptions that are not caught will abnormally terminate program execution thereby indicating a failure that the developer should investigate. Unchecked exceptions that are caught may be problematic for our analysis, but only if methods that throw them are called within loops that our analysis targets for transformation. Information about the absence of such methods within loops can be used to confirm the soundness and completeness of our monitor optimization.

5.6.3 Try-catch blocks inside loops

The presence of exception `throw` statements and `try-catch` blocks containing observables inside of loops complicates the detection of stuttering loops and their transformation.

Moreover, the computation of trailers for observable statements within `try-catch` blocks inside of loops is also complicated and we leave its general treatment for future work. At present, even if a loop is determined to unit stutter, we do not transform it if it contains a `try-catch` block containing an observable statement.

5.7 Evaluation

In this section we investigate the potential of the proposed technique and assess its performance against existing monitoring approaches. More specifically, we wish to explore two questions: (RQ1) How effective is the optimization in reducing the number of events processed?, and (RQ2) How well does the optimization perform when the monitoring problem is scaled?

5.7.1 Artifacts

We did not reuse the artifacts that we used in Section 4.5.2 as we have a more complete implementation of our analysis that can be evaluated on larger benchmarks and we wanted to evaluate the effectiveness of our analysis by comparing its results with those presented by other researchers [20, 16, 26]. Hence, the natural choice was to use the same artifacts that were used by other researchers. To assess RQ1 we consulted [26] which reports data on the number of events generated and monitoring overhead incurred when checking 6 finite-state properties against 11 Dacapo [14] benchmarks. Of the 66 program-property pairs reported in Table 3 of [26], we systematically selected the ones that incurred overhead of greater than 25% when using JavaMOP (the ones where further optimization seemed valuable). This selection resulted in 4 pairs comprised of the combination of two programs, bloat and pmd, and two properties, HasNext (shown in Figure 5.2) and FailSafeIter which checks that no call to `update` a collection is made between any pair of calls to `next` on an iterator

for that collection. The FailSafeIter property is a multi-object property and the analysis must keep track of events on both an iterator object and the collection being iterated over. The latter are generally spread across multiple methods making them a challenge for the analysis.

To expand the scope of our study, we selected an an additional open source artifact for which monitoring overhead promised to be high. JGraphT version 0.8.1 is a substantial library, consisting of 172 classes, that is designed for manipulating large graphs. It exhibits a number of design features that promote reuse, configurability, and high-performance. JGraphT includes a load test which we analyzed against both properties selected above.

5.7.2 Design

The overhead of runtime monitoring, and the effectiveness of our optimization depend on the program, the property, and the input to the monitored program execution. There are some programs that never instantiate the type(s) that are related to the property – overhead in this case is trivial. In other situations, only specific program inputs will give rise to significant monitoring overhead. Finally, the details of the property encoding can give rise to variation in overhead.

For RQ1 we consider three monitoring treatments. The first treatment, *original*, consists of no monitoring at all, serving as a baseline to assess the other approaches. The second treatment, *control*, consists of inserting the property monitor into the application, representing the state of the practice and an upper bound on performance overhead. The third treatment, *optimized*, corresponds to applying our optimization technique, then inserting the property monitor after the program transformation.

For each combination of program and property, we apply each of the three treatments to integrate the property monitor into the program, and execute the program with its default

input. This design results in the collection of 6 observations (3 programs and 2 properties) per treatment, 18 for the whole study.

For RQ2 we consider two dimensions of scalability: the program input size and the per-event monitor processing.

For input size, we investigated the use of large Dacapo inputs, but found that those inputs are not simply larger versions of the default input. They are structurally quite different and thus one cannot compare performance on default and large inputs directly to infer a technique's scalability. For jgrapht, we are able to manipulate the size of the input while maintaining its structure. We use three input size treatments: 0.25 times the default size, the default size, and 4 times the default size, where size is defined by the number of nodes and edges of the graph that serves as input to jgrapht.

For monitor processing, we used three treatments to vary the per-event processing in monitoring. The *low* cost monitor does the absolute minimum processing and corresponds to the properties used in [26]. In reality, it is likely that some information will be recorded during monitoring to aid in fault diagnosis and debugging. We use our instrumented aspect as a representative of a *high* cost monitor. Finally, we consider a monitor that records significantly less information, but still permits the developer to identify the source locations that incur high overload. We refer to this as a *medium* cost monitor.

5.7.3 Measures

Our primary measure is the number of *events* generated during the execution of a benchmark. This measure is valuable since it represents a platform independent measure of the work performed during monitoring. Reducing the number of events generated is guaranteed to reduce monitoring overhead.

We also report the runtime *overhead* incurred due to monitoring. We consider this a secondary measure, since variations in the execution platform can give rise to variation in measured overhead that is not due to our optimization. For example, when two measures of monitoring overhead are within 3% of one another the benefit of the optimization is obscured; comparing events permits that benefit to be observed.

So as to not conflate these measures, we use two variants of each property to collect data. To record overhead we use the exact same property encodings as were used in [26]. To record events, we customize the property to record the number of events during a run and the source line of the program at which each event is generated.

For a given program and property, a subset of the program loops will involve events appearing in the property – these are candidates for our optimization. We record the number of candidate *loops* that are transformed. This provides an indirect indication of the potential for our optimization to reduce monitoring costs across a range of input values; the higher the percentage of transformed loops the larger the space of inputs our technique is effective on.

5.7.4 Infrastructure

The technique presented in Section 5.5 could be implemented in any number of compiler infrastructures. It requires access to a high-level representation of the program, to ease the detection and analysis of loops, and a low-level representation, to enable analyses of the semantics of instructions within the loops. We chose to prototype the analysis in Soot [70], since we were familiar with it and it is a rich analysis infrastructure, and used its Dava [64, 65] decompilation framework to provide the high-level information we required.

Figure 5.12 shows an architecture of the stutter-equivalent loop transformation analysis. The Dava toolchain works in multiple phases. First, Soot processes a `.class` file

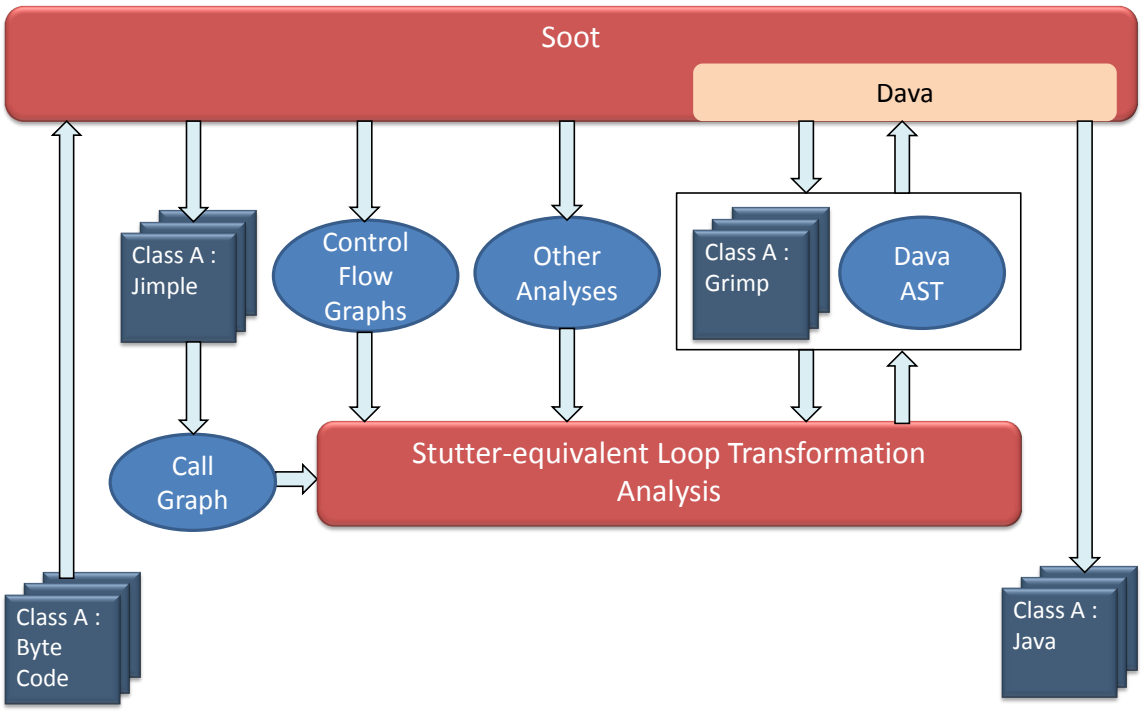


Figure 5.12: Architecture for the stutter-equivalent loop transformation analysis.

to produce Jimple, a 3-address representation of bytecodes, and call graph construction is performed, using either class hierarchy analysis (CHA) or the more precise Spark [57] points-to analysis framework. We used Spark for bloat, but were forced to revert to CHA for pmd and jgrapht to control analysis time. Second, Dava constructs an AST-like representation of the program based on Grimp, another higher level representation of code. It is after this second step where our analyses is inserted. The transformed loops are encoded in Dava's AST representation, with appropriately cloned versions of the underlying Grimp representation. The remainder of Dava involves analysis of the AST, optimization of its structure for pretty printing, and generation of Java source.

Dava was not designed to be used as a high-level program representation and, unlike the rest of Soot, it has not been actively maintained since 2006. This presented two significant problems: (1) Dava includes significant functionality for creating *programmer friendly* decompiled output and this leads to poor runtime performance, and (2) Dava's evaluation did not include the Dacapo benchmarks and those programs exposed numerous latent bugs in its implementation. These problems impacted our study in two ways.

First, bugs in Dava caused our transformation to fail, through no fault of our technique, and we had to detect and workaround those problems. We did this in two ways: (a) We instrumented our analysis and transformation implementations to log information about the methods they processed so that we could detect when a Dava bug aborted that process. In such cases, we simply output the original method leaving it untransformed. This means that limitations in Dava force us to underestimate the benefit of our optimization. (b) For each program-property pair we manually inspected the transformed program comparing it to the original program to ensure that the program semantics were preserved. Note that the transformation log allowed us to focus on just the methods that were transformed, but in some of the benchmarks there were hundreds of such methods. We also executed each transformed program and compared its output to the original program to confirm that it computed the same result.

Second, the cost of running Dava and, especially, the efforts we went to confirm the correct operation of Dava and of the transformed programs made performing our experiments quite expensive. The combination of bloat and hasNext is representative of the cost of analyzing all of the program property pairs. Running Soot's points-to and call-graph construction takes 150 seconds for this program, a necessary precursor of our analysis which takes 45 seconds, including the processing related to the logs described above. The rest of Dava's execution involves constructing and optimizing ASTs and generating Java code

Program Property	Control events	Optimized events	ratio
bloat			
HasNext	157126421	2272471	0.01
FailSafeIter	80970380	2733640	0.03
pmd			
HasNext	4457941	120399	0.03
FailSafeIter	2261768	72576	0.03
jgrapht			
HasNext	16120016	80024	0.005
FailSafeIter	8040006	8040006	1.0

Table 5.2: Events processed during monitoring

which takes about 20 to 25 minutes. Auditing the log files and working around Dava bugs takes approximately 2 hours per program-property pair.

In the future work, we plan to shift our analysis to a framework that eliminates the limitations we encountered with Dava which will allow us to significantly scale our evaluation.

We use JavaMOP 2.0 as the mechanism for encoding properties and generating them in aspectJ. Those aspects are then woven, using ajc 1.6.8, and the resulting Java program is executed on an Opteron 250 running CentOS 5.2 and JVM 1.6.0 with 16 Gigabytes of memory.

For the Dacapo benchmarks, we run them using the `-converge` option until the variation in execution time is less than 3%. We report the average execution time and use it to compute runtime overhead. For jgrapht, we take a less rigorous approach. We run the program five times and then use the average execution time.

5.7.5 Results and Discussion

Table 5.2 reports, for each program-property pair, the number of events generated by JavaMOP on the original program and on the optimized program. It also reports the ratio of

Program Property	Original time	Control time	Optimized time	Optimized loops
bloat	5.2			
HasNext		173%	73%	232/293
FailSafeIter		702%	298%	192/349
pmd	5.7			
HasNext		8.8%	3.5%	108/131
FailSafeIter		3.5%	3.5%	122/164
jgrapht	4.8			
HasNext		808%	69%	29/31
FailSafeIter		594%	594%	5/38

Table 5.3: Monitoring time and loops optimized

Scaling factor	events		orig.	per-event monitor cost					
	ctl.	opt.		low		medium		high	
				ctl.	opt.	ctl.	opt.	ctl.	opt.
0.25	4060016	40024	1.0	34.8	18.9	84.4	44.6	91.5	41.1
1.0	16120016	80024	4.6	15.5	3.5	80.2	21.4	88.9	22.7
4.0	64240016	160024	29.2	7.7	6.3	70.6	4.1	83.7	6.7

Table 5.4: Effects of scaling on jgrapht-HasNext. Overhead in percentage.

events reported under optimization to those reported without optimization. In 5 of the 6 pairs, we observe reductions in generated events ranging from between 1 and 3 orders of magnitude. The data for jgrapht-FailSafeIter indicates that the optimization was completely ineffective in this case; we discuss this in detail below.

Table 5.3 reports the execution time of the original program execution in seconds, with no monitoring, and the overhead of the control and optimized treatments. For the optimized treatment, we also report the fraction of candidate loops that could be optimized. We observe that in 4 out of 6 cases the use of optimization significantly reduces overhead.

It is interesting to note that for pmd-FailSafeIter the overhead of optimized and control treatments are the same, despite a significant reduction in events processed. We discuss this issue in detail below.

As a final observation, we see in Table 3 that the performance of jgrapht-FailSafeIter was due to the fact that its execution did not transit any of the 5 loops in the program that were transformed. We present a more detailed analysis of this case below.

Table 5.4 reports on our limited study of scalability. The second and third columns present data on how the number of events generated varies with input size. The event measure is independent of monitor processing, but overhead is not. The fourth column shows the execution time of the unmonitored program as input scales. The remaining columns report, for each monitoring cost treatment, how overhead varies with input size.

The event data indicates that the number of events scales linearly with input size for the optimized program, whereas in the control treatment the number of events appears to grow exponentially with input size. For low cost monitors, the trend of reduced overhead appears to be somewhat obscured by measurement noise, but for higher cost monitors it is quite clear. The significant reduction in execution time when using monitoring gives rise to a strong downward trend in overhead as input size scales; this is due in no small part to the fact that program execution time appears to be growing super-linearly.

5.8 Observations and Lessons Learned for the Evaluation

The data suggest that stutter-equivalent loop transformation can significantly reduce the number of events generated during program monitoring. That reduction can yield significant reductions in runtime overhead and, based on limited data, those reductions appear to increase as programs run longer and when more *realistic* monitors that record data, e.g., for fault localization, are used.

While generally positive, these results point to the need for additional optimization of program monitoring if it is to become a technology that is widely deployed. In addition to the number of events generated, the number of objects that are monitored is a significant source of overhead – and one that our optimization does not directly target. Our recording aspects provided data on the number of monitors created during monitoring. We found that bloat had upwards of 1 million, pmd had approximately 58 thousand, and jgrapht approximately 25 thousand monitors generated for each program run. Unlike processing an event, the time required to create a monitor is non-trivial and we conjecture that for pmd-FailSafeIter the bulk of the overhead is comprised of monitor creation, thus obscuring differences related to reductions of generated events.

Research targeting at reducing the number of monitors is sorely needed. Several researchers have investigated object sampling techniques [18, 5] that make a randomized choice when object creation events occur and either create a monitor or not. These can dramatically reduce monitoring cost, but they sacrifice fault detection – our event reduction optimization does not.

There is an existing line of research that has explored the use of static analysis to reduce the cost of property monitoring. Eric Bodden's doctoral thesis [16, 17], reports the results of a sequence of different static analysis culminating in a Nop-shadows analysis. We cannot directly compare to that analysis since it is designed specifically for reasoning about properties expressed as Tracematches [8]. We note, however, that the application of Nop-shadows to the four Dacapo program-property pairs that we studied led to modest reductions (see Table 5.6 in [16]) whereas we observed greater than 2 times overhead reduction on three of those four programs.

We observed that in the data we report the overhead of monitoring program-property pairs with JavaMOP differs, in some cases significantly, with the data reported in [26]. Our platform differs from the one they used in two significant ways: we use linux and they used

Windows and we use JavaMOP 2.0 – their latest release. We have shared our experimental results with the developers of JavaMOP to determine whether the performance differences are related to optimizations in their latest release or whether some other factor is the cause. We will investigate this in the future.

Of the 38 loops in jgrapht that involve events in the FailSafeIter property our technique transformed 5. Unfortunately the application we studied only executed 4 of the 38 loops and none were one's that were transformed. Moreover, a single loop was responsible for generating over 8 million events – 99.5% of the events generated during monitoring. That loop is part of a custom iteration framework that supports the efficient iteration over graphs using different strategies, e.g., breadth-first or depth-first. That framework implements a `next` method that tests and updates internal collections to ensure that a vertex, or edge, is only visited once during an iterator traversal. The points-to analyses that support our optimization are unable to determine that the updates to the internal collections are not updates to the collection being iterated over and thus, the loop is not transformed. A modest extension of the points-to analysis would allow this loop to be transformed, but we refrained from implementing problem-driven extensions to our framework to better understand its strengths and weaknesses.

We plan to explore two such extensions that we believe will allow our technique to perform significantly better on properties like FailSafeIter and other properties that follow the structure of a constrained-response pattern [34]. We will report our findings in the future.

Additionally, the number of monitors associated with the objects in the case of multi-object properties can be a major source of runtime overhead as discussed in Chapter 3. We present an analysis technique that we call monitor reclamation analysis in Section 6.2.1.1 and would like to implement that in the future. This analysis would target the monitors that do not need to be tracked as they can never reach the error state and reclaim those, thereby

limiting the number of monitors associated with the objects. Hence, this analysis would reduce the cost of monitor traversal and transitions (C_V and C_T as mentioned in Chapter 3) which would result in reduced runtime overhead.

Our implementation currently handles only two commonly occurring special cases of nested loops. Moreover, it does not support transformation of loops when the observables within loops are associated with possibly more than one receiver objects. This indicates that our results underestimate the potential overhead reduction that could be achieved given a more comprehensive engineering of our analysis implementation. We did not perform a detailed analysis of the optimization opportunities that were lost due to this limitation. However, we studied our log data for PMD-FailSafeIter and determined 19 nested loops were not optimized and 9 loops with multiple monitored objects were not optimized. Clearly, handling those loops would make it possible to achieve better overhead reduction, but for the Dacapo workloads those 28 loops contributed a total of just 24 of the 2.2 million events produced when performing monitoring; so for this program, property, and workload the lost optimization opportunity was negligible.

We intentionally performed a focused evaluation to target the most costly program-property pairs reported in the literature, but the scope of the evaluation is limited. Our findings should be interpreted as providing preliminary evidence that the number of events generated in runtime monitoring can be significantly reduced through the use of stutter-equivalent loop transformation. In the future, we would like to perform followup studies that consider more programs, properties, and a diversity of program inputs to establish the breadth of applicability and effectiveness of the technique.

Both of the analysis techniques that we developed including the residual analysis presented in Chapter 4 target the number of events. This target was chosen based on the insights provided by the cost models in Chapter 3, as it indirectly targets all the operational costs by skipping some events. Based on the evaluation, we see that these optimization

techniques indeed reduced the runtime overhead. The stutter-equivalent loop transformation was motivated by the weaknesses that we observed in the residual analysis technique. In particular, we had observed some safe regions being rejected by the analysis due to imprecision in it. We also observed that the cost missing those regions was high particularly when the regions were loops. The stutter-equivalent loop transformation technique provides a solution by transforming such a failed loop, if that stutters at distance k, into a sequence of $k + 1$ copies of the loop where all except the last copy may be *unsafe*. An execution of the loop in the original program is semantically equivalent to the execution of the loop of the whole sequence of the copies of the loop in the transformed program. The benefits of this techniques come from the fact that each of the potentially *unsafe* copy of the loop executes the instrumentation inside at the most once during the execution of the sequence of the copies of the loop.

5.9 Acknowledgements

This work has been performed in collaboration with my adviser Dr. Matthew Dwyer and Dr. Sebastian Elbaum. Dr. Madeline Diep contributed during the initial phase of conception and design. A paper based on this work has appeared in the Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010 [68]. Being the lead author, my contribution toward the conception, design, implementation and evaluation was greatest among all the co-authors. All co-authors contributed equally in the writing of this work.

Chapter 6

Conclusion and Future Work

This chapter summarizes the work that has been performed and presented in this dissertation. In addition, it also provides directions for the future work. In the future work, we provide an outline for the optimization techniques that we would like to develop in the near future.

6.1 Conclusion

In this dissertation, we presented our research that focuses on developing techniques to reduced the overhead of runtime monitoring without compromising error reporting. We presented our hybrid approach that combines static analysis with dynamic analysis and exploits program and property structures. The approach is motivated by the insights provided by the cost models for runtime monitoring presented in Chapter 3. We identified two basic algorithmic approaches used in the development of runtime monitoring tools and presented the cost models that describe the key factors that influence the runtime overhead for both approaches. We revisited some of the previous results and explained those based on the new perspective provided by the cost models. Our research is guided by the cost

models. The optimization techniques that we have developed as well those that we plan to develop in future target various cost factors described by the models and the targets have been prioritized based on the insights provided by the models.

In this dissertation, we presented two optimization techniques, namely, the residual typestate analysis and the stutter-equivalent loop transformation in Chapters 4 and 5 respectively. Both techniques target the number of events, but can be extended to target the number of monitors. We gave highest priority to developing techniques that target the number of events as doing so would indirectly target all the factors that appear in the cost definitions of the models. The residual analysis identifies safe regions in the program that may not contribute to property violations. The analysis then drops the instrumentation inside those regions and, if required, adds an instrumentation that triggers a summary transition at the of the region. Although effective as shown by its evaluation, we observed two weaknesses of this analysis technique. The first weakness was the high cost of static analysis as it requires expensive whole-program analysis. The second weakness was its imprecision that may fail to identify a region as safe, even when it is. This may result into higher monitoring overhead particularly when the region is a loop, as observable statements inside a loop may get executed frequently. We addressed both of these weaknesses by developing an optimization technique based on the transformation of stuttering loops. This transformation aims at minimal generation of events occurring within loops while still ensuring correct error reporting. Both analyses generate correct results except when certain conditions are met in the presence of unchecked exceptions. We explained the conditions that may affect the analysis correctness in Sections 4.3 and 5.6. Due to the reasons explained in Section 4.3, we believe that these scenarios would be infrequent in practice. The results of the evaluation of the techniques using some open source benchmarks and applications show that the techniques can effectively reduce the runtime monitoring overhead for most of the program and property combinations.

In the future, we would like to target other key factors that influence runtime monitoring overhead for optimization. In the following section, we outline a technique that targets a factor that impacts the cost of monitor traversal and transition. We also provide an outline for a technique that may improve the effectiveness of residual as well as loop transformation analysis techniques by targettng conditional program regions.

6.2 Future Work

In spite of the optimization that we have achieved using the two analysis techniques presented in Chapters 4 and 5, we have occasionally observed that monitoring may still incur unacceptably high overhead especially for multi-object properties. Here we describe a novel optimization technique that may limit the monitoring overhead in the case of multi-object properties by targeting the factors that impact the cost of monitor traversal and transition. We also provide an outline for a technique that may improve the effectiveness of residual as well as loop transformation analysis techniques by identifying target program regions conditionally. In addition, we describe a monitoring approach that we call *symbol-based* and propose an optimization that can be performed using this approach for efficient monitoring. We plan to conduct a study that would include wider range of properties and benchmarks. At the end, we provide some interesting research directions that extend the current scope of dynamic analysis.

6.2.1 A Novel Optimization Technique

The results of our previous studies presented in Sections 4.5 and 5.7 as well the results of the partial validation of the models, indicated that apart from the number of events, the cost of monitor traversal and transitions may severely affect the runtime overhead. This cost may be controlled, in the case of multi-object properties, if the number of monitors associ-

ated with objects can be controlled by eliminating the monitors that are no longer needed. This is shown by the θ, η and β related terms in the cost model definitions of C_V and C_T in Figures 3.7 and 3.8. The monitor reclamation analysis introduced in Section 6.2.1.1 will target these unwanted monitors. This technique is based on our hybrid approach and would perform optimizations based on the results of a static analysis.

6.2.1.1 Monitor Reclamation Analysis

We illustrate the main idea behind the technique using the same code snippet that we used in Figure 1.3 and the property `FailSafeIter` shown in Figure 1.2. For the code snippet, the monitor reclamation analysis would first perform an escape analysis that would infer that after the loop is executed, the reference variable $iter$ becomes dead as it is never read later. Moreover, the object referenced by $iter$ does not escape the method, hence it becomes unreachable at the end of the method. At this point, the analysis can infer from the FSA of the property `FailSafeIter` that, a monitor may never reach the error state, irrespective of its state, as it will never observe `next` events in future.

Figure 6.1 shows the output of the analysis that has all relevant method calls instrumented. In addition it has an extra statement `OBSERVE.reclaimMonitor(iter)` at the end of the loop, that advises the monitoring system about reclamation of the monitor. We would also like to explore the strategy in which the system may even reuse a monitor by resetting and pushing it back to the free monitor object pool. We hope that this approach would reduce the burden of memory management by reusing existing monitors and would also reduce monitor creation time that is indicated by C_C terms in Figures 3.7 and 3.8.

The goal of this analysis is similar to the goal of *memory leak elimination* proposed by Avgustinov et al. [8]. Their technique is completely dynamic which eliminates the cost of static analysis. However, the technique in its current form can be applied only to state-driven monitoring tools. Moreover, it may result in higher overhead during runtime,

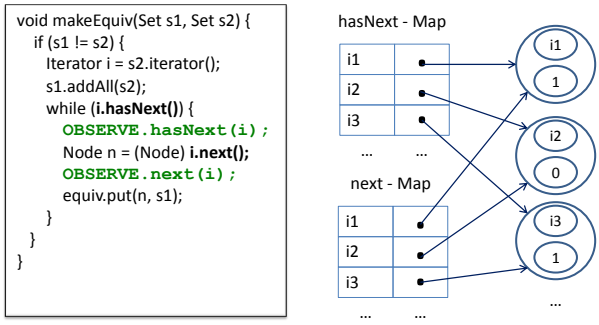
```
void makeEquiv(Node node1, Node node2) {
  Set s1 = equivalent(node1);
  Set s2 = equivalent(node2);

  if (s1 != s2) {

     Iterator iter = s2.iterator();
     OBSERVE.monitor(s2, iter, "create");

     while (iter.hasNext()) {
         Node n = (Node) iter.next();
         OBSERVE.monitor(iter, "next");
         equiv.put(n, s1);
     }
     OBSERVE.reclaimMonitor(iter);

  }
}
```

Figure 6.1: Monitor Reclamation.

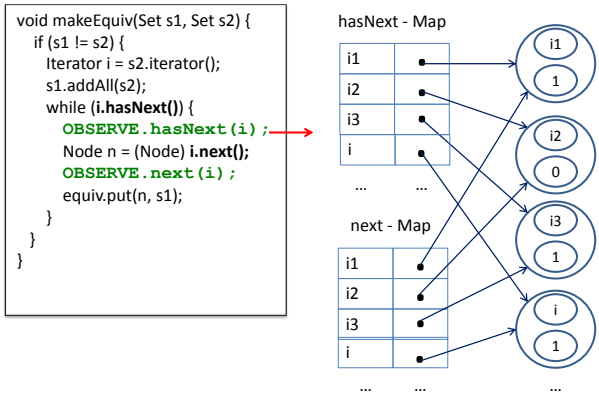
as disjuncts need to be freshly created when states are changed and if required, reordered. Additionally, its performance is VM dependent, that is, the unwanted disjuncts are eliminated by a garbage collector, and hence it depends on how frequently it is invoked. Interestingly, as mentioned by the authors, systems with larger heap suffer more than the ones with smaller heap as a garbage collector is typically invoked when less memory is available. Our newly proposed technique can overcome all these deficiencies mainly at the cost of static analysis. However, we believe that runtime cost is more crucial to programmers and our technique has only a small runtime overhead.

6.2.2 Symbol-based Monitoring

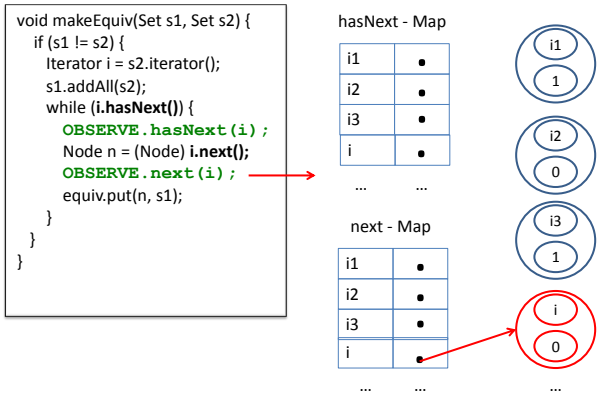
In Chapter 3, we discussed two algorithmic approaches to monitoring, namely object-based and state-based. Our choice for this division was mainly driven by the state-of-art monitoring tools that are based on one of the two approaches. In particular, JavaMOP uses



(a) Before the generation of a creation event

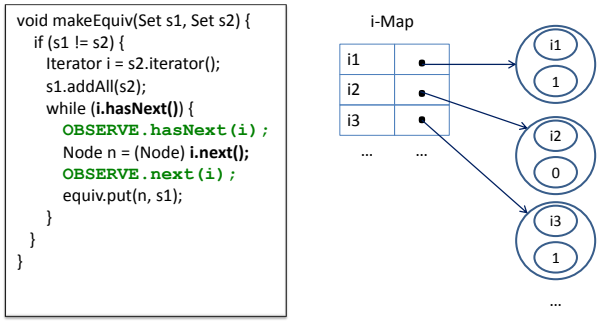


(b) After the generation of a creation event

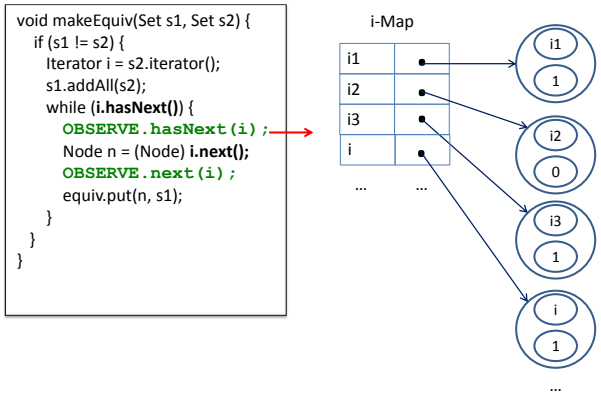


(c) After the generation of event `next`

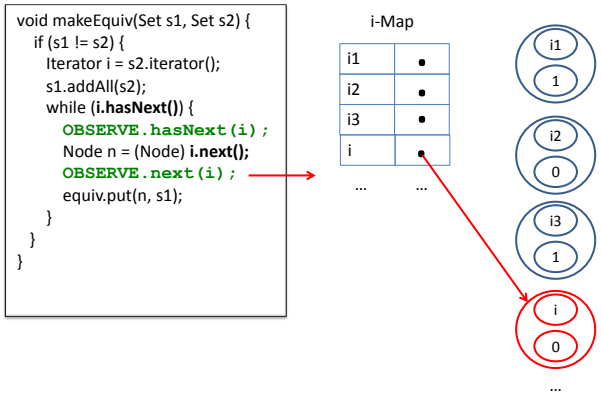
Figure 6.2: Simple symbol-based monitoring scheme for complete matching.



(a) Before the generation of a creation event



(b) After the generation of a creation event



(c) After the generation of event `next`

Figure 6.3: Simple object-based monitoring scheme for complete matching.

object-based approach and Tracematches is an implementation of state-based approach. Here, we propose an orthogonal approach that is based on symbols. The intuition behind our approach comes from the fact that if an object-based monitoring tool could distinguish between the monitors based on the states, then it would track the associated monitors only if the input symbol is going to change its state. In other words, the tool can then directly exploit the property structure by skipping the symbols that result in self-loops. Hence, we felt a need for a symbol-based monitoring approach that would use symbol-specific index trees to keep monitors based on their states. We would like to illustrate our approach with an example.

We use the same code snippet in Figure 6.2 that we used in Section 3.2.3. However, we illustrate our new approach using property `HasNext` shown in Figure 2.3 as this property would highlight the key differences between this approach and other approaches discussed in Chapter 3. For property `HasNext`, `hasNext` and `next` are the relevant symbols. Figure 6.3(a) shows the monitoring data structures before the execution of a creation call, which in this case would be a call to method `hasNext`. The call generates a monitoring *event* which consists of a tuple (l, b) where l is a set of associated objects, in this case $\{i\}$, and b is an FSA symbol, which in this case is `hasNext`. The scheme provides two maps corresponding to two symbols. The maps, as we have seen before in Section 3.2.3, are keyed by sets of related objects that have been involved in previous events. We see that both maps have the same set of keys, because both symbols are related to the same sets of objects. Note that, if we were monitoring for a multi-object property such as `FailSafeIter` that has symbols associated with different sets of objects, then the maps would have different keys.

Figure 6.3(b) shows the situation after the scheme handles the creation event. Since no monitors are associated with iterator `i`, a new monitor is created and references to it are associated with a new key `i` in both maps.

Figure 6.3(c) shows how a subsequent `next(i)` event, which is triggered by a call to `i.next()`, is handled. The map associated with symbol `next` is accessed and the monitors (in this case only 1) associated with `i` in the map are retrieved. For each such monitor, an FSA transition is simulated to update the state. In this example, the monitor associated with `i` was previously in state 1 so it is updated to $\delta(1, \texttt{next}) = 0$ based on the FSA in Figure 2.3.

To contrast this approach with object-based monitoring approach, we show how the object-based approach would handle the same events in Figure 6.3. The main difference here is the usage of only one map irrespective of the number of symbols as all symbols are associated with only one kind of objects i.e. iterator objects.

An object-based approach with optimized indexing has a set of sets of related objects, each member of which serves as a key to one index tree. The set is partitioned among different trees that correspond to symbols for efficient monitor access. In other words, there is only one index tree for symbols that have common sets of objects. In contrast, the symbol-based approach would keep all the sets of objects that are related to a symbol in the index tree associated with the symbols. This means that multiple index trees may have a few common keys. The symbols will share the trees only if their behavior with respect to the property is identical. For example, in the property `SocketChannel` shown in Figure 1.1 has symbols `read` and `write` whose behavior with respect to the property is identical and hence they can share their index trees.

It is clear that when compared with the object-based approach, symbol-based approach may need to perform the extra work to update multiple symbols trees during creation events. So an obvious question is *how can symbol-based approach be useful?* We provide an answer to this question in Section 6.2.2.1. We describe an optimization technique that leverages the fact that the symbol-based approach can maintain separate monitor lists and

the lists can be manipulated depending on characteristics of a property. That means that symbol-based approach allows a useful optimization that other approaches cannot.

6.2.2.1 Targeting Property Self-loops

The cost of monitoring can be controlled if traversals and transitions are performed only when required, that is only when, these operations may change the state of monitors. As we discussed in Chapter 5, the looping sequences of transitions cannot contribute to an error and hence need not be observed. A majority of the events during program execution may generate such sequences of transitions and hence can be skipped while monitoring. The stutter-equivalent transformation technique targets such events when they are generated by program loops and as the results show, can effectively control the overhead for many of the program-property combinations. However, we noticed that an imprecise points-to analysis may result in rejecting a valid loop transformation for transformation because the analysis may fail to show the relationship among associated objects. The analysis may also reject a valid loop, because it might consider infeasible start states in the computation that a program may never reach. Moreover, it does not consider stuttering events that are generated by recursion and not by loops. All of these weaknesses may result in retaining instrumentation over some observable statements that otherwise could have been dropped.

In order to deal with these deficiencies, we propose an optimization technique based on symbol-based monitoring that would effectively control the extra cost of monitor traversals and transitions, which would otherwise be incurred due to stuttering events. We demonstrate the technique with the help of property `FailSafeIter` in Figure 1.2. We propose the technique for a symbol-based monitoring tool that supports complete centralized indexing.

It is obvious that for every state in the property FSA shown in Figure 1.2, only the outgoing transitions that do not self-loop may change the state. For example, in state 1, only

`update` transitions may change the state and for state 2 only `next` transitions may change the state. This means that, we only need to have the indexing support for the monitors with respect to outgoing transitions. Hence, in symbol-based monitoring, we maintain an index tree for every symbol. This is in contrast to the object-based monitoring where indexing would be provided only to the related set of objects. In other words, an object-based monitoring indexing scheme would keep a common index tree for two symbols that are related to the same set of objects, whereas a symbol-based monitoring indexing scheme will keep two separate trees. For example, for property `HasNext` in Figure 2.3, an object-based monitoring approach would keep only one index tree for both observables `hasNext` as well as `next`, as both symbols are related to an iterator object. However, a symbol-based monitoring would keep two trees corresponding to symbols `hasNext` and `next`. In the case of property `FailSafeIter`, an object-based monitoring tool would keep 3 maps; for a collection, an iterator, and a pair of collection and iterator. In this case, symbol-based monitoring approach would also result in the same 3 maps as there are only three symbols in this property. A transition in the case of optimized symbol-based monitoring would involve adding entries to the maps that are associated with outgoing transitions of the target state and deleting monitor entries from the currently active maps that will not be active in the target state. It should be noted here that if there are multiple monitors associated with an event, the cost of performing a transition would be linear with respect to the product of the number of monitors and the symbols that are affected in both, source and target states.

Figure 6.4 shows how the maps will be maintained and a transition would be performed. For simplicity, we have shown only two maps, one corresponding to `update` and the other corresponding to `next`, and the transition from state 1 to 2 after receiving an `update` event on collection c is shown.

After receiving an `update` symbol related to a collection, the monitoring tool would first retrieve the associated monitors by performing a lookup on the index tree associated

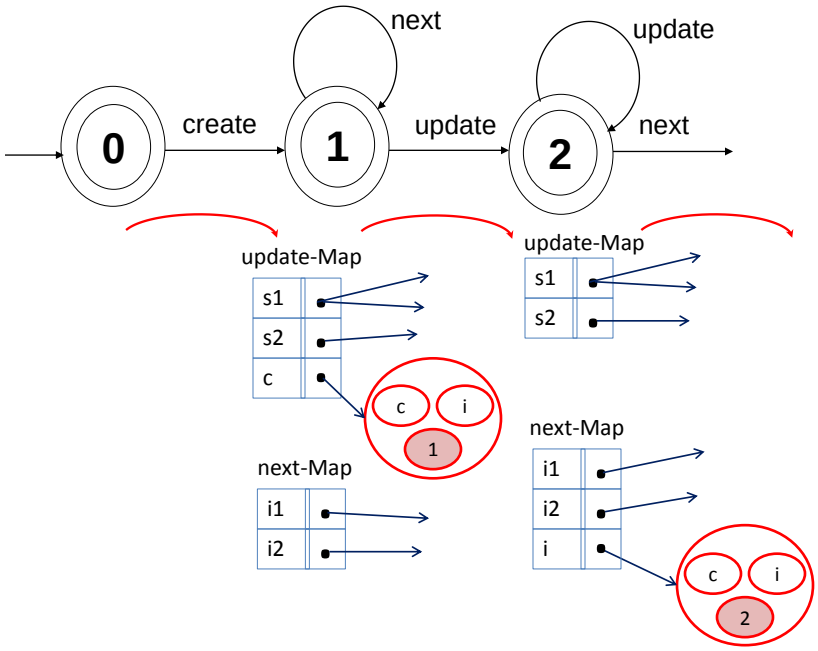


Figure 6.4: Transitioning from state 1 to state 2.

with the incurred symbol. For each one of the monitors, it would check the target state and update the relevant index tree corresponding to the non-self-looping outgoing transitions at the target state. In this case, symbol `update` is received by a collection c. The tool would retrieve monitors associated with c, which in this case is only one with associated iterator i. Since `update` symbol would push the monitor to state 2, the tool checks the non-self-looping outgoing transitions at 2 which in this case is `next`. The tool then inserts an entry corresponding to iterator i in the index tree that corresponds to `next` and deletes the entry for collection c from the tree that corresponds to `update` as shown in Figure 6.4.

If the monitoring tool after receiving an event cannot find an entry for the related objects in the corresponding index tree, it will assume that updating a state, that is performing a transition, is not required as it only self-loops. It will perform a transition only when the lookup is successful, that is when it finds any associated monitors. For clarity, we have

shown in Figure 6.4 only one monitor associated with collection object. In practice there could be more monitors associated with collection objects. It is easy to see that the rewards for optimization are higher, when the monitoring tool skips transitions for objects that have more than one monitor associated.

Unlike stutter-equivalent loop transformation technique, this optimization comes with a non-zero but small cost for referencing. However, an update corresponding to non-self-looping transitions could be expensive depending on the number of associated monitors and symbols. Moreover, it may only optimize self-loops and not any arbitrary loops of length greater than one. However, it can optimize repeating symbols irrespective of their source in the program. In addition, it does not incur an extra cost of expensive static program analysis as it only needs to analyze the property structure. In general, we expect it to be effective for the properties that have frequently occurring self-looping symbols. For example, the technique may be more effective for properties like `FailSafeIter` that have self-loops and change their states very infrequently. At the same time it may not be effective for properties like `HasNext` that continuously change the monitor states. Hence, it would be useful to develop a program or property analysis that can predict the effectiveness of this technique.

Dwyer et al.'s technique [36] can be seen as an instance of symbol-based approach, that drops the symbol instrumentation dynamically when there are no monitors interested in the symbol. Compared to their technique approach, our new technique is more light-weight and does not need any additional infrastructure or dynamic instrumentation that Dwyer et al.'s technique needs. Moreover, unlike Dwyer et al.'s technique, it targets individual monitors that are not interested in a symbol. That means it would track only those monitors that would change their states for the incurred symbol. On the other hand Dwyer et al.'s technique would disable the instrumentation only when there is no monitor that would change its state for the incurred symbol. A program execution that generates a large number

```
void foo(Set<Item> s1, Set<Item> s2) {
    int len = 0;

    while(len < MAXSIZE) {
        Item item = new Item(len);
        s1.add(item)*; // inst
        if(!item.isStandard())
            s2.remove(item); // inst
        len++;
    }
}
```

Figure 6.5: Conditionally transformed loop: original code.

of monitors, such a scenario would be infrequent. Due to these reasons, we expect our new approach to be more effective than Dwyer et al.'s technique.

6.2.3 Improvement Techniques

The results of the study presented in Section 5.7 indicate that although the technique was effective for most of the program and property combinations, a few combinations still incurred a high overhead. Our investigation revealed that many of the failed loops were discarded by the analysis due to an imprecision in the pointer analysis. As we mentioned earlier, it is very difficult to have a highly precise static analysis that scales to real software systems. However, an object aliasing can be checked at a very small cost during runtime. Here, we try to exploit the hybrid nature of our approach and first use the conclusive *must-alias* results statically. However, the conditions that remained unresolved due to imprecision in the analysis would be added back to the program in the form of predicates, whenever possible, and would be checked dynamically. If such a condition is satisfied, control flow will be directed toward an optimized transformed loop; if not, the program would execute as it would have executed normally taking the unoptimized path. This technique can also be applied to residual analysis to find conditional safe regions, that behave

```
void foo(Set<Item> s1, Set<Item> s2) {
    int len = 0;

    if(s1 == s2){
        while(len < MAXSIZE) {
            Item item = new Item(len);
            s1.add(item)*;              // inst
            if(!item.isStandard())
                s2.remove(item);        // inst
            len++;
            break;
        }
        while(len < MAXSIZE) {
            Item item = new Item(len);
            s1.add(item);               // uninst
            if(!item.isStandard())
                s2.remove(item);        // uninst
            len++;
            break;
        }
    }else{
        while(len < MAXSIZE) {
            Item item = new Item(len);
            s1.add(item)*;              // inst
            if(!item.isStandard())
                s2.remove(item);        // inst
            len++;
        }
    }
}
```

Figure 6.6: Conditionally transformed loop: transformed code.

like safe regions under certain conditions that are checked by added predicates. Moreover, it can be extended to have predicates based on typestates, so that a condition that failed the analysis due to an infeasible start state, can be checked dynamically for better optimization.

The technique is illustrated using an example in Figure 6.5. The property that we are checking is `FailSafeIter`. The technique performs a static analysis that is bottom-up and a pointer analysis that is intraprocedural. Our loop transformation analysis would fail here, if it cannot conclude whether $s1$ would point $s2$ or not. If $s1$ does not point to $s2$,

then symbol `update` may be encountered by both objects, that is the one referenced by $s1$ and the other referenced by $s2$. The conditional statement inside the loop indicates that the symbol may not be encountered by the object referenced by $s2$ during the first iteration of the loop, and hence, the loop transformation may not be safe. However, the analysis can still conclude that the references do not change within the loop and if they are aliases of each other, the loop would be safe to transform. Our proposed analysis would use this partial but useful result to generate the condition "s1 == s2" as shown in Figure 6.6 by the `if-else` statement enclosing the loop and directs the control-flow accordingly. That means if the condition is satisfied the control will be directed toward optimized path, otherwise toward unoptimized path.

We expect the technique to improve the effectiveness of both, the residual analysis as well as the loop transformation analysis, at a small cost of additional instrumentation required to test the predicates.

The future work also includes improvements mentioned in this dissertation such as extending the analyses to target the number of monitors and perform property simulation.

6.2.4 Broader Study

Most of the current research [19, 20, 25, 62, 68] is based on evaluation using DaCapo benchmarks and properties that specify legal rules for using Java API. It is evident from the previous studies, that monitoring overhead is influenced by the factors related to the program and property interaction. Hence, even the properties with similar structure may incur different overheads depending on their application usage. Although interesting in many ways, these studies are narrow and indicate a need for more richness and diversity particularly in properties. A question that a researcher may ask is *what properties really matter?*.

Before we attempt to answer this question, we need a formal way of characterizing properties. What are the attributes of a property that may be of interest? We provide a list of some attributes that may make one property different than another as well as more interesting than another.

a) Structure: A property monitoring may be influenced by the structural attributes of the property. This may include the number of states, the number of symbols, the number of self-looping symbols, being related to multiple objects or expressed by a richer formalism such as context free grammars. A monitoring approach such as object-based monitoring may not be sensitive to the number of states, but another approach such as state-based monitoring may be. Similarly, state-based monitoring may be sensitive to self-looping symbols. The properties that have been recently studied by the researchers [19, 20, 25, 62, 68] have typically less than 4 or 5 states and a similar number of symbols. It is possible that the real properties that would matter have a much larger number of states and symbols. The answer to the question of whether the current properties that have been studied or being studied matter or not would also be influenced by the practitioners who will start using these techniques and tools in the future.

b) Program Behavior: Generally speaking, faults are hard to detect when they cause property failures only during certain program executions or for certain program states. Such faults may easily go undetected during the testing phase. Based on this intuition, we broadly divide properties into following categories based on the program behavior at the time of a failure: i. the program *always* throws an exception, ii) semantics are not clearly defined, and iii) the program throws an exception depending on its state defined by variable values. We see clearly that the faults that are related to the last category of properties may be hardest to detect as the program may behave normally for most of the states and hence, such faults may get undetected during the testing phase. On the other hand, program may invariably throw an exception for certain types of violations and such faults

are more likely to get detected during testing. Hence, we feel that properties that properties that fall under second and third categories are likely candidates for runtime monitoring.

6.2.5 Beyond Error Detection

Detecting faults during runtime that may lead to failures is certainly helpful. However, a question that a researcher may ask at this point is *can we do more?*. It will be very useful if dynamic analysis can not only detect a fault, but also steer the program execution to avoid a failure? This would be particularly useful for mission-critical applications where avoiding a failure would be of utmost importance. Similarly an approach could be to diagnose faults and a timely proactive recovery that would keep the system under control by avoiding unexpected failure [38]. An interesting question would be to find out the information that needs to be generated during runtime for this kind of advanced dynamic analysis, especially considering the resource constraints and functional requirements of these software applications. These are some challenging future research directions that would be critical to the success of mission-critical systems.

Appendix A

Java programs used in the study with JavaMOP

In this appendix, we provide the Java programs that we used in the study with JavaMOP presented in Section 3.5.

```
public class ChangeThetaUni1{

    public static void main(){
        long startTime = System.nanoTime()/1000;
        Set<Integer> s1 = new HashSet<Integer>();

        for (int i = 0; i < 1000000; i++){
            s1.add(i);
        }
        for (int i = 0; i < 100; i++){
            Iterator<Integer> iter = s1.iterator();
            while (iter.hasNext()){
                int someint = (Integer) iter.next();
                foo(someint);
            }
        }
        long endTime = System.nanoTime()/1000;
            System.out.println("TimeTakenForExecution:"
                                + (endTime - startTime));
    }

    static void foo(int  i){
        if (i == -10){
            System.err.println(-10);
        }
    }
}
```

Figure A.1: Changing the number of associated monitors keeping the number of uni-monitor events same. Scale = 1.

```
public class ChangeThetaUni10
{
    public static void main(){
        long startTime = System.nanoTime()/1000;
        Set<Integer> s1 = new HashSet<Integer>();

        for (int i = 0; i < 100000; i++){
            s1.add(i);
        }
        for (int i = 0; i < 1000; i++){
            Iterator<Integer> iter = s1.iterator();
            while (iter.hasNext()){
                int someint = (Integer) iter.next();
                foo(someint);
            }
        }
        long endTime = System.nanoTime()/1000;
        System.out.println("TimeTakenForExecution:"
                                    + (endTime - startTime));
    }

    static void foo(int i){
        if (i == -10){
            System.err.println(-10);
        }
    }
}
```

Figure A.2: Changing the number of associated monitors keeping the number of uni-monitor events same. Scale = 10.

```
public class ChangeThetaMulti1{

    public static void main(){
        long startTime = System.nanoTime()/1000;
        Set<Integer> s1 = new HashSet<Integer>();
        Set<Integer> s2 = new HashSet<Integer>();

        for (int i = 0; i < 1; i++){
                s1.add(i);
        }
        for (int i = 0; i< 1; i++){
                s2.add(i);
        }
        for (int i = 0; i < 10; i++){
                Iterator<Integer> iter = s1.iterator();
                if (iter.hasNext()){
                        foo(1);
                }
        }
        for (int i = 0; i < 99990; i++) {
                Iterator<Integer> iter = s2.iterator();
                if (iter.hasNext()){
                        foo(1);
                }
        }
        for (int i = 0 ; i < 1000000; i++){
                s1.add(i);
        }
        long endTime = System.nanoTime()/1000;
        System.out.println("TimeTakenForExecution:"
                                + (endTime - startTime));
    }

    static void foo(int i){
        if (i == -10){
            System.err.println(-10);
        }
    }
}
```

Figure A.3: Changing the number of associated monitors keeping the number of multi-monitor events same. Scale = 1.

```
public class ChangeThetaMulti10{

    public static void main(){
        long startTime = System.nanoTime()/1000;
        Set<Integer> s1 = new HashSet<Integer>();
        Set<Integer> s2 = new HashSet<Integer>();

        for (int i = 0; i < 1; i++){
                s1.add(i);
        }
        for (int i = 0; i< 1; i++){
                s2.add(i);
        }
        for (int i = 0; i < 100; i++){
                Iterator<Integer> iter = s1.iterator();
                if (iter.hasNext()){
                        foo(1);
                }
        }
        for (int i = 0; i < 99900L; i++) {
                Iterator<Integer> iter = s2.iterator();
                if (iter.hasNext()){
                        foo(1);
                }
        }
        for (int i = 0 ; i < 1000000; i++){
                s1.add(i);
        }
        long endTime = System.nanoTime()/1000;
        System.out.println("TimeTakenForExecution:"
                                + (endTime - startTime));
     }

     static void foo(int i){
        if (i == -10){
            System.err.println(-10);
        }
    }
}
```

Figure A.4: Changing the number of associated monitors keeping the number of multi-monitor events same. Scale = 10.

```
public class ChangeTransUni1{

    public static void main(){
        long startTime = System.nanoTime()/1000;
        Set<Integer> s1 = new HashSet();

        for (int i = 0; i < 1; i++){
                s1.add(i);
        }
        for (int i = 0; i < 10000; i++) {
                Iterator<Integer> iter = s1.iterator();
                while(iter.hasNext()){
                        int j = iter.next();
                        foo(j);
                }
        }
        long endTime = System.nanoTime()/1000;
        System.out.println("TimeTakenForExecution:"
                                + (endTime - startTime));
    }

    static void foo(int i){
        if (i == -10){
            System.err.println(-10);
        }
    }
}
```

Figure A.5: Changing the number of uni-monitor events keeping the number of associated monitors same. Scale = 1.

```
public class ChangeTransUni10{

    public static void main(){
        long startTime = System.nanoTime()/1000;
        Set<Integer> s1 = new HashSet();

        for (int i = 0; i < 10; i++){
                s1.add(i);
        }
        for (int i = 0; i < 10000; i++) {
            Iterator<Integer> iter = s1.iterator();
            while(iter.hasNext()){
                int j = iter.next();
                foo(j);
            }
        }
        long endTime = System.nanoTime()/1000;
        System.out.println("TimeTakenForExecution:"
                                    + (endTime - startTime));
    }

    static void foo(int i){
        if (i == -10){
            System.err.println(-10);
        }
    }
}
```

Figure A.6: Changing the number of uni-monitor events keeping the number of associated monitors same. Scale = 10.

```
public class ChangeTransMulti1{

    public static void main(){

        long startTime = System.nanoTime()/1000;
        Set<Integer> s1 = new HashSet();

        for (int i = 0; i < 1; i++){
                s1.add(i);
        }
        for(int i = 0; i < 10000; i++){
            Iterator<Integer> iter = s1.iterator();
            if (iter.hasNext()){
                foo(1);
            }
            for(int j = 0; j < 100; j++){
                foo(j);
            }
        }
        for (int i = 0; i < 100; i++){
            s1.add(i);
        }
        long endTime = System.nanoTime()/1000;
        System.out.println("TimeTakenForExecution:"
                                    + (endTime - startTime));
    }

    static void foo(int i){
        if (i == -10){
            System.err.println(-10);
        }
    }
}
```

Figure A.7: Changing the number of multi-monitor events keeping the number of associated monitors same. Scale = 1.

```
public class ChangeTransMulti10{

    public static void main(){

        long startTime = System.nanoTime()/1000;
        Set<Integer> s1 = new HashSet();

        for (int i = 0; i < 1; i++){
                s1.add(i);
        }
        for(int i = 0; i < 10000; i++){
            Iterator<Integer> iter = s1.iterator();
            if (iter.hasNext()){
                foo(1);
            }
            for(int j = 0; j < 100; j++){
                foo(j);
            }
        }
        for (int i = 0; i < 1000; i++){
            s1.add(i);
        }
        long endTime = System.nanoTime()/1000;
        System.out.println("TimeTakenForExecution:"
                                    + (endTime - startTime));
   }
    static void foo(int i){
        if (i == -10){
            System.err.println(-10);
        }
    }
}
```

Figure A.8: Changing the number of multi-monitor events keeping the number of associated monitors same. Scale = 10.

Appendix B

Java programs used in the study with Tracematches

In this appendix, we provide the Java programs that we used in the study with Tracematches presented in Section 3.5.

```
public class ChangeThetaUni1{

    public static void main(){

        long startTime = System.currentTimeMillis();
        LinkedList l1 = new LinkedList();
        LinkedList l2 = new LinkedList();
        LinkedList l3 = new LinkedList();
        LinkedList l4 = new LinkedList();

        for(int i = 0; i < 1000000; i++){
                l1.addLast(i);
        }
        for (int i = 0; i < 1; i++){
                l3.addLast(l1.iterator());
        }
        for (int i = 0; i < 9999; i++){
                l4.addLast(l2.iterator());
        }
        Iterator iter = (Iterator) l3.getLast();
        while (iter.hasNext()){
                foo((Integer) iter.next());
        }
        foo(l4.size());
        long endTime = System.currentTimeMillis();
        System.out.println("TimeTakenForExecution:"
                                + (endTime - startTime));
    }
    static void foo(int  i){
        if (i == -10){
            System.err.println(-10);
        }
    }
}
```

Figure B.1: Changing the number of associated monitors keeping the number of uni-monitor events same (Property 1). Scale = 1.

```
public class ChangeThetaUni10{

    public static void main(){

        long startTime = System.currentTimeMillis();
        LinkedList l1 = new LinkedList();
        LinkedList l2 = new LinkedList();
        LinkedList l3 = new LinkedList();
        LinkedList l4 = new LinkedList();

        for(int i = 0; i < 1000000; i++){
                l1.addLast(i);
        }
        for (int i = 0; i < 10; i++){
                l3.addLast(l1.iterator());
        }
        for (int i = 0; i < 9990; i++){
                l4.addLast(l2.iterator());
        }
        Iterator iter = (Iterator) l3.getLast();
        while (iter.hasNext()){
                foo((Integer) iter.next());
        }
        foo(l4.size());
        long endTime = System.currentTimeMillis();
        System.out.println("TimeTakenForExecution:"
                                + (endTime - startTime));
    }
    static void foo(int  i){
        if (i == -10){
            System.err.println(-10);
        }
    }
}
```

Figure B.2: Changing the number of associated monitors keeping the number of uni-monitor events same (Property 1). Scale = 10.

```
public class ChangeThetaMulti1 {

    public static void main(){

        long startTime = System.currentTimeMillis();
        LinkedList l1 = new LinkedList();
        LinkedList l2 = new LinkedList();
        LinkedList l3 = new LinkedList();
        LinkedList l4 = new LinkedList();

        for (int i = 0; i < 1; i++){
                l3.addLast(l1.iterator());
        }
        for (int i = 0; i < 9999; i++){
                l4.addLast(l2.iterator());
        }
        for (int i = 0; i < 1000000; i++){
                l1.add(i);
                if (!(l1.isEmpty())){
                    foo(l1.size());
            }
        }
        foo(l3.size());
        foo(l4.size());
        long endTime = System.currentTimeMillis();
        System.out.println("TimeTakenForExecution:"
                                    + (endTime - startTime));
    }
    static void foo(int  i){
        if (i == -10){
            System.err.println(-10);
        }
    }
}
```

Figure B.3: Changing the number of associated monitors keeping the number of multi-monitor events same (Property 1). Scale = 1.

```
public class ChangeThetaMulti10{

    public static void main(){

        long startTime = System.currentTimeMillis();
        LinkedList l1 = new LinkedList();
        LinkedList l2 = new LinkedList();
        LinkedList l3 = new LinkedList();
        LinkedList l4 = new LinkedList();

        for (int i = 0; i < 10; i++){
                l3.addLast(l1.iterator());
        }
        for (int i = 0; i < 9990; i++){
                l4.addLast(l2.iterator());
        }
        for (int i = 0; i < 1000000; i++){
                l1.add(i);
                if (! (l1.isEmpty())){
                    foo(l1.size());
            }
        }
        foo(l3.size());
        foo(l4.size());
        long endTime = System.currentTimeMillis();
        System.out.println("TimeTakenForExecution:"
                                    + (endTime - startTime));
    }
    static void foo(int  i){
        if (i == -10){
            System.err.println(-10);
        }
    }
}
```

Figure B.4: Changing the number of associated monitors keeping the number of multi-monitor events same (Property 1). Scale = 10.

```
public class ChangeTransUni1{

    public static void main(){

        long startTime = System.currentTimeMillis();
        LinkedList l1 = new LinkedList();
        LinkedList l2 = new LinkedList();

        for (int i = 0; i < 100; i++){
            l1.addLast(i);
        }
        for (int i = 0; i < 10000L; i++){
                l2.addLast(l1.iterator());
        }
        Iterator iter = (Iterator) l2.getLast();
        while (iter.hasNext()){
                foo((Integer) iter.next());
        }
        foo(l2.size());
        long endTime = System.currentTimeMillis();
        System.out.println("TimeTakenForExecution:"
                                + (endTime - startTime));
    }
    static void foo(int i){
        if (i == -10){
            System.err.println(-10);
        }
    }
}
```

Figure B.5: Changing the number of uni-monitor events keeping the number of associated monitors same (Property 1). Scale = 1.

```
public class ChangeTransUni10{

        public static void main(){

        long startTime = System.currentTimeMillis();
        LinkedList l1 = new LinkedList();
        LinkedList l2 = new LinkedList();

        for (int i = 0; i < 1000; i++){
                l1.addLast(i);
        }
        for (int i = 0; i < 10000L; i++){
                l2.addLast(l1.iterator());
        }
        Iterator iter = (Iterator) l2.getLast();
        while (iter.hasNext()){
                foo((Integer) iter.next());
        }
        foo(l2.size());
        long endTime = System.currentTimeMillis();
        System.out.println("TimeTakenForExecution:"
                                + (endTime - startTime));
    }
    static void foo(int i){
        if (i == -10){
            System.err.println(-10);
        }
    }
}
```

Figure B.6: Changing the number of uni-monitor events keeping the number of associated monitors same. Scale = 10.

```
public class ChangeTransMulti1{

    public static void main(){

        long startTime = System.currentTimeMillis();
        LinkedList l1 = new LinkedList();
        LinkedList l2 = new LinkedList();

        for (int i = 0; i < 10000L; i++){
                Iterator iter = l1.iterator();
                l2.addLast(iter);
        }
        for(int i = 0; i < 100; i++) {
                l1.add(i);
                if (!(l1.isEmpty())){
                        foo(l1.size());
                }
        }
        foo(l2.size());
        long endTime = System.currentTimeMillis();
        System.out.println("TimeTakenForExecution:"
                                + (endTime - startTime));
    }
    static void foo(int i){
        if (i == -10){
            System.err.println(-10);
        }
    }
}
```

Figure B.7: Changing the number of multi-monitor events keeping the number of associated monitors same (Property 1). Scale = 1.

```
public class ChangeTransMulti10{

    public static void main(){

        long startTime = System.currentTimeMillis();
        LinkedList l1 = new LinkedList();
        LinkedList l2 = new LinkedList();

        for (int i = 0; i < 10000L; i++){
                Iterator iter = l1.iterator();
                l2.addLast(iter);
        }
        for(int i = 0; i < 1000; i++) {
                l1.add(i);
                if (!(l1.isEmpty())){
                        foo(l1.size());
                }
        }
        foo(l2.size());
        long endTime = System.currentTimeMillis();
        System.out.println("TimeTakenForExecution:"
                                + (endTime - startTime));
    }
    static void foo(int i){
        if (i == -10){
            System.err.println(-10);
        }
    }
}
```

Figure B.8: Changing the number of multi-monitor events keeping the number of associated monitors same (Property 1). Scale = 10.

```
public class ChangeThetaUni1{

    public static void main(){

        long startTime = System.currentTimeMillis();
        LinkedList l1 = new LinkedList();
        LinkedList l2 = new LinkedList();
        LinkedList l3 = new LinkedList();
        LinkedList l4 = new LinkedList();

        for (int i = 0; i < 1000000; i++) {
                l1.addLast(i);
        }
        for (int i = 0; i < 1; i++){
                l3.addLast(l1.iterator());
        }
        for (int i = 0; i < 9999; i++){
                l4.addLast(l2.iterator());
        }
        Iterator iter = (Iterator) l3.getLast();
        while (iter.hasNext()){
                foo((Integer) iter.next());
        }
        foo(l4.size());
        long endTime = System.currentTimeMillis();
        System.out.println("TimeTakenForExecution:"
                                + (endTime - startTime));
    }
    static void foo(int i){
        if (i == -10){
            System.err.println(-10);
        }
    }
}
```

Figure B.9: Changing the number of associated monitors keeping the number of uni-monitor events same (Property 2). Scale = 1.

```
public class ChangeThetaUni10{

    public static void main(){

        long startTime = System.currentTimeMillis();
        LinkedList l1 = new LinkedList();
        LinkedList l2 = new LinkedList();
        LinkedList l3 = new LinkedList();
        LinkedList l4 = new LinkedList();

        for (int i = 0; i < 1000000; i++) {
                l1.addLast(i);
        }
        for (int i = 0; i < 10; i++){
                l3.addLast(l1.iterator());
        }
        for (int i = 0; i < 9990; i++){
                l4.addLast(l2.iterator());
        }
        Iterator iter = (Iterator) l3.getLast();
        while (iter.hasNext()){
                foo((Integer) iter.next());
        }
        foo(l4.size());
        long endTime = System.currentTimeMillis();
        System.out.println("TimeTakenForExecution:"
                                + (endTime - startTime));
    }
    static void foo(int i){
        if (i == -10){
            System.err.println(-10);
        }
    }
}
```

Figure B.10: Changing the number of associated monitors keeping the number of uni-monitor events same (Property 2). Scale = 10.

```
public class ChangeThetaMulti1{

    public static void main(){

        long startTime = System.currentTimeMillis();
        LinkedList l1 = new LinkedList();
        LinkedList l2 = new LinkedList();
        LinkedList l3 = new LinkedList();
        LinkedList l4 = new LinkedList();

        for(int i = 0; i < 1; i++){
                l3.addLast(l1.iterator());
        }
        for (int i = 0; i < 9999; i++){
                l4.addLast(l2.iterator());
        }
        for (int i = 0; i < 1000000; i++){
                l1.add(i);
                if (!(l1.isEmpty())){
                        foo(l1.size());
                }
        }
        foo(l3.size());
        foo(l4.size());
        long endTime = System.currentTimeMillis();
        System.out.println("TimeTakenForExecution:"
                                + (endTime - startTime));
    }
    static void foo(int i){
        if (i == -10){
            System.err.println(-10);
        }
    }
}
```

Figure B.11: Changing the number of associated monitors keeping the number of multi-monitor events same (Property 2). Scale = 1.

```
public class ChangeThetaMulti10{

    public static void main(){

        long startTime = System.currentTimeMillis();
        LinkedList l1 = new LinkedList();
        LinkedList l2 = new LinkedList();
        LinkedList l3 = new LinkedList();
        LinkedList l4 = new LinkedList();

        for(int i = 0; i < 10; i++){
                l3.addLast(l1.iterator());
        }
        for (int i = 0; i < 9990; i++){
                l4.addLast(l2.iterator());
        }
        for (int i = 0; i < 1000000; i++){
                l1.add(i);
                if (!(l1.isEmpty())){
                        foo(l1.size());
                }
        }
        foo(l3.size());
        foo(l4.size());
        long endTime = System.currentTimeMillis();
        System.out.println("TimeTakenForExecution:"
                                + (endTime - startTime));
    }
    static void foo(int i){
        if (i == -10){
            System.err.println(-10);
        }
    }
}
```

Figure B.12: Changing the number of associated monitors keeping the number of multi-monitor events same (Property 2). Scale = 10.

```
public class ChangeTransUni1{

    public static void main(){

        long startTime = System.currentTimeMillis();
        LinkedList l1 = new LinkedList();
        LinkedList l2 = new LinkedList();

        for (int i = 0; i < 100; i++){
                l1.addLast(i);
        }
        for (int i = 0; i < 10000; i++){
                l2.addLast(l1.iterator());
        }
        Iterator iter = (Iterator) l2.getLast();
        while (iter.hasNext()){
                foo((Integer) iter.next());
        }
        foo(l2.size());
        long endTime = System.currentTimeMillis();
        System.out.println("TimeTakenForExecution:"
                                + (endTime - startTime));
    }
    static void foo(int i){
        if (i == -10){
            System.err.println(-10);
        }
    }
}
```

Figure B.13: Changing the number of uni-monitor events keeping the number of associated monitors same (Property 2). Scale = 1.

```
public class ChangeTransUni10{

    public static void main(){

        long startTime = System.currentTimeMillis();
        LinkedList l1 = new LinkedList();
        LinkedList l2 = new LinkedList();

        for (int i = 0; i < 1000; i++){
                l1.addLast(i);
        }
        for (int i = 0; i < 10000; i++){
                l2.addLast(l1.iterator());
        }
        Iterator iter = (Iterator) l2.getLast();
        while (iter.hasNext()){
                foo((Integer) iter.next());
        }
        foo(l2.size());
        long endTime = System.currentTimeMillis();
        System.out.println("TimeTakenForExecution:"
                                + (endTime - startTime));
    }
    static void foo(int i){
        if (i == -10){
            System.err.println(-10);
        }
    }
}
```

Figure B.14: Changing the number of uni-monitor events keeping the number of associated monitors same (Property 2). Scale = 10.

```
public class ChangeTransMulti1{

    public static void main(){

        long startTime = System.currentTimeMillis();
        LinkedList l1 = new LinkedList();
        LinkedList l2 = new LinkedList();

        for (int i = 0; i < 10000L; i++){
            Iterator iter = r1.iterator();
            l2.addLast(iter);
        }
        for (int i = 0; i < 100; i++){
            l1.add(i);
            if(!(l1.isEmpty())){
                foo(l1.size());
            }
        }
        foo(l2.size());
        long endTime = System.currentTimeMillis();
        System.out.println("TimeTakenForExecution:"
                                    + (endTime - startTime));
    }
    static void foo(int i){
        if (i == -10){
            System.err.println(-10);
        }
    }
}
```

Figure B.15: Changing the number of multi-monitor events keeping the number of associated monitors same (Property 2). Scale = 1.

```
public class ChangeTransMulti10{

    public static void main(){

        long startTime = System.currentTimeMillis();
        LinkedList l1 = new LinkedList();
        LinkedList l2 = new LinkedList();

        for (int i = 0; i < 10000L; i++){
            Iterator iter = r1.iterator();
            l2.addLast(iter);
        }
        for (int i = 0; i < 1000; i++){
            l1.add(i);
            if(!(l1.isEmpty())){
                foo(l1.size());
            }
        }
        foo(l2.size());
        long endTime = System.currentTimeMillis();
        System.out.println("TimeTakenForExecution:"
                                    + (endTime - startTime));
    }
    static void foo(int i){
        if (i == -10){
            System.err.println(-10);
        }
    }
}
```

Figure B.16: Changing the number of multi-monitor events keeping the number of associated monitors same (Property 2). Scale = 10.

Bibliography

[1] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 25–34, New York, NY, USA, 2007. ACM.

[2] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 345–364, New York, NY, USA, 2005. ACM.

[3] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, pages 4–16, New York, NY, USA, 2002. ACM.

[4] Glenn Ammons, David Mandelin, Rastislav Bodík, and James R. Larus. Debugging temporal specifications with concept analysis. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 182–195, New York, NY, USA, 2003. ACM.

[5] Matthew Arnold, Martin Vechev, and Eran Yahav. Qvm: an efficient runtime for detecting defects in deployed systems. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 143–162, New York, NY, USA, 2008. ACM.

[6] Pavel Avgustinov, Eric Bodden, Elnar Hajiyev, Laurie J. Hendren, Ondrej Lhoták, Oege de Moor, Neil Ongkingco, Damien Sereni, Ganesh Sittampalam, Julian Tibble, and Mathieu Verbaere. Aspects for trace monitoring. In *FATES/RV*, pages 20–39, 2006.

[7] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible AspectJ compiler. In *Proceedings of the 4th international conference on Aspect-oriented software development*, AOSD '05, pages 87–98, New York, NY, USA, 2005. ACM.

[8] Pavel Avgustinov, Julian Tibble, and Oege de Moor. Making trace monitors feasible. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 589–608, New York, NY, USA, 2007. ACM.

[9] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2009.

[10] Nels E. Beckman, Kevin Bierhoff, and Jonathan Aldrich. Verifying correct usage of atomic blocks and typestate. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 227–244, New York, NY, USA, 2008. ACM.

[11] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 301–320, New York, NY, USA, 2007. ACM.

[12] Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. Practical API protocol checking with access permissions. In *Genoa: Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, pages 195–219, Berlin, Heidelberg, 2009. Springer-Verlag.

[13] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Comput.*, 21:592–597, June 1972.

[14] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.

[15] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, SPAA '91, pages 3–16, New York, NY, USA, 1991. ACM.

[16] Eric Bodden. *Verifying Finite-State Properties of Large-Scale Programs*. PhD thesis, McGill University, June 2009.

[17] Eric Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 5–14, New York, NY, USA, 2010. ACM.

[18] Eric Bodden, Laurie Hendren, Patrick Lam, Ondřej Lhoták, and Nomair A. Naeem. Collaborative runtime verification with tracematches. In *Proceedings of the 7th international conference on Runtime verification*, RV'07, pages 22–37, Berlin, Heidelberg, 2007. Springer-Verlag.

[19] Eric Bodden, Laurie Hendren, and Ondřej Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *Euro. Conf. on Obj. Oriented Prog.*, pages 525–549, July 2007.

[20] Eric Bodden, Patrick Lam, and Laurie Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 36–47, New York, NY, USA, 2008. ACM.

[21] Eric Bodden, Patrick Lam, and Laurie Hendren. Clara: a framework for partially evaluating finite-state runtime monitors ahead of time. In *Proceedings of the First international conference on Runtime verification*, RV'10, pages 183–197, Berlin, Heidelberg, 2010. Springer-Verlag.

[22] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. Abcd: eliminating array bounds checks on demand. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 321–333, New York, NY, USA, 2000. ACM.

[23] Guillaume Brat and Roger Klemm. Static analysis of the mars exploration rover flight software. In *Proceedings of the 1st International Space Mission Challenge for Information Technology*, pages 321–326, 2003.

[24] Guillaume Brat and Arnaud Venet. Precise and scalable static program analysis of NASA flight software. In *Proceedings of the 2005 IEEE Aerospace Conference*, 2005.

[25] Feng Chen and Grigore Roşu. Java-MOP: A monitoring oriented programming environment for Java. In *Int'l. Conf. Tools Alg. Const. Anal. Sys.*, LNCS, 2005.

[26] Feng Chen and Grigore Roşu. Mop: an efficient and generic runtime verification framework. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 569–588, New York, NY, USA, 2007. ACM.

[27] Marcelo d'Amorim and Klaus Havelund. Event-based runtime verification of Java programs. In *Proceedings of the third international workshop on Dynamic analysis*, WODA '05, pages 1–7, New York, NY, USA, 2005. ACM.

[28] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 57–68, New York, NY, USA, 2002. ACM.

[29] Robert Deline and Manuel Fähndrich. Typestates for objects. In *ECOOP'04: Proceedings of 18th European Conference on Object-Oriented Programming*, pages 465–490. Springer, 2004.

[30] Isil Dillig, Thomas Dillig, and Alex Aiken. Fluid updates: Beyond strong vs. weak updates. In *Proceedings of the 19th European Symposium on Programming*, ESOP '10, pages 246–266, 2010.

[31] Isil Dillig, Thomas Dillig, and Alex Aiken. Reasoning about the unknown in static analysis. *Commun. ACM*, 53:115–123, August 2010.

[32] Nurit Dor, Stephen Adams, Manuvir Das, and Zhe Yang. Software validation via scalable path-sensitive value flow analysis. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software testing and analysis*, ISSTA '04, pages 12–22, New York, NY, USA, 2004. ACM.

[33] M. B. Dwyer, M. Diep, and S. Elbaum. Reducing the cost of path property monitoring through sampling. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 228–237, Washington, DC, USA, 2008. IEEE Computer Society.

[34] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 411–420, New York, NY, USA, 1999. ACM.

[35] Matthew B. Dwyer, Lori A. Clarke, Jamieson M. Cobleigh, and Gleb Naumovich. Flow analysis for verifying properties of concurrent software systems. *ACM Trans. Softw. Eng. Methodol.*, 13(4):359–430, 2004.

[36] Matthew B. Dwyer, Alex Kinneer, and Sebastian Elbaum. Adaptive online program analysis. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 220–229, Washington, DC, USA, 2007. IEEE Computer Society.

[37] Matthew B. Dwyer and Rahul Purandare. Residual dynamic typestate analysis exploiting static analysis: results to reformulate and reduce the cost of dynamic analysis. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 124–133, New York, NY, USA, 2007. ACM.

[38] Matthew B. Dwyer, Rahul Purandare, and Suzette Person. Runtime verification in context: can optimizing error detection improve fault diagnosis? In *Proceedings of the First international conference on Runtime verification*, RV'10, pages 36–50, Berlin, Heidelberg, 2010. Springer-Verlag.

[39] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen Mccamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. In *Science of Computer Programming*, pages 35–45, 2006.

[40] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. In *Proceedings of the 2006 international symposium on Software testing and analysis*, ISSTA '06, pages 133–144, New York, NY, USA, 2006. ACM.

[41] Cormac Flanagan. Hybrid type checking. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 245–256, New York, NY, USA, 2006. ACM.

[42] Mark Gabel and Zhendong Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 339–349, New York, NY, USA, 2008. ACM.

[43] Mark Gabel and Zhendong Su. Symbolic mining of temporal specifications. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 51–60, New York, NY, USA, 2008. ACM.

[44] http://gcc.gnu.org/.

[45] Simon F. Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 385–402, New York, NY, USA, 2005. ACM.

[46] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 291–301, New York, NY, USA, 2002. ACM.

[47] Klaus Havelund and Grigore Roşu. An overview of the runtime verification tool Java PathExplorer. *Formal Meth. Sys. Design*, 24(2):189–215, 2004.

[48] http://www.hibernate.org.

[49] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[50] Hiroshi Inoue, Hiroshige Hayashizaki, Peng Wu, and Toshio Nakatani. A trace-based Java JIT compiler retrofitted from a method-based compiler. In *Proc. International Symp. Code Generation and Optimization*, 2011. to appear.

[51] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[52] Moonjoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg V. Sokolsky. Java-MaC: A run-time assurance approach for Java programs. *Formal Meth. Sys. Design*, 24(2):129–155, 2004.

[53] Alex Kinneer, Matt Dwyer, and Gregg Rothermel. Sofya: A Flexible Framework for Development of Dynamic Program Analyses for Java Software. Technical Report TR-UNL-CSE-2006-0006, University of Nebraska - Lincoln, April 2006.

[54] Alex Kinneer, Matthew B. Dwyer, and Gregg Rothermel. Sofya: Supporting rapid development of dynamic program analyses for Java. In *Companion to the proceedings of the 29th International Conference on Software Engineering*, ICSE COMPANION '07, pages 51–52, Washington, DC, USA, 2007. IEEE Computer Society.

[55] Leslie Lamport. What good is temporal logic? In *IFIP Congress*, pages 657–668, 1983.

[56] Choonghwan Lee, Feng Chen, and Grigore Roşu. Mining parametric specifications. Technical Report http://hdl.handle.net/2142/16947, University of Illinois, August 2010.

[57] Ondřej Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, Dec 2002.

[58] David Lo, Leonardo Mariani, and Mauro Pezzè. Automatic steering of behavioral model inference. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 345–354, New York, NY, USA, 2009. ACM.

[59] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 501–510, New York, NY, USA, 2008. ACM.

[60] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a Program Query Language. *SIGPLAN Not.*, 40:365–383, October 2005.

[61] Patrick Meredith, Dongyun Jin, Feng Chen, and Grigore Roşu. Efficient monitoring of parametric context-free patterns. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering(ASE '08)*, pages 148–157. IEEE/ACM, 2008.

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

[63] S. P. Midkiff, J. E. Moreira, and M. Snir. Optimizing array reference checking in Java programs. *IBM Syst. J.*, 37(3):409–453, 1998.

[64] Jerome Miecznikowski and Laurie J. Hendren. Decompiling Java bytecode: Problems, traps and pitfalls. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 111–127, London, UK, 2002. Springer-Verlag.

[65] Nomair A. Naeem. Programmer-friendly decompiled Java. Master's thesis, McGill University, Aug 2006.

[66] Nomair A. Naeem and Ondrej Lhotak. Typestate-like analysis of multiple interacting objects. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented*

programming systems languages and applications, OOPSLA '08, pages 347–366, New York, NY, USA, 2008. ACM.

[67] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[68] Rahul Purandare, Matthew B. Dwyer, and Sebastian Elbaum. Monitor optimization via stutter-equivalent loop transformation. In *OOPSLA '10: Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 270–285, New York, NY, USA, 2010. ACM.

[69] http://sofya.unl.edu.

[70] http://www.sable.mcgill.ca/soot/.

[71] R E Strom and S Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.

[72] Raja Vallée-Rai. SOOT: A Java bytecode optimization framework. Master's thesis, School of Computer Science, McGill University, Montreal, Canada., Oct 2000.

[73] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: mining temporal API rules from imperfect traces. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 282–291, New York, NY, USA, 2006. ACM.