Engineering Instrumentation for Runtime Verification and Monitoring

Chukri Soueidi

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : MSTII - Mathématiques, Sciences et technologies de l'information, Informatique
Spécialité : Informatique
Unité de recherche : Centre de recherche Inria de l'Université Grenoble Alpes

Ingénierie de l'instrumentation pour la vérification de l'exécution

Engineering Instrumentation for Runtime Verification and Monitoring

Présentée par :

Chukri SOUEIDI

Direction de thèse :

Gwen SALAUN PROFESSEUR DES UNIVERSITES, UNIVERSITE GRENOBLE ALPES	Directeur de thèse
Yliès FALCONE MAITRE DE CONFERENCES, UNIVERSITE GRENOBLE ALPES	Co-encadrant de thèse

Rapporteurs :

KLAUS HAVELUND
SENIOR SCIENTIST, NASA JET PROPULSION LABORATORY
WALTER BINDER
FULL PROFESSOR, UNIVERSITA DELLA SVIZZERA ITALIANA

Thèse soutenue publiquement le **13 mai 2024**, devant le jury composé de :

SADDEK BENSALEM, PROFESSEUR DES UNIVERSITES, UNIVERSITE GRENOBLE ALPES	Président
GWEN SALAUN, PROFESSEUR DES UNIVERSITES, UNIVERSITE GRENOBLE ALPES	Directeur de thèse
KLAUS HAVELUND, SENIOR SCIENTIST, NASA JET PROPULSION LABORATORY	Rapporteur
WALTER BINDER, FULL PROFESSOR, UNIVERSITA DELLA SVIZZERA ITALIANA	Rapporteur
SYLVAIN HALLE, FULL PROFESSOR, UNIVERSITE DU QUEBEC A CHICOUTIMI	Examinateur
JULIEN SIGNOLES, INGENIEUR DE RECHERCHE, CEA CENTRE DE PARIS-SACLAY	Examinateur

Invités :

YLIES FALCONE
MAITRE DE CONFERENCES, UNIVERSITE GRENOBLE ALPES

ABSTRACT

Runtime Verification is an essential dynamic verification technique that enables formal reasoning on program executions based on specified properties. Runtime verification can be used in various stages of application development or online in production environments. Instrumentation plays a critical role in ensuring that monitors receive accurate traces that abstract needed information from the executing program. Otherwise, the integrity of the monitoring process is compromised, rendering the results unreliable.

In this thesis, our primary focus is on online monitoring and concurrent programs. We identify three key challenges in applying runtime verification to these areas: capturing the correct traces, effectively guiding the instrumentation process, and assessing the validity of the collected traces. These challenges often render existing monitoring frameworks inadequate in various scenarios. To address the first two challenges mentioned above, this thesis introduces BISM, a bytecode-level instrumentation framework designed for JVM languages. BISM provides high-level abstractions suitable for different types of users within the domain of runtime verification and fulfills the following expressiveness capabilities.

First, BISM allows the extraction of events at the bytecode level accommodating the monitoring of properties specified over events with various granularity levels. Second, BISM is designed to support writing static analyzers within the instrumentation specification. This enables guiding the instrumentation process to consider property and program semantics in order to optimize instrumentation. Third, BISM provides advanced users with the flexibility of unrestricted code modification. This feature is essential for deploying inline monitors or enforcing certain properties such as a sequential order of concurrent events.

Building on the capabilities of the instrumentation framework, we introduce a novel method for residual runtime verification of parametric properties. This approach minimizes instrumentation points by integrating both property semantics and program behavior and applies to a broad range of safety and co-safety properties. Notably, this method is designed to be independent of external static analysis frameworks and can be performed fully within the instrumentation process, allowing for modular integration into various runtime verification workflows.

For concurrent programs, we focus on the online monitoring of general behavioral properties. We highlight the shortcomings of using linear traces for capturing the behavior of concurrent programs and propose that a trace must satisfy two critical properties—soundness and faithfulness—to provide an accurate representation. To meet these criteria, we introduce a trace-collection methodology that employs a real-time vector clock algorithm to establish event causality in a non-intrusive manner. This algorithm can run off the program's critical path, minimizing the impact on the execution. Additionally, we present new criteria for assessing the validity of collected traces in concurrent programs, addressing the third challenge mentioned above. We redefine the concept of trace monitorability based on automata-based formalisms. Our refined definition integrates a causal dependence relation extracted from a given property to identify non-permutable events within a trace. This allows us to evaluate whether a trace contains sufficient order information to yield a sound monitoring verdict. We also develop and implement an opportunistic monitoring framework that leverages existing synchronization points to synchronize thread local monitors with a global monitor. This framework allows for the monitoring of global properties without the need for additional synchronization mechanisms that could disrupt program execution.

Our contributions are evaluated through extensive experiments on various benchmarks, and real-world and synthetic applications from the literature. We also utilize BISM in the broader context of dynamic analysis, particularly in log analysis, testing coverage, and profiling. Collectively, these contributions enhance the reliability and efficiency of runtime monitoring for concurrent programs. We substantiate our theoretical claims with implemented solutions, providing practical tools for the advancement of the field.

Keywords: *runtime verification, monitoring, instrumentation, concurrent programs.*

Ingénierie de l'instrumentation pour la vérification de l'exécution

RÉSUMÉ

La vérification à l'exécution est une technique de vérification dynamique cruciale pour le raisonnement formel sur les exécutions de programmes, applicable tant au développement d'applications qu'en ligne dans des environnements de production. L'instrumentation joue un rôle crucial pour garantir que les moniteurs reçoivent des traces précises qui abstraient les informations nécessaires du programme en exécution. Sinon, l'intégrité du processus de surveillance est compromise, rendant les résultats peu fiables.

Cette thèse se focalise sur la surveillance à l'exécution et les programmes concurrents, en abordant trois défis principaux : la capture des traces correctes, le guidage efficace de l'instrumentation, et l'évaluation de la validité des traces collectées. Pour répondre à ces défis, nous introduisons BISM, un cadre d'instrumentation au niveau du bytecode pour les langages JVM, offrant des abstractions de haut niveau pour divers utilisateurs dans le domaine de la vérification à l'exécution.

BISM permet l'extraction d'événements au niveau du bytecode, accommodant la surveillance de propriétés spécifiées avec divers niveaux de granularité. BISM permet également d'écrire des analyses statiques en spécifiant l'instrumentation. Il offre aux utilisateurs avancés la possibilité de modifier de manière non restreinte et flexible le code existant. Ces dernières caractéristiques sont essentielles pour déployer des moniteurs en ligne ou imposer certaines propriétés comme un ordre séquentiel d'événements concurrents.

Nous introduisons aussi une méthode nouvelle pour la vérification à l'exécution résiduelle de propriétés paramétriques, réduisant les points d'instrumentation et applicable à des propriétés de sécurité. Pour les programmes concurrents, nous mettons en évidence les limites de l'utilisation de traces linéaires et introduisons une méthodologie de collecte de traces utilisant un algorithme d'horloge vectorielle en temps réel, minimisant l'impact sur l'exécution. Cette méthode intègre une relation de dépendance causale, permettant d'évaluer la validité des traces dans les programmes concurrents pour un verdict de surveillance fiable.

Nos contributions sont évaluées à travers des expériences sur divers benchmarks et applications, utilisant BISM dans un contexte plus large d'analyse dynamique, y compris l'analyse de logs, la couverture des tests, et le profilage. Ces travaux améliorent la fiabilité et l'efficacité de cette surveillance pour les programmes concurrents, fournissant des outils pratiques pour l'avancement du domaine.

Mots-clés: *vérification à l'exécution, instrumentation, programmes concurrents.*

Acknowledgements

Completing this thesis has been an incredibly rewarding and exciting journey. I owe a great deal of gratitude to several key individuals who have supported and guided me along the way.

First and foremost, I extend my deepest gratitude to my supervisor, Yliès Falcone, whose profound knowledge and commitment to creating well-founded rigorous methods and tools have significantly impacted my work. Throughout this journey, Yliès was consistently available for consultation, providing the encouragement and direction needed to improve my research. His ability to see the bigger picture, coupled with his friendly and humane approach, made the challenges of research much more manageable and enjoyable. I am forever grateful for his support and guidance.

I am grateful for the opportunity to work with Sylvain Hallé, whose collaborations introduced new perspectives to my work and greatly boosted my enthusiasm. My visits to the beautiful Chicoutimi and Saguenay region, where Sylvain is based, were particularly inspiring and enriching.

Additionally, Gwen Salaün, my co-supervisor, offered essential support when it was needed. I would also like to express my appreciation to Julien Signoles for his external follow-up and valuable insights.

Thanks to the esteemed members of my jury for their expert evaluation and valuable critiques. I am deeply thankful to Klaus Havelund and Walter Binder for their thorough review and insightful feedback on my manuscript. Their expert attention to detail and constructive suggestions were crucial in enhancing the quality of my thesis. Special thanks to Sadek Bensalem for presiding over the jury.

I would also like to acknowledge Paul Attie, my master's thesis supervisor from the American University of Beirut, whose mentorship ignited my research interest and laid the foundation for my academic pursuits.

My experience was further enriched by the camaraderie and discussions with the entire CORSE team at Inria, who graciously hosted me during my research. Particular thanks to Christophe Guillon, whose discussions were invaluable. I am also grateful to Imma Presseguer for her administrative and friendly support, which helped make my research experience smoother and more enjoyable.

Living and working in Grenoble, surrounded by the majestic mountains, provided inspiration and tranquility, significantly contributing to my overall well-being and productivity during this journey. A special thanks to Antoine El Hokayem, who provided useful feedback on my work on various occasions.

The unwavering support from my family has been a cornerstone throughout this journey. I am deeply appreciative of my parents, whose faith in me has been a constant source of encouragement. Additionally, I would like to thank my brother and sisters for their love and support.

Finally, I extend my deepest gratitude to my partner Lena. Her strength and love have been my essential support system throughout this journey, and her encouragement has been invaluable.

Contents

1 Introduction **1**
- 1.1 An Overview of Computer-Aided Verification 2
  - 1.1.1 Static Approaches 2
  - 1.1.2 Dynamic Approaches 3
  - 1.1.3 Combining Static and Dynamic Approaches 4
- 1.2 Runtime Verification 4
- 1.3 Identified Challenges 6
  - 1.3.1 Capturing Correct Traces (C1) 7
  - 1.3.2 Guiding the Instrumentation Process (C2) 7
  - 1.3.3 Trace Validity Assessment for Concurrent Programs (C3) 7
- 1.4 Detailed Problem Statement 8
  - 1.4.1 Instrumentation Frameworks 8
  - 1.4.2 Monitoring Concurrent Programs 9
- 1.5 Summary of Contributions 9
- 1.6 Structure of the Thesis 10
- 1.7 Associated Publications 11

2 Preliminaries **13**
- 2.1 Programs, Methods and the CFG 13
- 2.2 Event Traces, Properties, and Monitoring 15
- 2.3 Parametric Monitoring 16
- 2.4 Upward Closure 17
- 2.5 Relations, Partial and Total Orders 18
- 2.6 Concurrent Executions 18
  - 2.6.1 Actions 18
  - 2.6.2 Execution Order 20
  - 2.6.3 Concurrent Execution 21
- 2.7 Vector Clocks 21

I Program Instrumentation **23**

3 Program Instrumentation and Existing Frameworks **25**
- 3.1 Introduction 27
- 3.2 Understanding Instrumentation 27
  - 3.2.1 Unveiling the Complete Picture 27
  - 3.2.2 Observing the Execution 28

3.3 Instrumentation for Runtime Verification 29
3.3.1 The Program 30
3.3.2 The Observation 30
3.3.3 The Analysis 32
3.3.4 The Instrumentation Language 32
3.4 Instrumentation Requirements 33
3.5 Evaluating Instrumentation 34
3.6 Existing Instrumentation Frameworks 34
3.6.1 Bytecode Manipulation Libraries 34
3.6.2 Aspect-Oriented Approaches 36
3.7 The Need for a Comprehensive Instrumentation Framework 40
3.8 Conclusion 41

4 A Comprehensive Instrumentation Model 43
4.1 Introduction 45
4.2 Instrumentation Model 45
4.2.1 Context Objects 45
4.2.2 Join points 46
4.2.3 Advice 46
4.2.4 Shadows 46
4.2.5 Selectors 49
4.2.6 Instruction Visibility 50
4.2.7 Transformers 50
4.2.8 Instrumentation Process 50
4.3 Transformer Composition 50
4.3.1 Motivations for Composition 50
4.3.2 Composition of Transformers 51
4.3.3 Transformer Collision 51
4.3.4 Order Matters 51
4.4 Conclusion 52

5 BISM: Bytecode Instrumentation for Software Monitoring 53
5.1 Introduction 55
5.2 BISM in a Nutshell 55
5.2.1 Overview 55
5.2.2 Design Goals and Features 56
5.3 BISM Instrumentation Language 57
5.3.1 Selectors 57
5.3.2 Static Context 59
5.3.3 Dynamic Contexts 61
5.3.4 Advice Methods 61
5.3.5 Instrumentation Scoping 62
5.3.6 User Configuration 62
5.3.7 Transformer Composition 63
5.4 The External DSL for BISM 64
5.4.1 Design Considerations 64
5.4.2 Pointcuts 64
5.4.3 Events 65
5.4.4 Monitors 65
5.4.5 Code Generation 66
5.5 Implementation 66
5.5.1 The DSL 68
5.6 An Observation Layer for BISM 69
5.7 Discussion 69
5.8 Conclusion 71

II Guiding Instrumentation with Residual Analysis **73**

6 Residual Runtime Verification of Parametric Properties **75**
6.1 Introduction 77
6.2 Residual Analysis of Parametric Properties 77
6.3 Residual Analysis via Intraprocedural Reachability Analysis 79
6.3.1 Motivating with an Example 79
6.3.2 Capturing a Program Model 80
6.3.3 Extending the Automaton of Bad Prefixes 82
6.3.4 Cutting the Behavior 83
6.3.5 Scope and Soundness of the Analysis 84
6.4 Implementation 85
6.5 Related Approaches 86
6.6 Conclusion 86

III Monitoring Concurrent Programs **87**

7 Representative Traces for Concurrent Programs **89**
7.1 Introduction 91
7.2 Trace Collection for Concurrent Programs 92
7.2.1 Issues with Linear Traces 93
7.3 Concurrent Traces 94
7.4 Sound and Faithful Concurrent Traces 94
7.5 Obtaining Sound Concurrent Traces 95
7.5.1 Atomicity and Instrumentation Requirements 95
7.5.2 The Reordering Algorithm 96
7.5.3 Algorithm Cost 96
7.5.4 Algorithm Correctness 96
7.6 Criteria For Monitorability 98
7.6.1 Monitor Causal Dependence 98
7.6.2 Trace Monitorability of Concurrent Executions 100
7.6.3 Optimal Faithfulness 100
7.7 Implementation 101
7.8 Related Approaches 101
7.9 Conclusion 102

8 Opportunistic Monitoring **105**
8.1 Introduction 107
8.2 Opportunistic Runtime Verification 108
8.3 Implementation 110
8.4 Conclusion 111

IV Evaluation and Use Cases **113**

9 Evaluation **115**
9.1 Introduction 115
9.2 Evaluating BISM 116
9.2.1 Methodology 116
9.2.2 Advanced Encryption Standard (AES) 117
9.2.3 Financial Transaction System 118
9.2.4 DaCapo Benchmarks 121
9.2.5 Threats to Validity 122
9.3 Evaluating the BISM DSL 122
9.3.1 Performance Evaluation 123
9.3.2 User Experience Evaluation 123

9.4 Evaluating the Residual Analysis 124
9.5 Evaluating Concurrent Traces 125
9.5.1 Effectiveness and Cost 125
9.5.2 Causal Dependence Relation in Specification Patterns 128
9.6 Evaluating the Opportunistic Monitoring 129
9.6.1 Readers-Writers 129
9.6.2 Other Benchmarks 130
9.7 Conclusion 132

10 UseCases 133
10.1 Introduction 133
10.2 Law of Demeter Checker 134
10.3 Code Analysis of Programs 134
10.3.1 Mc Cabe Complexity. 136
10.3.2 ABC Complexity. 136
10.3.3 Unused Variables. 137
10.4 Obfuscation 137
10.4.1 Renaming Obfuscator 138
10.4.2 Junk Code Obfuscator 138
10.5 Mutation of Programs 139
10.5.1 Return Mutator: *Value Mutation* 139
10.5.2 Instruction Mutator: *Operator Mutation* 139
10.5.3 Void Call Mutator: *Statement Mutation* 140
10.6 Runtime Verification and Enforcement 141
10.6.1 Good Java Practices: HasNext Property 141
10.6.2 Concurrent Executions: Forcing Advice Atomicity 141
10.6.3 Test Inversion Attack Detection and Enforcement 141
10.7 Logging 143
10.8 Dynamic Profiling 143
10.8.1 Call Graph 143
10.8.2 Object Allocation 143
10.9 Dynamic Analysis with Complex Event Processing 145
10.10Conclusion 145

11 Dynamic Program Analysis with BISM and Complex Event Processing 147
11.1 Introduction 149
11.2 Dynamic Program Analysis 150
11.2.1 Existing Approaches 150
11.2.2 Limitations 151
11.3 BeepBeep Overview 152
11.4 The BISM-BeepBeep Integration 153
11.4.1 Implementation 153
11.4.2 Runtime Verification: Monitoring and Synthesis 153
11.4.3 Profiling: The Dynamic Call Graph 155
11.4.4 Log Analysis: Complex Instrumented Events 155
11.4.5 Coverage: Versatile Metrics 157
11.5 Experimental Evaluation 159
11.5.1 Monitoring 159
11.5.2 Coverage 160
11.5.3 Profiling 160
11.6 Conclusion 161

V Conclusion and Perspectives 163

12 Conclusion and Perspectives 165
12.1 Contributions 165

12.1.1 Program Instrumentation 165
12.1.2 Guiding the Instrumentation Process with Residual Analysis 166
12.1.3 Monitoring of Concurrent Programs 166
12.1.4 Contributions to the Broader Scope of Dynamic Program Analysis 167
12.2 Perspectives 167
12.2.1 BISM 167
12.2.2 Combinining Static and Dynamic Analysis 168
12.2.3 Monitoring of Concurrent Programs 169

Bibliography **186**

VI Appendix **187**

A Other Works **189**
A.1 Leveraging Runtime Verification for the Monitoring of Digital Twins 189
A.2 Monitoring Business Process Compliance Across Multiple Executions with Stream Processing 189

Lists **191**
A.3 List of Definitions, Propositions, Theorems, Corollaries, Lemmas, and Examples 191
List of Definitions, Propositions, Theorems, Corollaries, Lemmas, and Examples 191
List of Figures 194
List of Tables 195

CHAPTER 1

Introduction

COMPUTERS are now integral to our lives, serving as the backbone for a diverse range of infrastructure from transportation systems like trains and planes, to communication networks, healthcare, financial services, energy systems, and military operations. As software systems grow in scale and functionality, assuming increasingly critical roles, their reliability has never been more important. Software failures can have devastating consequences on human life, the environment, and the economy, as demonstrated by several disasters in recent history. For example, the recent Boeing 737 MAX crashes, which claimed nearly 350 lives, were attributed to flawed flight control software and highlighted deficiencies in software verification and validation [HBM20]. The fatal radiation overdoses from the Therac-25 medical machine underscored the critical role of software quality in medical devices [LT93]. The 2010 Flash Crash revealed systemic vulnerabilities introduced by automated trading systems [KKST17]. Widespread security vulnerabilities like Heartbleed further expose the risk of relying on insecure software libraries [DLK+14].

The software that powers our modern world, although structured and seemingly intuitive to users, stands on a deceptively complex foundation of components. Even the most basic programs we use in our daily lives rely on an intricate structure of hardware, operating systems, and programming languages. Each of these programs is a set of instructions designed to instruct a computer to perform specific actions. During the creation of these instruction sequences, developers might introduce software *faults* due to the usage of incorrect commands, wrong sequencing of operations, or an inaccurate representation of user requirements. These faults can manifest as software *errors* during the program's execution which indicates that the program has entered into some wrong state. In turn, such errors can lead to software *failures* which are deviations from the expected behavior of the software. A common word that is often used to refer to these faults, errors, and failures is *bugs*.

Identifying and preventing bugs in software systems remains a non-trivial task. In his 1972 Turing Award Lecture [Dij72], Dijkstra envisaged a future where each program would be validated by a mathematical proof of its correctness. Despite early promising works such as [McC59, Hoa69] that have advanced formal proof systems for program correctness, these techniques have found limited real-world application. The primary reason is scalability; due to the complex nature of real-world software systems which often defies easy formalization [CHV+18]. The complexity of modern-day software emerges from several factors. Firstly, the deliberately built layers of abstractions in programming languages, while aiding in development by allowing programmers to focus on high-level tasks, also may conceal the system's full behavior. Secondly, software modularity creates complexities due to interactions among different modules, while external dependencies like third-party libraries further complicate verification efforts. Another significant challenge is concurrency, where asynchronous operations may lead to non-deterministic behaviors. Additionally, the dynamic nature of software, affected by updates, user input, hardware interactions, and environmental factors, add to these challenges.

As such, one of the very active fields in software engineering is software *verification* [IEE17] which aims to ensure that software systems meet their defined specifications. Verification approaches are employed at various stages of the software development life cycle to systematically identify and prevent bugs. While a complete *a priori* verification is ideally desired for a program, there is no silver bullet when it comes to verifying a program statically

i.e., before executing it. As such *dynamic verification* approaches, which analyze the program during or after its execution, are often employed to complement static verification techniques. In fact, in critical systems, the combination of static and dynamic analysis has long become the norm and often even mandatory [BS93].

This work aims to contribute to enhancing the dependability of software systems by developing more rigorous dynamic verification methods. Specifically, the work will address the limitations when capturing and abstracting the system behavior for dynamic verification techniques, and the challenges of efficiently monitoring software systems post-deployment.

1.1 An Overview of Computer-Aided Verification

Verification and validation are essential activities in software engineering, encompassing a broad range of tasks such as code reviews, quality audits, performance monitoring, and simulations [IEE17]. While verification confirms that the system meets specific requirements, validation ensures those specifications align with user needs. Although often used interchangeably, verification is a necessary but not sufficient condition for validation. We here focus on verification and more particularly on computer-aided approaches.

The central question that verification aims to answer is straightforward yet intricate: *Does the system under study operate according to specifications?* Specifications are set by the user and define the expected *properties* of the structure or behavior of a system and can be expressed in various ways, including formal specifications, requirements, or constraints [DAC99, FCT10]. To add rigor to the verification process, formal methods are often employed. These methods are often based on modeling and logic, and implemented in a structured framework for specification, design, and verification. A cornerstone of formal methods is the notion of mathematical proof to establish system correctness. This typically involves two key efforts: a modeling task and an algorithmic task. The modeling task involves identifying appropriate models that encapsulate both the system's behavior and the properties being verified. Conversely, the algorithmic task is focused on developing an algorithm capable of verifying these properties within the chosen model.

Before discussing further aspects, it is crucial to acknowledge the limitations of verification. Early results in the theory of computation, such as Turing's Halting Problem [Tur36] and Rice's Theorem [Ric53], show that complete algorithmic verification is an unsolvable problem in a general sense. This means that there are inherent limitations to what can be algorithmically determined about programs in general and that many non-trivial semantic properties of programs are undecidable. However, these theorems deal with general cases and that is what makes the field of software verification challenging and interesting.

Verification techniques can be characterized by several factors, including but not limited to whether they produce sound and complete results, the level of automation, the rigor involved, the syntactic and semantic knowledge of the program they require, and the type of properties they can verify. Soundness and completeness are two important properties of verification techniques. A technique is said to be *sound* if it does not produce false positives, i.e., it does not report a property as violated when it is not. A technique is said to be *complete* if it does not produce false negatives, i.e., it does not report a property as satisfied when it is not. Moreover, techniques are generally categorized into *static* and *dynamic* approaches. In the following subsections, we will informally present some of these techniques.

This thesis focuses on runtime verification, a dynamic verification technique that we present in Section 1.2. In Section 1.3, we then discuss the challenges we identified in runtime verification. In Section 1.4, we detail the problem statement of this thesis. We then present a summary of our contributions in Section 1.5, the thesis structure in Section 1.6, and the publications that resulted from this thesis in Section 1.7.

1.1.1 Static Approaches

Static verification operates without executing the program and encompasses methods such as model checking, deductive verification, and static analysis.

Model Checking. Model Checking [CE81, QS82, BCMD91] is used to verify whether a model of the system satisfies a given property. The model is typically a transition system that represents different states of the system's behavior and the property is usually expressed by the user in a temporal logic. A model-checking algorithm exhaustively explores the state space of the model to verify whether the property holds. One example of model

checking is verifying the correctness of a network protocol design. One might formally define the states and transitions of the protocol and use a model checker to verify that deadlock situations cannot occur. A main challenge in model checking is the state explosion problem, where the number of states in the model that need to be explored grows exponentially. For instance, in reactive systems where the system maintains an ongoing interaction with its environment, the number of states can be infinite. Another challenge is the mismatch between existing models (that represent programs and properties) and real-world systems. Existing models are often not effective in specifying constructs of real-world systems. For example, a model may not be able to capture the behavior of a system that relies on external libraries.

Deductive Verification. Deductive verification [Hoa69, Dij75] aims to formally prove that a program satisfies its specification by using logical reasoning. It involves designing and attaching predicates to various nodes within a program's control flow graph. These predicates are typically checked using theorem provers to verify that they hold whenever the control flow reaches these specific nodes. In this framework, a *contract* is often expressed as preconditions and postconditions surrounding code fragments. Then a set of inference rules is used to prove the partial correctness of this contract and thus the program. However, unlike some other verification techniques, deductive verification often requires manual effort, as human input is often needed to specify conditions and invariants.

Static Analysis. Static analysis [AC76, Hec77, CC77, NNH15] encompasses a wide range of approaches to automatically verify the properties of a program without executing it. The static analysis aims to statically predict the properties of the dynamic behavior of the program relying on conservative overapproximations, which often lead to false positives. Static analysis can be used in program optimization, development, and verification. One of the most common forms of static analysis is type checking, which is used to detect type errors such as assigning an integer to a string variable. Another common form of static analysis is data-flow analysis, which can be used to identify variables that are never used or to detect potential null pointer dereferences. Static analysis relies on approximations of the program behavior which may also yield false positives.

While static techniques provide a robust analysis of a system's behavior given the limitations we discussed, their scope is limited to various unpredictable factors. Soft errors such as bit flips in memory, hardware failures, and incorrect assumptions about the execution environment can introduce unforeseen errors that escape static analysis.

1.1.2 Dynamic Approaches

Dynamic verification techniques analyze the program during or after its execution. These techniques do not necessarily require access to the source code and can treat the program as a black box analyzing its execution traces. With access to the dynamic information generated by the program, dynamic techniques often produce fewer false positives than static techniques which rely on approximations or inaccurate models. However, they are inherently incomplete as they examine single execution paths.

We present some of these techniques informally below. We leave a more detailed discussion of runtime verification in the next section.

Testing. Testing is the most adopted method to detect errors in programs. However, the only errors it detects are the ones that are manifested during the executions of the given test suites. Testing is achieved by executing the program with a set of inputs where then a test oracle compares the actual output with the expected output. Testing approaches can be manual [IML09], model-based [JVCS07], with various automated testing tools [GKS05, Run06, Bec04]. Other approaches focus on generating test cases automatically [GA14, RC17, McM04]. Calculating various coverage metrics is often associated with evaluating the quality of a test suite utilizing tools such as [JaC, jco]. *Query-based testing* [HSTV09], is a generalization of these metrics. For example, a query may impose that a specific line be visited, then that a variable be assigned a specific value, etc. One of the main challenges in testing is the difficulty of generating a comprehensive set of test cases that cover all possible execution paths. Another challenge is the difficulty of generating test cases that expose errors. For example, a test case that does not cover a particular execution path may not expose an error that occurs on that path.

Assertion Checking. Assertion checking [CR06, SKV17, AGVY11, RLL+13] involves inserting assertions into the code that are checked during execution. Unlike other approaches, which often run separately and observe the

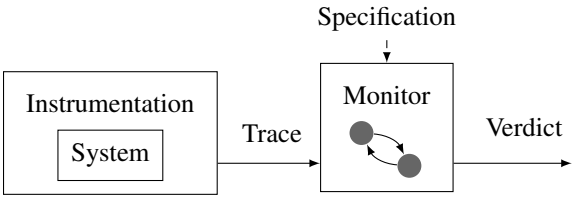


Figure 1.1: The typical setup of Runtime Verification.

running system, assertion checking is embedded directly into the code. For instance, assertions could be used to ensure that an array index is within bounds. Hybrid approaches combining static and dynamic techniques can also be used to minimize the assertions needed to be checked online [DKS13].

1.1.3 Combining Static and Dynamic Approaches

Given the limitations and strengths of both static and dynamic verification methods, combining static and dynamic techniques is natural and widely adopted [EMN12]. Often, static methods such as deductive verification and static analysis are employed in the development process to provide guarantees as early as possible. Complementing this, dynamic methods like runtime verification and assertion checking verify the system during its actual operation. Combining static and dynamic methods provides a comprehensive approach to software verification, effectively addressing a broader set of challenges and mitigating the individual shortcomings of each technique.

Below are some methodical examples of combining static and dynamic verification techniques.

- **Concolic Testing [GKS05, Sen07]**: This approach combines static symbolic execution with dynamic concrete execution to achieve broader code coverage and find tricky bugs. For instance, a concolic tester might use symbolic execution to identify test cases that lead to a particular code path and then run those test cases to find bugs. One example is to achieve high code coverage and identify edge cases that may lead to failures.
- **Residual Analysis [DP07, BLH12]**: This method aims to reduce the overhead of runtime monitoring by using static analysis to prune parts of the program that can be statically verified. This is done by analyzing the property and the program to identify parts of the program or the property that are irrelevant to the verification process.
- **Bounded Model Checking [BCC+09]**: This technique involves exploring the state space of a program by checking the truth of a property over a finite initial segment of its execution paths. By using a constraint solver, it verifies if a counterexample exists within the given bounds, making it feasible to explore more states within a limited amount of time.

1.2 Runtime Verification

Runtime Verification (RV), also known as runtime monitoring [FHR13a, BFFR18a, FKRT21], is a lightweight formal verification technique that focuses on analyzing a system during (or after) its execution to verify whether it satisfies or violates a property. The property needs to be *monitorable* [FFM12, PH18] i.e., its truth or falsehood can be determined by observing a finite prefix of the system's execution, a concept we will elaborate on later in this chapter.

RV focuses on individual program executions rather than exploring all possible scenarios (as in model checking) and serves as a dynamic alternative for scenarios that are too complex for static verification methods. It can be employed either during the testing phase as a formalized test oracle or during system deployment as part of a fault protection strategy, where corrective actions may be taken if the specification is violated. RV has been used in combination with formal static verification methods such as model checking [Leu12], deductive verification [CAPS15] and static analysis [DP07, BLH10, ACP20], as well as informal dynamic methods such as testing [CAS18] and debugging [JFMP17]. Runtime verification has been applied in a variety of domains and industries such as software engineering [LHX+16], aerospace [PWNG13, ZAD+23], automotive [BFB12], railroads [CZS+23], smarthomes [EHF22], gaming [VLGH17a], healthcare [JSW+16], and finance [CWG+17].

Figure 1.2: An overview of the taxonomy of Runtime Verification (from [FKRT21]).

Figure 1.1 illustrates the typical RV setup. The inputs to a runtime verification system are the system under study and a specification that expresses a property of interest in the desired behavior of the system. Before executing the program, the system is instrumented so that it produces a trace of events that abstracts its execution. Also, the specification is used to synthesize a monitor, which is responsible for analyzing the execution trace and determining whether the property is satisfied or violated. When the system is executed, the monitor consumes the trace of events and outputs a verdict indicating whether the execution satisfies or violates the specification.

We now present a high-level overview of the major concepts involved in runtime verification. In [FKRT21], the authors organized the various aspects of runtime verification into a taxonomy, we here present a summary of this taxonomy. Figure 1.2 illustrates the taxonomy of runtime verification.

Trace. The predominant approach to modeling program behavior in runtime verification involves observing an execution and abstracting it into a *trace of events*. Capturing all the information produced by the execution is infeasible, so the trace is typically a sequence of observations that abstracts the behavior of interest in the program for the monitor. Traces assume dual roles: firstly, they encapsulate the information extracted from the execution of the program; secondly, they function as mathematical models for formal reasoning about the properties of the RV technique. For instance, properties expressed using LTL formulae are usually interpreted over infinite traces of sets of atomic propositions. Additionally, the collection of a trace often necessitates the association of information with temporal markers. These could be timestamps, logical clocks, or other temporal abstractions.

Specification. A specification is a description of a property written with a well-defined formalism. It can be explicitly defined using formal languages like temporal logic, regular expressions, or state machines. Other properties can be implicitly and directly integrated within the monitor such as generic concurrency properties like data-race and deadlock freedom. Moreover, some techniques focus on learning specifications from execution traces. The choice of language for explicit specification depends on the specific verification requirements. For example, temporal logic is suitable for expressing properties that reason about the ordering and timing of temporally distinct

program states or events, whereas regular expressions are more suitable for expressing patterns related to the occurrence of events within a sequence. Other specification languages include finite state machines[RCR15a, CP17], stream equations [LSS+18, GS21], and first-order logic. The formalization of properties in terms of specifications requires human effort and can be error-prone leading to resistance from practitioners.

Monitor. From each property, a monitor is synthesized which is the decision procedure for the property. A monitor is spawned at runtime; it receives events of the trace and is responsible for analyzing the execution of the system and determining whether the property is satisfied or violated. The synthesis process is usually automated and performed offline. Monitors are typically implemented as state machines or formula rewriting [HR01] systems. Typically online monitors incrementally consume events in a step-by-step manner on each program observation yielding a verdict when the property is satisfied or violated. The verdict indicates whether the execution meets the specification or not. Offline monitors consume the entire trace at once and produce a verdict at the end. Offline monitors are typically used for post-mortem analysis. The verdicts usually correspond to a set of truth values. A standard set of truth values is $\{\top, \bot, ?\}$, where $\top$ and $\bot$ respectively signify adherence and non-adherence to the specifications, and the verdict ? implies that a definitive conclusion has not been reached yet. Monitors are often monotonic in the sense that they do not change their verdict when more events are observed. The monitor should also typically produce a verdict as early as possible. Offline monitors consume the entire trace at once and produce a verdict at the end. Offline monitors are typically used for post-mortem analysis.

Instrumentation. Extracting traces typically relies on *instrumentation*, a technique that entails transforming the base program to emit events that abstract its behavior. Instrumentation consists of two main steps: 1) identifying the program points corresponding to the events of interest, and 2) inserting additional code into the base program to extract information. A property is typically formalized using an alphabet of abstract events, denoted as Σ_a. The program generates concrete events, denoted as Σ_c, which must be mapped into a trace of abstract events compatible with the runtime analysis. Instrumentation serves as the mechanism for capturing these concrete events and translating them into their corresponding abstract events, thereby constructing a trace that is compatible with the monitor. Instrumentation is particularly suitable for runtime verification. It provides flexibility in capturing concrete events by pinpointing arbitrary locations in the source code, as opposed to being limited to specific events provided by the execution environment.

Deployment. The deployment of monitors in a system is a critical aspect that necessitates well planning to determine how they are integrated within the program and its execution. In *online* monitoring, verification is conducted concurrently with system execution. Conversely, *offline* monitoring involves post-execution analysis of the system's trace. The monitor can be placed to execute *inline* sharing the same execution environment and address space as the system under observation, or they can be placed *outline* in a separate execution environment and address space. In [CFA+17], the authors present a survey of the various deployment strategies for runtime verification.

Interference and Reaction. The introduction of monitors into a system often leads to interference, affecting aspects like memory or thread scheduling. This is particularly critical in real-time systems where additional delays may be incurred. In terms of reaction, runtime verification techniques can be either passive or active. Passive techniques solely observe the system and report or fail upon specification violations. Active techniques, such as runtime enforcement, go beyond observation and directly intervene in the system to ensure compliance with the specification. These active monitors are capable of actions like snapshotting, rolling back, and suppressing events, and may even allow temporary deviations from the specification before restoring compliance or failing.

1.3 Identified Challenges

Runtime verification presents a multitude of challenges, particularly in the context of online monitoring and concurrent programs. In this thesis, we focus on three challenges: 1) the capturing of correct traces, 2) guiding the instrumentation process, and 3) the assessment of the validity of the collected traces. These challenges often render existing monitoring frameworks inadequate in various scenarios. This limitation is counterproductive considering that these frameworks have been refined to support diverse specification languages and to synthesize efficient monitors. Each of these challenges will be discussed in detail in the following subsections.

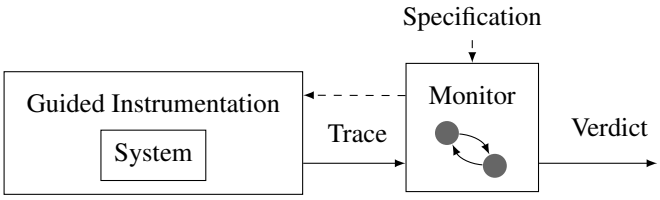


Figure 1.3: Guided instrumentation in RV setup.

1.3.1 Capturing Correct Traces (C1)

Runtime verification approaches often implicitly assume that the acquired traces correctly represent the program. However, this assumption is not always valid. Acquiring accurate and complete traces is crucial for runtime verification, as they serve as the foundation for subsequent monitoring and analysis. If the traces are inaccurate or incomplete, the integrity of the monitoring process is compromised, rendering the results unreliable. This underscores the critical importance of reliable methods for trace collection. Several factors, such as the completeness of captured events, their sequential order, granularity level, and the concurrency model of the program under examination, affect trace quality.

Instrumentation is pivotal in this context, tasked with ensuring that monitors receive accurate and relevant information specific to the monitoring scenario at hand. In concurrent programs, acquiring *sound* traces, i.e. traces where the order of events is correct, necessitates specialized instrumentation techniques. For instance, capturing events in the correct sequential order in concurrent programs might necessitate additional enforced synchronization to ensure the atomic execution of program actions and the corresponding instrumented code which notifies the monitor. Alternatively, reconstructing the causality and establishing the *happens-before* relationship between events requires observing additional synchronization points during the execution. Both scenarios demand the capability to cover and extract detailed low-level information from the program's execution and often result in significant overhead and interference. In some cases, the instrumentation process may require the ability to insert arbitrary bytecode instructions in the program. Current widely adopted instrumentation frameworks, particularly those integrated within monitoring frameworks, lack the adaptability to meet these requirements, thereby highlighting the need for more adaptable solutions.

1.3.2 Guiding the Instrumentation Process (C2)

The instrumentation process commonly takes an alphabet of events as input and is tasked with capturing the corresponding concrete events from the program execution. This process is often designed without considering the semantics of the property or the program being verified, in some cases leading to either excessive or incomplete instrumentation. The former results in unnecessary computational overhead when monitoring online, while the latter may cause false negatives.

In many cases, the property can be considered to guide the instrumentation process (Figure 1.3). This integration would involve a pre-instrumentation analysis phase that extends beyond simple event mapping. It can consider both the property and program semantics to optimize the instrumentation points, improve the quality of the collected traces, and reduce the overhead of instrumentation. Current frameworks for instrumentation generally lack the flexibility to incorporate such weave-time analyses using the proper abstractions. A more adaptable framework could facilitate the integration of different considerations into the instrumentation process such as program semantics, property semantics, and the monitoring approach. This opens the path for the development of more efficient instrumentation techniques that can be tailored to the specific monitoring scenario at hand.

1.3.3 Trace Validity Assessment for Concurrent Programs (C3)

Assessing the validity of a captured trace presents another challenge, particularly in the context of concurrent programs. For example, if two concurrent events execute in parallel, they are incompatible with a monitoring system designed to expect a total order of events. Attempting to linearize the trace of such concurrent events and then passing it to the monitor would lead to unreliable results. In formal terms, such a trace is not *monitorable*, i.e. there exists no monitor that can correctly verify the property using the trace. The trace may contain all the events necessary to monitor a property but may be an inadequate model for the property. Trace monitorability involves a

semantic assessment of the trace against the property under consideration. Inadequate trace models can lead to false positives or false negatives during monitoring, making it crucial to establish robust methods for assessing the monitorability of traces. This is particularly important in the context of concurrent programs, where the notion of trace monitorability is not yet well-defined and the ordering relation of events in the trace is critical for determining the usefulness of a trace. Therefore, equipping monitors with robust methods for assessing the monitorability of traces can open the door for revisiting and extending the plethora of existing monitoring approaches that rely on total order formalism to handle concurrent programs.

1.4 Detailed Problem Statement

The challenges we identified in Section 1.3 are intrinsically tied to the instrumentation and trace collection process. In this section, we present our problem statement from two perspectives: instrumentation frameworks and concurrent programs.

1.4.1 Instrumentation Frameworks

Given the challenges C1 and C2, we find that widely adopted instrumentation frameworks are inadequate. They either lack the expressiveness required for low-level event capture and program modification or do not provide the flexibility to guide the instrumentation process with semantic considerations.

In the landscape of program instrumentation, two categories of frameworks are commonly employed: low-level code manipulation libraries and high-level aspect-oriented programming frameworks. Low-level libraries offer comprehensive capabilities for fine-grained code coverage and transformations; however, their verbosity imposes a steep learning curve. They do not offer the proper level of abstraction for instrumentation. Effective use of these libraries necessitates a deep understanding of program traversal strategies and bytecode syntax and semantics. Conversely, high-level frameworks simplify the instrumentation process by providing more abstract languages for specification. These are widely adopted in existing runtime verification approaches. However, they often lack the expressiveness required to capture intricate details of program execution, such as individual bytecode instructions, stack values, and local variables. Moreover, the ability to modify the base program in these languages is restricted to the language constructs provided by the framework. The dichotomy between these two types of frameworks highlights the need for a more versatile solution that combines the best of both worlds: the expressiveness of low-level libraries and the user-friendliness of high-level frameworks.

As such, monitoring tools relying on aspect-oriented programming frameworks like AspectJ [KHH+01a] are often confined to coarse-grained events. While some monitoring scenarios require only high-level events, others necessitate low-level details. For instance, while typestate property monitoring often focuses on high-level operations such as method calls [KF19, DP07, BLH10], other monitoring scenarios, such as control-flow integrity, require the capture of low-level details [KF19]. Being limited to coarse-grained events restricts the scope of what the monitoring system can observe. This limitation is particularly counterproductive given that these monitoring frameworks are refined to allow for various specification languages and produce efficient monitors. Their coupling with the instrumentation framework, however, limits their expressiveness leaving their advanced capabilities unused.

Additionally, the pointcut/advice model of aspect-oriented programming frameworks does support custom analyses in the instrumentation process. The user is restricted to specifying only the code to be inserted, without allowing for compile-time program analyses that could incorporate syntactic or semantic considerations. For example, in the context of dynamic dispatch analysis, it may be desirable to emit an event only when a method is overloaded by another method in the same class. Such pre-instrumentation analysis is not feasible without resorting to cumbersome compiler customizations, as seen in [ACH+05, AM07, BLH10].

As such various optimization opportunities are missed. For instance, in the context of residual runtime verification [DP07, BLH10, ACP20], static verification can be employed to prune unnecessary instrumentation points, thereby reducing monitoring overhead. This requires analyzing both the semantics of the property under verification and program behavior during instrumentation. The absence of support for semantic inclusion in the instrumentation process limits the optimization possibilities of the monitoring process.

The existing landscape of runtime monitoring lacks a comprehensive instrumentation framework that can address the challenges of instrumentation in the context of various monitoring scenarios. In this thesis, we seek to provide a framework that can combine the simplicity of the instrumentation model from aspect-oriented languages with

the flexibility and control provided by bytecode manipulation libraries. This framework aims to fully utilize the capabilities of existing monitoring methods and provides the foundation for enabling new monitoring approaches.

1.4.2 Monitoring Concurrent Programs

Given the challenges C1, C2 and C3, we find that existing monitoring techniques are not yet fully equipped for the demands of online monitoring of concurrent programming. Concurrent programs introduce unique challenges for dynamic verification. While a concurrent program behaves non-sequentially, the trace collection is sequential. A correct trace must be both *sound*, providing no false information about event order, and *faithful*, containing all necessary ordering information. Such a trace is ideally represented by a partial order over the relevant events.

Establishing this partial order between events is crucial for sound monitoring, yet computationally expensive. Consider the general behavioral precedence property, which specifies that a resource can only be granted (event g) in response to a request (event r), and can be expressed in LTL as $\neg g \ \mathbf{W} \ r$ (not g weak until r). The order of events g and r is important for the satisfaction of this property and the *happens-before* [Lam78] relation between events must be established. While the execution is concurrent, the trace is sequential. This mismatch between the trace and the execution requires careful handling. Concurrent (parallel) events are arbitrarily ordered in a trace when linearizing their execution, which may not correspond to the actual execution leading to false positives or false negatives when monitoring. In that case, if g and r are concurrent events, then a monitor may produce either verdict.

Vector clock algorithms [CL02, MV20, AG05, RS04] have been typically employed in collecting partial orders. However, very few (we know of [RGB20]) are directed towards online monitoring of behavioral properties; where a final trace consists of property-related events only. These algorithms typically require blocking the execution; by synchronizing the instrumentation, program actions, and algorithm's processing to avoid data races [CL02]. This introduces coarse-grained synchronization and interferes with the execution affecting scheduling and performance. With the quadratic bound on their runtime complexity and the coarse-grained synchronization they introduce, one might want to run such an algorithm off the critical path of the program. As such, existing techniques that target behavioral properties and can establish causality, such as [CSR08, JS08, SCR12, HMR14a], are more suitable for the testing phase instead of the deployment phase in production environments. These tools can also utilize the extracted causal model to predict violations. However, most of them assume a sequentially consistent execution model, limiting the space of possible event interleavings and thus their completeness. On the other hand, classical monitoring frameworks, such as Java-MOP [CR05a], Tracematches [AAC+05a, BHL+10], MarQ [RCR15b], and LARVA [CPS09b], that rely on totally ordered traces are not yet equipped for monitoring concurrent programs. Many of these tools were developed to handle single-threaded programs. They linearize the execution of concurrent events, using incorrect instrumentation and ordering assumptions, leading to unsound verdicts [RGB20]. Moreover, these tools rely on AspectJ which is not expressive enough to capture the synchronization points in the program. As such, again they are rendered unusable for monitoring behavioral properties in concurrent programs.

This thesis proposes methods for collecting representative traces and online monitoring techniques capable of being deployed in production environments to address these challenges in sound trace collection and optimizing instrumentation overhead and interference.

1.5 Summary of Contributions

The contributions of this thesis can be summarized as follows:

- We review the instrumentation process for runtime verification along with existing frameworks and techniques.
- We introduce a novel, dedicated, versatile, and expressive instrumentation framework tailored for the collection of correct traces for runtime verification. Its expressiveness facilitates the capture of events at multiple levels of granularity and its design allows the use to guide the instrumentation and incorporate compile-time pre-instrumentation analyses (C1 and C2).
- We implement this framework in BISM, a state-of-the-art tool designed for JVM languages. The instrumentation language strikes a balance between expressivity and usability. BISM offers two languages for instrumentation specification: an API-based approach, granting users granular control over the instrumentation process, and an external DSL approach focused on RV, providing a declarative mechanism for specifying instrumentation.

- We define residual runtime verification for parametric properties [CR09, BFH+12]. We instantiate it at the intra-procedural level using novel overapproximation approaches for the program behavior that relies on the upward closure of the monitor automaton. This technique integrates both the semantics of the property under verification and the program to optimize the instrumentation process and reduce instrumentation points (C2). It applies to both safety and co-safety properties and is agnostic to specific static analysis techniques. This modularity enables seamless integration within RV approaches and in combination with traditional static analyses. We evaluate its effectiveness on a set of benchmarks.
- We formalize and define trace soundness and faithfulness for concurrent traces. These trace qualities apply to any set of monitorable properties and serve as guidelines for assessing the correctness in representing the behavior of a concurrent program (C1).
- We present a novel approach for collecting *concurrent traces* from concurrent programs for online runtime monitoring. This approach establishes on the fly the causal ordering of events using a novel vector clock algorithm that does not require blocking the execution (C1).
- We define *trace monitorability* for concurrent traces based on automata-based formalisms. We introduce a causal dependence relation derived from the given property and establish necessary conditions for trace monitorability, thereby ensuring sound monitoring outcomes. (C2 and C3).
- We implement and evaluate the *opportunistic monitoring* approach for the online monitoring of multithreaded programs. This approach, originally presented in [EH18], avoids additional synchronization when monitoring a property using existing monitoring frameworks. Global properties are specified over regions of the program assumed to execute atomically, whereas thread-local monitors are deployed for each thread. This approach is capable of reasoning on concurrent events and monitoring general behavioral properties and is evaluated on a set of benchmarks (C1, C2, and C3).
- We introduce a modular approach to dynamic program analysis for JVM-based languages that integrates our instrumentation framework with a complex event processing engine [Hal18]. Equipped with the ability to capture fine-grained events and perform guided instrumentation, this approach allows for the decoupling of instrumentation and analysis. This enhances expressiveness, promotes reusability, and allows for simultaneous and independent analyses. The approach is demonstrated to be versatile, and capable of supporting various analyses such as monitoring, profiling, coverage measurement, and complex event generation (C1 and C2).
- We present several use cases to demonstrate how BISM can be used as a generic tool for instrumenting and analyzing programs. Using the same abstractions users can perform a range of static and dynamic analyses. We demonstrate the use of BISM in activities such as mutation testing, obfuscation, and instrumentation for other dynamic analyses.
- We evaluate our instrumentation framework and the proposed monitoring techniques using a comprehensive set of benchmarks. The benchmarks include synthetic programs and real-world applications containing both sequential and concurrent programs. Our results demonstrate the effectiveness of the proposed methods in addressing the challenges of runtime verification (C1, C2, and C3).

1.6 Structure of the Thesis

This thesis is organized into several parts, each serving a distinct purpose in building the overall argument for the contributions made in the fields of runtime verification and instrumentation.

Chapter 2: We review the basic concepts and definitions relevant to traces and runtime verification used throughout the thesis.

Chapter 3: We review various concepts and considerations related to program instrumentation, laying the groundwork for subsequent discussions.

Chapter 4: We present the instrumentation model of our framework, detailing the concepts of join points, advice, shadows, selectors, and transformers.

Chapter 5: We present the implementation of BISM, a state-of-the-art tool for JVM languages, which implements our proposed instrumentation framework.

Chapter 6: We introduce and define the concept of residual runtime verification for parametric properties and present our instantiation of this concept at the intra-procedural level using novel overapproximation approaches.

Chapter 7: We present the notions of sound concurrent traces and the monitorability of such traces, as well as our approach for collecting them based on vector clocks.

Chapter 8: We present our contributions to the opportunistic monitoring of multithreaded programs.

Chapter 9: We present a comprehensive evaluation of our proposed framework and methods through various use cases.

Chapter 11: We detail how expressive instrumentation can be combined with a Complex Event Processing engine for wider dynamic program analysis, utilizing our instrumentation framework in monitoring, profiling, testing, and coverage measurement.

Chapter 12: We summarize the contributions of this thesis and outline directions for future work.

1.7 Associated Publications

The work presented in this thesis has been published in the following publications:

[SKF20] (RV 2020, Tool Paper) We present our instrumentation framework BISM as a tool and evaluate it on a set of benchmarks.

[SMF23] (STTT, Journal Article) We expand BISM, detail our instrumentation model, and include various capabilities such as transformer composition.

[SF22a] (SAC '22, Short Paper) We extend BISM to capture and prepare models that abstract program behavior at the intra-procedural level.

[SF22b] (VSTTE 2022, Research Paper) We present residual runtime verification for parametric properties.

[SF23d] (SPIN 2023, Research Paper) We present sound concurrent traces for online monitoring.

[SEF23] (FASE 2023, Research Paper) We present opportunistic monitoring for multithreaded programs.

[SFH23a] (ISSRE'23, Tool Paper) We present a framework for dynamic program analysis using BISM and complex event processing.

[SF23b] (RV 23, Tool Paper) We present a domain-specific language for BISM aimed at simplifying instrumentation specification for RV.

[SF23c] (RV 23, Tutorial Paper) We explore various scenarios of instrumentation for runtime verification, ranging from basic monitoring to advanced use cases.

CHAPTER 2

Preliminaries

Contents

2.1	**Programs, Methods and the CFG**	**13**
2.2	**Event Traces, Properties, and Monitoring**	**15**
2.3	**Parametric Monitoring**	**16**
2.4	**Upward Closure**	**17**
2.5	**Relations, Partial and Total Orders**	**18**
2.6	**Concurrent Executions**	**18**
2.6.1	Actions	18
2.6.2	Execution Order	20
2.6.3	Concurrent Execution	21
2.7	**Vector Clocks**	**21**

In this chapter, we recall concepts related to monitoring in general, programs, instrumentation, and verification. We assume basic familiarity with automata theory such as the definitions of a finite-state machine, words, runs, and acceptance, and refer to [HMU06] for more details.

2.1 Programs, Methods and the CFG

All the source code examples in this thesis are written in Java, however, the concepts and techniques presented apply to any JVM language unless stated otherwise. Developers usually write their programs in a high-level language such as Java [Ora23b], Kotlin [Jet23], and Scala [Lig23]. The source code is then compiled into an intermediate form known as bytecode [Ora23c] which can be executed by the Java Virtual Machine (JVM) at runtime.

A program P can be defined as $P = \{C_1, C_2, \ldots, C_n\}$, where C_i is a class which contains a set of methods $\{m_1, m_2, \ldots, m_k\}$. We denote the set of all methods of a program P as *Methods*. A method m usually contains a sequence of bytecode instructions, we denote its set of instructions as $Instructions_m$. Let also *Instrs* be the set of all instructions in the program. Some methods in Java lack any bytecode representation, such as native methods and abstract methods, and as such their set of instructions is empty. Native methods are implemented in native languages via Java Native Interface (JNI) [Ora23a], and abstract methods have no implementation in their declaring class.

Throughout this thesis, the term *instruction* refers explicitly to a bytecode instruction unless stated otherwise. Bytecode instructions are more low-level than source code instructions, where a single source code statement can

be translated into multiple bytecode instructions. We denote the set of all bytecode instructions in a program as *Instrs*.

EXAMPLE 1 (JAVA METHOD) In the Listing 2.1, the Java method named `m` initializes an empty list *l* of type `String` and populates it. An iterator *it* is then created for traversing the list *l*. The method then checks whether *it* has a next element and, if so, advances *it* to that element. The method concludes by printing "done," signaling the successful execution of its operations. We show in Listing 2.2 the simplified bytecode resulting from compiling the method in Listing 2.1. Bytecode instructions in JVM operate on a stack-based memory model with access to a local variable array and a constant pool. Each instruction has a set of operands and a result. The operands are pushed onto the stack, the instruction is executed, and the result is pushed back onto the stack.

```
1  public void m() {
2     //Initialize a list of strings
3     List<String> l = new ArrayList<>();
4     l.add("A");
5
6     //Create an iterator to traverse the list
7     Iterator<String> it = l.iterator();
8
9     //Call next if iterator has more elements
10    if (it.hasNext())
11       it.next();
12
13    System.out.print("done");
14 }
```

Listing 2.1: A Java method example source code.

```
1  _new ArrayList
2  dup
3  invokespecial ArrayList.init ()V
4  astore 1
5  aload 1
6  ldc "A"
7  invoke List.add (Object;)Z
8  pop
9  aload 1
10 invokeinterface List.iterator ()Iterator;
11 astore 2
12 aload 2
13 invokeinterface Iterator.hasNext ()Z
14 ifeq L0
15 aload 2
16 invokeinterface Iterator.next ()Object;
17 pop
18 L0
19 getstatic System.out, PrintStream;
20 ldc done
21 invokevirtual PrintStream.print (String;)V
22 return
```

Listing 2.2: JVM Bytecode for the method in Listing 2.1.

Control Flow Graph and Basic Blocks. The control flow graph (CFG) of a method is a directed graph that represents the control flow paths that can be taken during the execution. The control flow graph is constructed statically by analyzing the bytecode of the method. Each node in the graph represents a basic block, which is a maximal sequence of consecutive instructions that can only be entered at its first instruction and exited at its last instruction.

DEFINITION 1 (BASIC BLOCK) A basic block $\mathtt{b}$ is a maximal sequence of consecutive bytecode instructions, denoted as $\mathtt{s}_1 \ldots \mathtt{s}_n$, which satisfies two conditions:

- It can only be entered at its first instruction, $\mathtt{s}_1$, and no other instruction within $\mathtt{b}$ can be the target of a branch (jump) instruction.
- It can only be exited at its last instruction, $\mathtt{s}_n$, and no other instruction within $\mathtt{b}$ can be a branch instruction or a return instruction.

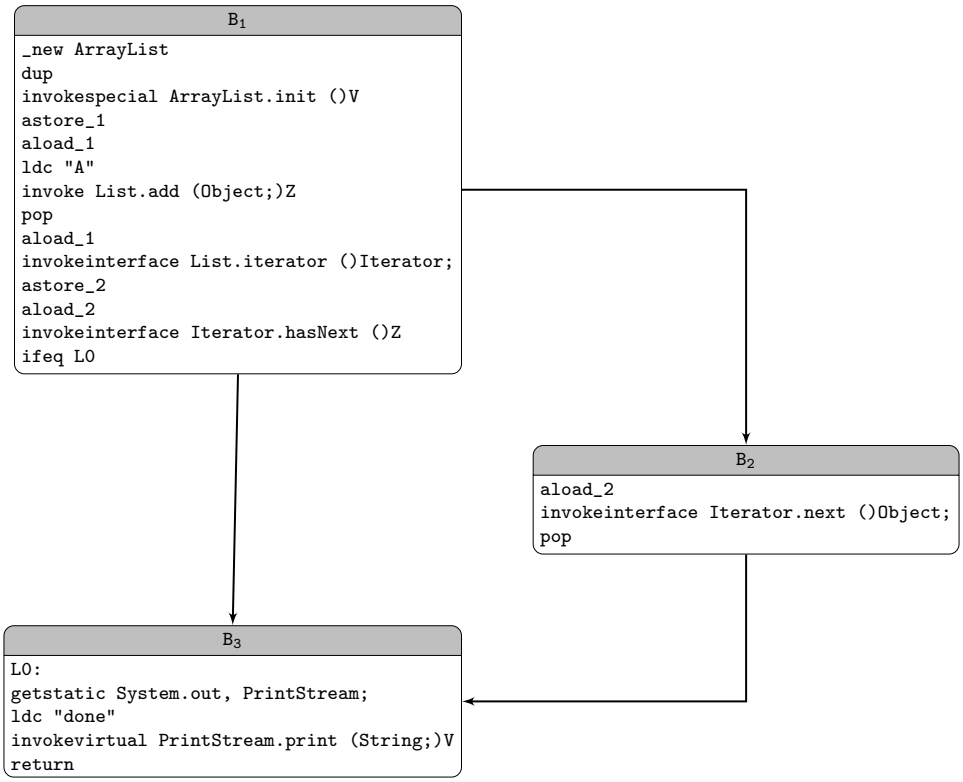


Figure 2.1: Control flow graph for the method in Listing 2.1.

We denote the instructions in a basic block b by $b.instr$. Other variations for the CFGs exist such as the Single-Instruction CFGs where each node contains a single instruction. In this thesis, we use the standard CFG definition where each node represents a basic block. Let *Blocks* be the set of all basic blocks in the program.

DEFINITION 2 (CONTROL FLOW GRAPH) A control flow graph for a method m is a directed graph $CFG_m = (B_m, E_m)$, where B_m is the set of basic blocks in m, and $E_m \subseteq B_m \times B_m$ is the set of directed edges between basic blocks.

An edge $(\mathtt{b}, \mathtt{b}') \in E_m$ corresponds to a possible flow of control from basic block $\mathtt{b}$ to basic block $\mathtt{b}'$. Here $\mathtt{b}'$ is a successor of $\mathtt{b}$ and $\mathtt{b}$ is a predecessor of $\mathtt{b}'$. That means that after executing the last instruction of $\mathtt{b}$, the next instruction to be executed should be the first instruction of one of its successors. We use a boolean flag $b.entry$ to indicate if b is the entry basic block of the method. This block has no predecessors and is the first block to be executed. Conversely, basic blocks with no successors are exit basic blocks, often characterized by a last instruction being a `return` instruction. We use the boolean flags $b.exit$ to indicate if b is the exit block of the method.

EXAMPLE 2 (BYTECODE AND CFG) Consider the method in Listing 2.1. The bytecode for this method is shown in Listing 2.2. Figure 2.1 shows the control flow graph for this method. You can see that $\mathtt{B}_1$ is the entry block and $\mathtt{B}_3$ is the exit block. The edges between the blocks represent the possible control flow paths. For example, $\mathtt{B}_1$ can flow to $\mathtt{B}_2$ or $\mathtt{B}_3$ depending on the evaluation of the `if` statement in the source code (Line 10).

2.2 Event Traces, Properties, and Monitoring

Let Σ be a set of events, Σ^* and Σ^ω are the sets of all finite and infinite traces over Σ, respectively. A finite trace is a sequence of events, a word in Σ, that can be modeled by a function $t : [1, n] \to \Sigma$ for a trace of length n. We say that an event belongs to the trace, noted $e \in t$, when $e \in \text{codom}(t)$.

A property φ is a *language* over Σ which is a subset of Σ^*. Given a trace t in Σ^*, the set of prefixes of t, noted $pre(t)$,

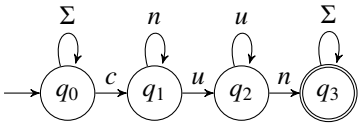


Figure 2.2: Monitor recognizing the language of bad prefixes for the *SafeIterator* property.

is defined as: $pre(t) = \{p \in \Sigma^* \mid \exists s \in \Sigma^* : t = ps\}$. The set of matching prefixes is the set of prefixes of a trace within a given language L.

DEFINITION 3 (MATCHING PREFIXES [BLH12]) Given a language $L \subseteq \Sigma^*$ and a trace $t \in \Sigma^*$, the matching prefixes of t in L is given by: $match_L(t) = pre(t) \cap L$.

Many monitoring techniques and approaches essentially rely on the detection of *bad and good prefixes*, which are intuitively the witnessing sequences allowing a monitor to conclude about monitoring the program based on the trace observed so far.

DEFINITION 4 (BAD/GOOD PREFIXES [KYV01]) Given a language $L \subseteq \Sigma^*$ of finite traces over Σ (or of infinite traces, $L \subseteq \Sigma^\omega$). A finite trace $u \in \Sigma^*$ is a *bad* prefix for L, if $\forall w \in \Sigma^* : uw \notin L$ (or $\forall w \in \Sigma^\omega : uw \notin L$, if L is over infinite traces). Moreover, u is a *good* prefix for L, if $\forall w \in \Sigma^*$: $uw \in L$ (or $\forall w \in \Sigma^\omega : uw \in L$, if L is over infinite traces).

The languages of bad and good prefixes are extension-closed; since every continuation of a bad or a good prefix for a language, L, by a finite word is also a bad (good) prefix for L. When monitoring at runtime, we are interested in reporting a violation/satisfaction of a property from a trace as early as possible. Matching a *bad* (alternatively *good*) prefix is sufficient to produce a final verdict since every continuation of the trace will produce the same result. For instance, the techniques in [BLS11a, FFM12] synthesize a monitor (as a finite-state automaton) that recognizes the good and bad prefixes of the language denoted by a temporal-logic formula or by an automaton over infinite traces.

EXAMPLE 3 (SAFEITERATOR MONITOR) Figure 2.2 shows the monitor that checks for the violation of the **SafeIterator** property which specifies that "*A collection in Java should not be updated when an iterator associated with it is created and being used*". Event c denotes the creation of an iterator associated with a list by calling `list.iterator()`, event u denotes an update on a list by calling `list.add(..)`, and event n denotes calling the next method `iterator.next()` on an iterator. The monitor recognizes the bad prefixes in the traces received from a running program. Note that the monitor reaches the accepting state when seeing the pattern $c.n^*.u^+.n$, as it suffices to conclude that the whole run violates the property. *

DEFINITION 5 (PROPERTY SATISFACTION) We say that a trace $t \in \Sigma^*$, satisfies a property $\varphi \subseteq \Sigma^*$ denoted by $t \models \varphi$ iff $t \in \varphi$. Alternatively, for a safety property φ with its language of bad-prefixes L, $t \models \varphi$ iff $match_L(t) = \emptyset$. And, for a co-safety property φ' with its language of good-prefixes L', $t \models \varphi$ iff $match_{L'}(t) \neq \emptyset$.

2.3 Parametric Monitoring

Monitoring is in practice often performed on parametric monitors that receive events accompanied by runtime information about the objects producing them, allowing to monitor each set of related objects in the program separately. There is a myriad of different approaches to parametric monitoring that differ in the manner they interpret events with runtime information and project these to instances of monitors. See [CR09, BFH+12] for example approaches and [HRTZ18a, FKRT21] for overviews. Here, we sketch a simple and general approach to parametric monitoring that can be adapted to several existing approaches.

We denote the set of variables defined by a parametric monitor by X, and the set of values that these variables can take by V. These values usually correspond to objects in the memory of the execution environment of the program. A variable binding $\theta : X \rightharpoonup V$ maps monitor parameters to their values and $\mathcal{B}$ is the set of all possible bindings in a

program. A parametric event $e\langle\theta\rangle$ is then a pair $(e, \theta) \in \Sigma \times \mathcal{B}$. We denote the set of all parametric events as $\Sigma\langle X\rangle$ and a parametric trace as a word in $\Sigma\langle X\rangle^*$.

EXAMPLE 4 (PARAMETRIC TRACES) Consider a scenario where a program creates two lists $l1$ and $l2$ and then creates an iterator it on $l1$. While monitoring the property from Example 3, a parametric monitor may get the following parametric trace $\tau = (u, [l \mapsto o(l1)])\ (c, [l \mapsto o(l1), i \mapsto o(it)])\ (u, [l \mapsto o(l2)])\ (u, [l \mapsto o(l2)])\ (n, [i \mapsto o(it)])\ (c, [l \mapsto o(l1), i \mapsto o(it)])\ (u, [l \mapsto o(l2)])$. The event $(c, [l \mapsto o(l1), i \mapsto o(it)])$ denotes the creation of an iterator, event c, where the variable l, representing the associated list, is bound to the runtime object of $l1$ denoted by $o(l1)$ and the variable i, representing the created iterator, is bound to the runtime object of it denoted by $o(it)$.

A parametric property $\Lambda X.\varphi$ (notation borrowed from [CR09]) is then defined over traces of parametric events such that $\Lambda X.\varphi \subseteq \Sigma\langle X\rangle^*$. To monitor each group of related objects separately, a parametric monitor slices a parametric trace according to the (dynamic) values from the program bound to the monitor parameters carried within events. Slicing is achieved by projecting a trace τ on all seen bindings using a projection function denoted by $\tau\downarrow_\theta$. We omit the formal details of the projection here for brevity. The projection results in a set of traces, we refer to as *projected traces*, where each trace contains non-parametric events that correspond to related objects in the program and is sent to a monitor that was spawned specifically for that slice.

DEFINITION 6 (PROJECTED TRACES) Given a parametric trace τ in $\Sigma\langle X\rangle^*$, the set of projected traces is denoted by $Proj(\tau) \subseteq \Sigma^*$, and is defined as:

$$Proj(\tau) = \bigcup_{\theta \in \mathcal{B}} \tau\downarrow_\theta$$

The above is a general definition of projected traces that can be instantiated in different ways to handle online monitoring. At runtime, upon receiving new events from the program, these projected traces will be appended with the new events and checked against the property using a monitor created and associated with the slice.

EXAMPLE 5 (PROJECTED TRACES) Consider τ from Ex. 4. A parametric monitor will check at runtime the relation between $o(l1)$ and $o(l2)$[1] then accordingly produce $Proj(\tau) = \{ucuuncu\}$ if $o(l1) = o(l2)$ or $Proj(\tau) = \{ucnc, uuu\}$ if $o(l1) \neq o(l2)$.

DEFINITION 7 (PARAMETRIC PROPERTY SATISFACTION) A parametric trace $\tau \in \Sigma\langle X\rangle^*$ satisfies a parametric property $\Lambda X.\varphi$ denoted by:

$$\tau \models \Lambda X.\varphi \stackrel{\text{def}}{=} \forall t \in Proj(\tau) : t \models \varphi$$

2.4 Upward Closure

We recall the notions for subwords and their closures for regular languages. We refer to [KNS16] for full details and borrow their definitions. For a word $x \in \Sigma^*$, the length of x is denoted by $|x|$, and for $1 \leq i \leq |x|$, let x_i denote the i-th letter of x. We denote the empty word by ϵ. A *subword* is obtained by removing certain letters from a word at arbitrary positions, and, a *superword* is obtained by inserting any number of letters into a word at arbitrary positions. We say that a word x is a subword of y, denoted by $x \sqsubseteq y$, equivalently y is a superword of x when there are positions $0 < p_1 < p_2 < \ldots p_l \leq |y|$ such that $x[i] = y[p_i]$ for all $1 \leq i \leq l = |x|$.

EXAMPLE 6 (SUBWORDS) For $\Sigma = \{a, b, c\}$, we have $\varepsilon \sqsubseteq ab \sqsubseteq acba$. *

[1] This can checked with == in Java.

DEFINITION 8 (UPWARD CLOSURE OF A LANGUAGE) For a language $L \subseteq \Sigma^*$, the upward closure of L, is denoted by $\uparrow L$ and defined as $\{x \in \Sigma^* \mid \exists y \in L : y \sqsubseteq x\}$.

For any language $L \subseteq \Sigma^*$, we have $L \subseteq \uparrow L$. Moreover, a language L is *upward-closed* if $L = \uparrow L$. For a regular language L recognized by a non-deterministic finite-state automaton (NFA), we can obtain an NFA recognizing $\uparrow L$ by simply adding transitions without increasing the number of states. More precisely, given an automaton $A = (\Sigma, Q, \delta, Q_0, F)$ recognizing L, the NFA $A^{\uparrow} = (\Sigma, Q, \delta', Q_0, F)$ recognizing $\uparrow L$ is obtained by adding a self loop on every state $q \in Q$ and every letter $s \in \Sigma$ such that $\delta' = \delta \cup \{\langle q, s, q\rangle \mid q \in Q, s \in \Sigma\}$.

EXAMPLE 7 (UPWARD CLOSURE) Consider the language L over the alphabet $\Sigma = \{a, b, c\}$, defined as words containing the subword ab, i.e., $L = \{x \in \Sigma^* \mid ab \sqsubseteq x\}$. The upward closure of L, denoted as $\uparrow L$, includes all words that have ab as a subword. For instance, if $ab \in L$, then $\varepsilon \sqsubseteq ab$, and $\varepsilon \in \uparrow L$. Similarly, if $acba \in L$, words like $baacba$ or $acbca$ are in $\uparrow L$. The upward closure encompasses words with the specified subword, allowing the insertion of letters at different positions while maintaining the inclusion of ab.

2.5 Relations, Partial and Total Orders

Given two binary relations, $S \subseteq X \times Y$ and $T \subseteq Y \times Z$, we denote their composition by $S \circ T = \{(x, z) \in X \times Z \mid \exists y \in Y : (x, y) \in S \wedge (y, z) \in T\}$. Let $R \subseteq X \times X$ be a binary relation on the set X, we can compose a relation with itself n times, denoted by R^n such that $R^1 = R$ and $R^{n+1} = R^n \circ R$.

The transitive closure of R is the smallest relation on X that contains R and that is transitive ($\langle x, y\rangle \in R$ and $\langle y, z\rangle \in R$ implies $\langle x, z\rangle \in R$), is denoted by $R^+ = \bigcup_{i=1}^{\infty} R^i$. For a relation on a finite set of n elements, the transitive closure of R can be determined by $R^+ = \bigcup_{i=1}^{n} R^i$. Moreover, the inverse of a relation R is denoted by $R^{-1} = \{\langle y, x\rangle \mid \langle x, y\rangle \in R\}$.

EXAMPLE 8 (TRANSITIVE CLOSURE) For a binary relation R on the set $X = \{a, b, c\}$ where $R = \{(a, b), (b, c)\}$, the transitive closure, R^+, is: $R^+ = R \cup \{(a, c)\} = \{(a, b), (b, c), (a, c)\}$

A binary relation $R \subseteq X \times X$ is a *partial order* if it is reflexive ($\forall x \in X : \langle x, x\rangle \in R$), antisymmetric ($\langle x, y\rangle \in R$ and $\langle y, x\rangle \in R$ implies x=y), and transitive. The transitive closure of R is the smallest relation on X that contains R and that is transitive ($\langle x, y\rangle \in R$ and $\langle y, z\rangle \in R$ implies $\langle x, z\rangle \in R$), is denoted by $R^+ = \bigcup_{i=1}^{\infty} R^i$. Moreover, R is a *linear* or *total order* if it is connected ($\forall x, y \in X : \langle x, y\rangle \in R$ or $\langle y, x\rangle \in R$).

The set X together with the partial order R is called a partially ordered set denoted by (X , R). A subset Y of the partially ordered set X is ordered by the restriction of R to Y, i.e. by $R \cap (Y \times Y)$. A *linearization* of a partial order R is a total order $R' \subseteq X \times X$ such that $R' \supseteq R$. Any partial order on a finite set X can be extended to a total order on X.

EXAMPLE 9 (LINEARIZATION OF A PARTIAL ORDER) Given a set $X = \{a, b, c, d\}$ and a partial order $R = \{(a, b), (a, c), (c, d)\}$. You can see that not all elements in X are related to each other. For example, b and d are not related. One linearization of R can be the total order $R' = \{(a, b), (a, c), (c, d), (b, d)\}$ another linearization can be $R'' = \{(a, b), (a, c), (c, d), (d, b)\}$

2.6 Concurrent Executions

In this section, we characterize a concurrent execution in a multi-threaded program. We introduce the notion of *actions*, describe their attributes, and classify them into types. We then define the orders that govern the execution of these actions and finally define a concurrent execution as a partially ordered set of actions.

2.6.1 Actions

An action represents the smallest observable execution step in a program. Observable in the sense that it is visible to the execution environment, and can be monitored or debugged. Examples of actions are method calls, method returns, and field reads. We define actions as follows:

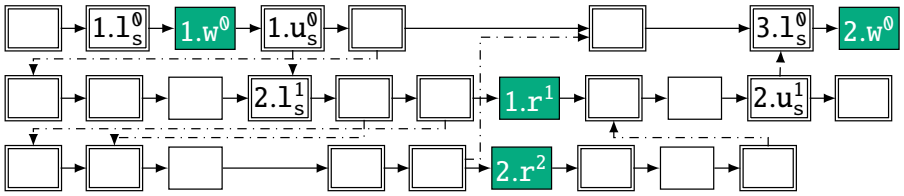


Figure 2.3: A concurrent execution of 1-Writer 2-Readers.

Formally, an action is defined as follows:

DEFINITION 9 (ACTION) An action is a tuple $\langle \texttt{id}, \texttt{ctx} \rangle$, where `id` is a unique identifier, and `ctx` is a set of attributes given by $\texttt{ctx} = \{\texttt{key} \mapsto \texttt{value}\}$.

We denote the set of all actions in an execution by $\mathbb{A}$. The unique identifier `id` distinguishes different instances of the same action. The context `ctx` may contain any static or dynamic information associated with the action. For example, the context may contain the label `lbl`, the thread identifier `tid`, a resource identifier `resid`, a value `value`, or a memory address. Although the label `lbl` is part of the context, it is often explicitly depicted for clarity and to distinguish actions.

To retrieve attributes from an action context, we use the dot notation `a.key` which is defined as follows $\texttt{a.key} = \texttt{value} \iff (\texttt{key} \mapsto \texttt{value}) \in \texttt{a.ctx}$. For instance, `a.tid` retrieves the thread identifier from the context of action `a`, and `a.lbl` retrieves the label of the action `a`.

For certain actions, we depict them as $\texttt{lbl}(\texttt{tid}, \texttt{resource}, \texttt{value})$ to distinguish them from other actions. For example, we depict a write action by $\texttt{write}(1, x, 3)$ to indicate that a thread with id 1 wrote value 3 to variable x.

In diagrams and examples, we depict actions using the notation $\texttt{id.label}^{\texttt{tid}}_{\texttt{res}}$. We omit `resid` when it is absent, and `id` when there is no ambiguity.

The set of all program actions is denoted by $\mathbb{A}$. We distinguish between two types of actions: *regular actions* (RAs) and *synchronization actions* (SAs).

Regular actions are often relevant to the properties we aim to monitor. These actions are not used for coordination and synchronization between threads.

EXAMPLE 10 (REGULAR ACTIONS) Fig. 2.3 shows the execution of a 1-Writer 2-Readers program with one thread for the writer and two threads for the readers. Reader threads need to acquire a shared lock, also used by the writer thread, to access the shared variable. While any reader thread holds the lock, another reader can also acquire the lock concurrently, but the writer must wait until all readers have released the lock. Boxes with single borders represent regular actions. We mention two actions highlighted in green, the regular actions $1.\texttt{w}^0$, $1.\texttt{r}^1$ with `lbl` equals w and r, respectively. The first action is performed by the thread with `tid` equals 0 (the writer), and the second action is performed by one of the threads with `tid` equals 1 or 2 (the readers). This example, while illustrated within the context of a 1-Writer 2-Readers scenario, is versatile enough to represent any program that utilizes a shared variable accessed by multiple threads under the MRSW lock pattern. Importantly, the operations represented here, such as read and write, are conceptual and can be applied to higher-level operations in various contexts, such as cache retrieval and cache update, not just limited to low-level memory operations. Regular actions may be of interest to a monitor that checks the correctness of the program. We omit the labels for several actions for brevity, however, these are actions that the program is executing and are of irrelevance to our example. ∗

Synchronization Actions. We refer to actions that provide synchronization for threads as *synchronization actions (SAs)*. We denote by $\mathsf{SA} \subseteq \mathbb{A}$ the set of all synchronization actions executing in a program. These actions are used for coordination and synchronization between threads when accessing shared variables in order to avoid thread interference and memory consistency errors. For example, to avoid a race condition, two threads operating on the same shared variable are required to synchronize their access in order not to have an inconsistent view of the variable data. By using a lock, the execution environment enforces a critical section on the shared variable, where

each thread needs to acquire the lock to have access and then release it once done. For example, the unlock and lock actions performed by the threads are synchronization actions.

DEFINITION 10 (RELEASE-ACQUIRE RELATION) We denote by RA $\subseteq$ SA $\times$ SA, the relation which specifies when two actions are capable of achieving together the release-acquire semantics on a shared variable and hence synchronize an execution. Given two threads t and u, the following synchronization actions can establish release-acquire semantics:

- `unlock`(t, l)/`lock`(t, l): release/acquire of lock l by t;
- `fork`(t, u)/`begin`(u): fork of u by t / first action by u;
- `end`(t)/`join`(u, t): last action t / u blocking until t ends;
- `write`(t, x, v)/`read`(t, x, v): value v on a volatile shared variable x;
- `notify`(t, s)/`wait`(t, s): notify/wait a signal s.

The relation RA can be derived from the specification of the programming language of the executing program, making the detection of synchronization actions feasible. In terms of shared memory, when a thread tr executes a release action, the visible side-effects of all actions (in particular writes) that were executed by tr are guaranteed to be visible to the thread tr' executing a corresponding acquire action. In Java, the semantics of release and acquire are documented in the memory model [133].

EXAMPLE 11 (SYNCHRONIZATION ACTIONS) We can see in Fig. 2.3 from Example 10 the synchronization actions depicted with double borders. The shown actions $1.\mathtt{l}^{\mathtt{0}}_{\mathtt{s}}$, $1.\mathtt{u}^{\mathtt{0}}_{\mathtt{s}}$ corresponds to the locking and unlocking actions of the lock with `resid` equals `s` by the thread with `tid` equals 0. $*$

2.6.2 Execution Order

In any concurrent execution, there is some causal precedence between actions. For two actions a and a' in a concurrent execution, we say that action a *causally precedes* action a', denoted by $a \prec a'$ when the execution of a' depends on the execution of a. In other words, action a' can only execute after a executes.

Thread Order. Actions executing in the same thread follow a *thread order*. The thread order is the total order of all the actions performed by a single thread. The thread order guarantees that the order of actions in $\mathbb{A}$ performed by thread t is the same as it would be generated by that thread running in isolation.

DEFINITION 11 (THREAD ORDER $\xrightarrow{\mathrm{p}}$) Let a and a' be two actions that are executed in the program, the thread order is a strict partial order $\xrightarrow{\mathrm{p}} \subseteq (\mathbb{A} \times \mathbb{A})$ such that:

$$\xrightarrow{\mathrm{p}} = \bigcup_{tr \in \mathbb{T}} \{ \langle a, a' \rangle \in \mathbb{A} \times \mathbb{A} \mid tid(a) = tid(a') = id(tr) \;\&\&\; a \prec a' \}$$

Synchronization Order. Synchronization actions operating on the same resource induce a synchronization order in an execution following the release-acquire semantics. Synchronization actions provide *release-acquire* ordering semantics [133] and establish a *synchronization order* on the execution. Essentially, an acquire action a' on a resource by some thread synchronizes with the latest release action a, if it exists, on that same resource by some other thread.

DEFINITION 12 (SYNCHRONIZATION ORDER $\xrightarrow{\mathrm{s}}$) Let a and a' be two synchronization actions $\in$ SA, the

synchronization order is a strict partial order $\overset{s}{\rightarrow} \subseteq \mathsf{SA} \times \mathsf{SA}$ such that:

$$\begin{aligned}\overset{s}{\rightarrow} = \{\langle a, a' \rangle \in \mathsf{SA} \times \mathsf{SA} \mid\ & \mathit{resource}(a) = \mathit{resource}(a') \\ \&\&\ & \mathit{tid}(a) \neq \mathit{tid}(a') \\ \&\&\ & \langle a, a' \rangle \in \mathrm{RA} \\ \&\&\ & a \prec a'\}\end{aligned}$$

Note that the thread order also orders synchronization actions produced by the same thread.

Execution Order The execution order of the actions is the combination of the thread order and synchronization order.

DEFINITION 13 (EXECUTION ORDER $\overset{e}{\rightarrow}$) The *execution order* $\overset{e}{\rightarrow} \subseteq (\mathbb{A} \times \mathbb{A})$ is a partial order that is the transitive closure of both *thread* and *synchronization* orders.

From the above definition, any ordering between two actions from different threads relies on the synchronization order, Hence, if two threads never synchronize, there will be no order between any of their actions.

EXAMPLE 12 (EXECUTION ORDER) Fig. 2.3 shows an execution of a 1-Writer 2-Readers program. The solid lines indicate the thread order and the dotted lines indicate the synchronization order. The execution order is the transitive closure of both orders. *

2.6.3 Concurrent Execution

DEFINITION 14 (CONCURRENT EXECUTION) A concurrent execution is a partially ordered set of actions, represented as a pair $\langle \mathbb{A}, \overset{e}{\rightarrow} \rangle$, where $\overset{e}{\rightarrow} \subseteq \mathbb{A} \times \mathbb{A}$ is a partial order over $\mathbb{A}$.

Two actions a_1 and a_2 are related (i.e., $\langle a_1, a_2 \rangle \in \overset{e}{\rightarrow}$) if a_1 happens before a_2.

EXAMPLE 13 (CONCURRENT EXECUTION) Fig. 2.3 shows a depiction of a fragment of the execution of a 1-Writer 2-Readers program. Each line represents a thread, and each box represents an action. The solid lines indicate the thread order and the dotted lines indicate the synchronization order. The execution is a partially ordered set of actions, represented as a pair $\langle \mathbb{A}, \overset{e}{\rightarrow} \rangle$, where $\overset{e}{\rightarrow} \subseteq \mathbb{A} \times \mathbb{A}$ is a partial order over $\mathbb{A}$.

2.7 Vector Clocks

We review notions of vector clocks as presented in seminal works [Mat88, Lam78]. A vector clock is a mapping from the set of threads in a system to integers, denoted as $V : T \rightarrow \mathbb{Z}^+$, where T is the set of all threads and $\mathbb{Z}^+$ represents the set of non-negative integers. There are three primary operations associated with vector clocks:

- **Join** ($\sqcup$): This operation computes the element-wise maximum of two vector clocks. Given two vector clocks V_1 and V_2, their join V_3 is defined such that $\forall t \in T : V_3(t) = max(V_1(t), V_2(t))$.
- **Comparison** ($\leq$): Vector clocks are compared by comparing their values element-wise. For two vector clocks V_a and V_b, we say $V_a \leq V_b$ if and only if $\forall t \in T : V_a(t) \leq V_b(t)$.
- **Increment** (***inc***): To increment the clock for a specific thread t, the *inc* function is used: $inc_t(V) = \lambda u.$ if $(u = t)$ then $(V(u) + 1)$ else $(V(u))$.

EXAMPLE 14 (VECTOR CLOCKS) Consider a program with three threads $T = \{t_1, t_2, t_3\}$. Initially, all vector clocks are at zero: $V_{init}(t) = 0$ for all $t \in T$. Let's examine an example where these threads interact:

1. Thread t_1 performs an action, so its clock is incremented: $V_{t_1} = inc_{t_1}(V_{init})$.
2. Concurrently, thread t_2 performs an action: $V_{t_2} = inc_{t_2}(V_{init})$.
3. t_1 and t_2 synchronize, and t_1 updates its clock by taking the join of its clock with t_2's: $V'_{t_1} = V_{t_1} \sqcup V_{t_2}$.
4. Finally, if t_3 has not performed any action, and we wish to compare its vector clock with t_1's, we would find that $V_{t_3} \leq V'_{t_1}$.

Through this example, we see how vector clocks can be used to track the causality of events across different threads in a concurrent or distributed program. *

Part I

Program Instrumentation

CHAPTER 3

Program Instrumentation and Existing Frameworks

Contents

3.1 Introduction **27**
3.2 Understanding Instrumentation **27**
3.2.1 Unveiling the Complete Picture 27
3.2.2 Observing the Execution 28
3.3 Instrumentation for Runtime Verification **29**
3.3.1 The Program 30
3.3.2 The Observation 30
3.3.3 The Analysis 32
3.3.4 The Instrumentation Language 32
3.4 Instrumentation Requirements **33**
3.5 Evaluating Instrumentation **34**
3.6 Existing Instrumentation Frameworks **34**
3.6.1 Bytecode Manipulation Libraries 34
3.6.2 Aspect-Oriented Approaches 36
3.7 The Need for a Comprehensive Instrumentation Framework **40**
3.8 Conclusion **41**

Chapter abstract

This chapter introduces instrumentation as a mechanism to observe the behavior of a program at runtime. Instrumentation is crucial in runtime verification because it should ensure that monitors are fed with relevant and accurate information about the executing program under monitoring. While expressive instrumentation is desirable to handle any possible monitoring scenario, instrumentation should also efficiently capture the just-needed information and impact the monitoring program as little as possible. We comprehensively overview the instrumentation process, its requirements, and considerations for single and multithreaded programs. We then present various metrics for evaluating the efficiency and effectiveness of instrumentation. We compare existing instrumentation frameworks and demonstrate how they can be used to instrument a program to monitor a property. We also discuss the limitations of existing frameworks and how they can be addressed. We then present an outline for a comprehensive instrumentation framework that can handle an extensive range of instrumentation needs.

3.1 Introduction

Motivation. Verification and analysis techniques are designed to focus on certain behavioral aspects of the system under study. Consequently, they require distinct behavioral *models* that accurately encapsulate the specific aspects of the system that are relevant to their intended reasoning.

The predominant approach to modeling software behavior in monitoring and runtime verification involves observing the software execution and abstracting it into a *trace of events*. Extracting these events frequently relies on *instrumentation*, a technique that entails transforming the base program. Instrumentation consists of two main steps: 1) identifying the program points corresponding to the events of interest, and 2) inserting additional code into the base program to extract information. Choosing the appropriate instrumentation framework for a monitoring scenario is a crucial decision, as it involves a variety of important factors such as the observation granularity, specification, and trace semantics [FKRT18]. The chosen framework must meet the monitoring goals and requirements.

Selecting the right instrumentation framework for a monitoring scenario in runtime verification is critical, as it requires careful consideration of specific factors. These include the precise monitoring goals, the level of observation granularity needed, the nature of the specifications to be monitored, and trace semantics [FKRT18]. It is essential to ensure that the instrumentation framework aligns precisely with the intended results of the runtime verification process. This alignment guarantees that the selected approach not only meets the defined monitoring goals but also adheres closely to the requirements

Contributions. This chapter contributes to the understanding of the instrumentation process, its considerations, and the differences between well-adopted instrumentation frameworks in runtime verification.

- We overview behavioral model generation for verification techniques.
- We compare alternative approaches to program observation.
- We discuss the instrumentation process and considerations for runtime verification. We look into these considerations from the perspective of the program, the analysis and observation requirements, and the instrumentation framework.
- We overview general requirements for instrumentation.
- We cover metrics for evaluating the efficiency and effectiveness of instrumentation.
- We compare existing instrumentation frameworks and outline a comprehensive instrumentation framework that can handle an extensive range of instrumentation needs.

Chapter organization. The chapter is organized as follows. Section 3.2 presents the instrumentation process, its role, alternative approaches. Section 3.3 covers the instrumentation considerations for runtime verification. Section 3.4 discusses general requirements for instrumentation. Section 3.5 covers metrics for evaluating instrumentation. Section 3.6 compares existing instrumentation frameworks. In Section 3.7 we discuss the need for a comprehensive instrumentation framework. In Section 3.8, we conclude the chapter.

3.2 Understanding Instrumentation

Dynamic analysis and verification techniques such as testing, profiling, and runtime verification involve examining a program while it is running. These techniques analyze a behavioral model extracted from the program in order to identify errors, bugs, or unusual behaviors. In this section, we overview a crucial component of these techniques: *instrumentation*. We discuss various considerations that affect the choice of the instrumentation technique. Specifically, we focus on managed languages, with a particular emphasis on JVM languages.

3.2.1 Unveiling the Complete Picture

Verification techniques necessitate abstracting the behavior of a program into a suitable model and subsequently verifying whether this model adheres to properties. This model represents the *actual* behavior of the system suppressing irrelevant details and enabling the application of automated analysis. The term model is extended here

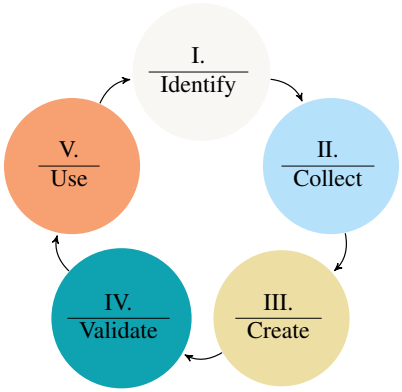


Figure 3.1: Cyclic process of model generation.

to include any artifact generated to represent the system's behavior including logs, traces, automata, etc. Figure 3.1 illustrates the typical steps involved in the process of generating such models.

The model generation process begins by **identifying elements of interest** (I) within the program. This step involves recognizing relevant elements based on user observation or a systematic analysis, which could include structural components or specific actions. Following this, the process involves **collecting information** (II) from these identified elements. Depending on the data required, this could be done statically or at runtime, especially when certain required information, such as the values of variables or program input, is only available during runtime. Once the necessary information has been gathered, along with any other assumptions, the next step is **creating the model** (III). Typically, the model is a mathematical object or maybe a log file, designed to be suitable for the analysis task. To ensure its accuracy and suitability for the intended analysis, the model may then undergo a **validation process** (IV). Finally, the **model is used for the intended analysis** (V). It can serve various purposes, such as being compared with another model in processes like model checking [CHVB18] or runtime verification [BFFR18a], or it can be used to generate test cases, thereby achieving the desired analysis results. While steps I and II assume a white-box approach to model generation, some processes start with execution traces, such as specification mining [BF72, KBM14, OHF+14]. Moreover, these steps can overlap, and the cycle can be reiterated for refinement.

3.2.2 Observing the Execution

Observation is crucial for the completion of steps I, II, and III of the model generation process for dynamic techniques. Different methods can be utilized for observing an executing program, each offering unique capabilities. Some are beyond the scope of this discussion, such as hardware performance counters and operating system tracing. In the following, we focus on JVM-based languages.

Debugging Interfaces

Managed languages usually feature built-in debugging interfaces, such as the Java Debug Interface (JDI). Debuggers can use these interfaces to manage a range of events, including setting breakpoints and watchpoints, performing step-by-step execution, controlling threads, handling exceptions, and inspecting or modifying variables and fields. However, while these debugging interfaces are powerful for step-by-step program inspection, they are not inherently designed for automated or broad-scale data collection. Manual setting of breakpoints and data extraction for thousands of events can be impractical, often leading users to resort to ad-hoc scripting for automation. This requires proficiency in compatible scripting languages and familiarity with the debugger's API. Moreover, debugging interfaces are limited to the event types and information they inherently provide.

Execution Callbacks

Managed languages like Java provide capabilities to register callbacks for specific execution events via the Java Virtual Machine Tool Interface (JVMTI). This native interface allows interaction with the JVM's internal events, including thread start and end, method entry and exit, field access and alteration, exception handling, and more. This

Figure 3.2: Considerations for RV Instrumentation

level of coverage offers detailed JVM activity monitoring, including monitoring internal environment events not directly linked to specific program instructions. Nevertheless, JVMTI presents limitations. Its scope of observation is confined to the event types and data it provides. Should custom events or additional context information be needed, the process can be complex and demand substantial platform knowledge.

Instrumentation

Instrumentation involves augmenting a program with additional code to collect data during its execution, often automated with the help of instrumentation languages. Unlike other observation techniques, instrumentation lets the user define events by identifying arbitrary sequences of instructions in the program, capturing a wide variety of behavioral aspects from fine-grained events such as local variable assignments to coarse-grained ones like method executions. While it is possible to perform instrumentation manually, it becomes significantly complex and prone to errors for large programs or when only a compiled version of the code is accessible. Compared to other observation methods, instrumentation is usually more portable, simpler, and performs better. In JVM, for instance, the added code for instrumentation can be optimized by the Just-In-Time (JIT) compiler, which can significantly reduce the instrumentation overhead. However, it can't observe internal environment-executed events not directly linked to specific program instructions, such as Java's garbage collection. Also, injecting code can modify program behavior, which can cause issues if the program is already in production.

3.3 Instrumentation for Runtime Verification

A property under runtime verification represents a set of constraints or behaviors that the system is expected to adhere to, formalized in terms of abstract events drawn from an alphabet denoted as Σ_a. The process of runtime verification usually encompasses three stages. First, a monitor is created from the property, referred to as *monitor synthesis*. This monitor interprets events from a program and gives outcomes based on the property's current satisfaction. Next, the program is instrumented to generate relevant events for the monitor, known as system

instrumentation. Seen as a generator of concrete events, denoted as Σ_c, the program's execution should be mapped into a trace of abstract events, rendering it suitable for runtime analysis. Instrumentation plays a key role in capturing these concrete events and mapping them into corresponding abstract ones to construct the suitable trace which is the model needed by the monitor.

These concrete events correspond to locations in the program source code, which we will refer to as program *shadows*, and execute at specific points in the program. Throughout this thesis, we will adopt AOP terminology (see 3.6.2) and refer to these points in the execution as *join points*. The instrumentation process consists of adding extra code, known as *advice*, around these program shadows to capture the concrete events when the program executes. Lastly, the system's execution is analyzed by the monitor, either in real-time or post-execution from logged events, a phase termed *execution analysis*. Instrumentation is particularly suitable for runtime verification. It provides flexibility in capturing concrete events by pinpointing arbitrary locations in the source code, as opposed to being limited to specific events provided by the execution environment.

We now go through various considerations for various applications of instrumentation for runtime verification, depicted in Figure 3.2. We will go through some of these in the following sections.

3.3.1 The Program

Various aspects of the program must be taken into consideration when selecting an instrumentation language. Figure 3.3 depicts some of these considerations.

In **concurrent** programs, threads need to be synchronized so that they coordinate safely without interference. While different threads interleave, many events may occur concurrently, and the order in which they occur is not deterministic. If the property being monitored requires reasoning about concurrent events, causality between events must be established. This is achieved by capturing the synchronization points in the program which correspond to the execution of various concurrency primitives. In Java, for instance, programmers may use different concurrency primitives while writing their programs such as locks, atomic variables, and concurrent data structures. Some of these primitives present various challenges for instrumentation due to their complexity, low-level operations, and subtle behavior. Moreover, capturing all synchronization points in a concurrent program with instrumentation is often infeasible, and instrumentation should be able to capture the necessary ones. Given that instrumentation can be costly, it is often desirable to minimize the number of instrumentation points. Moreover, the concurrency primitives are also employed internally by various components of the Java Standard Library for instance. As such, instrumentation may choose to target only visible synchronization points in the program under study. Moreover, other high-level concurrency abstractions exist as well such as the fork-join framework or software transactional memory [Lea00, HMPJH05] or message passing frameworks such as Akka [Akk22].

At the **source** level, it often necessitates compilation facilities and requires access to the application's source code. Weaving at the **bytecode** level has several advantages. It is often high-level enough to easily recognize constructs of the original language, even without direct access to the source code. Moreover, it is portable across different languages as many languages compile to the same bytecode such as Java, Scala, Kotlin, and Groovy.

Program **transformation** (weaving of instrumented code) can occur at different stages as well. **Independent** (or build-time) instrumentation is possible anytime resulting in a new statically instrumented program. However, it is limited to the code packaged within the application itself and may not extend to instrumenting Java class libraries used by the application. **Load-time** instrumentation intercepts class loading and performs instrumentation before a class is linked in the Java Virtual Machine (JVM). This allows also for targeting the libraries and core classes used by the application.

3.3.2 The Observation

In runtime verification, event **traces** serve as models for property-based detection and prediction techniques. An **event** typically captures an important action or a change in the system's state that is under observation. They may represent the program's state at a specific execution point, or they can be triggered by a program action. Depending on the analysis aim, **data** accompanying events may incorporate values from the program's memory, and various **time** representations like current time. Moreover, if events are tracking state changes, the observation should retain some memory instead of having to extract all the values of pertinent variables at each event. Figure 3.4 depicts some of the program observation considerations at runtime.

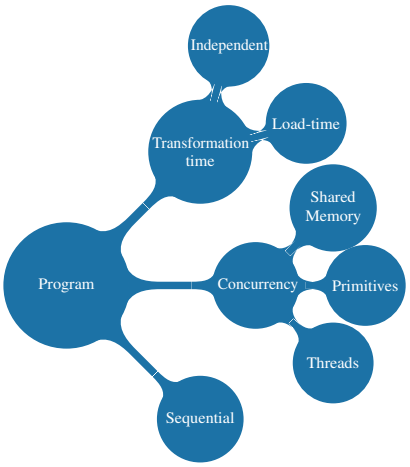


Figure 3.3: Program Considerations

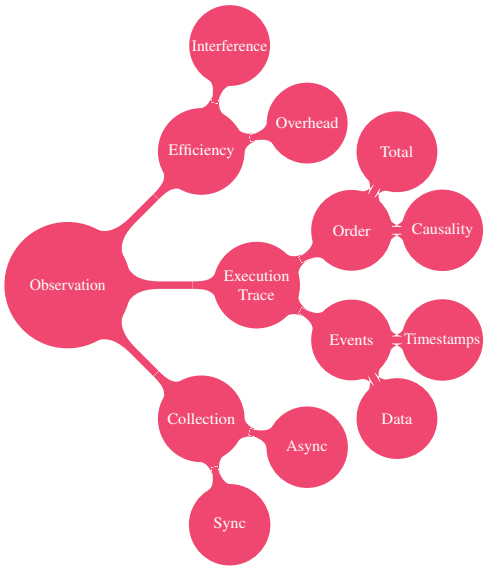


Figure 3.4: Observation Considerations

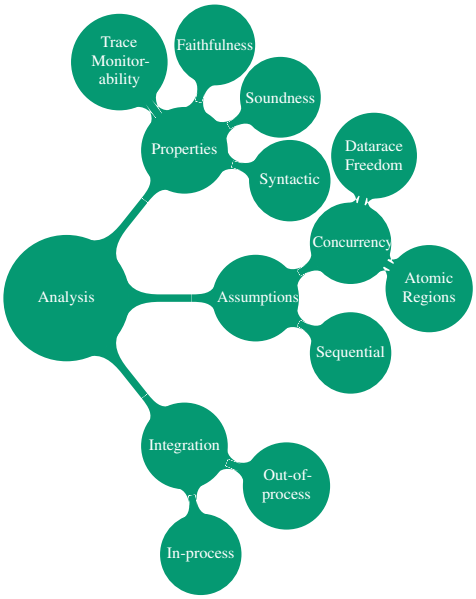


Figure 3.5: Analysis Considerations

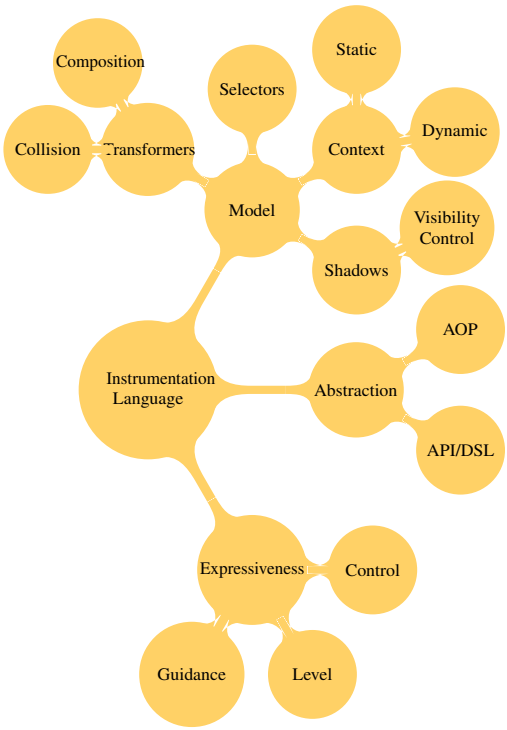


Figure 3.6: Language Considerations

When properties necessitate reasoning about program concurrency, establishing **causality** between events is essential during trace collection. Causality is best represented as a partial order over events which aligns well with different frameworks for understanding concurrent program behavior, such as weak memory consistency models [ANB+95, AG96, MPA05], and Mazurkiewicz traces [Maz86, GK10]. Collecting the trace of events can be done either **synchronously** or **asynchronously**. Synchronous refers to processing data simultaneously with its collection, whereas asynchronous involves receiving events and processing them at a later time. Asynchronous trace collection is ideal for scenarios where the monitoring overhead cannot be afforded and a small delay in the verdict can be tolerated. For instance, in real-time systems where the system is expected to produce a result within a defined strict deadline [SR94].

3.3.3 The Analysis

The execution analysis considerations are depicted in Figure 3.5. Depending on the analysis different **properties** may be desired. For example, if the analysis is to check for the occurrence of a specific event, then the instrumentation should be able to capture the event. Other concurrency-related properties that are concerned with event ordering may be desired such as **soundness**, **faithfulness**, and **trace monitorability** [SF23d]. These properties are affected by the completeness and correctness of the instrumentation. Monitoring techniques generally operate with the **assumption** of instrumentation completeness and correctness. Other approaches such as [TKH21] address runtime verification with incomplete or uncertain information. Some approaches assume certain concurrency-related properties such as **data-race freedom** and **atomic regions** to reduce the instrumentation points hence overhead and complexity. Moreover, the analysis integration with the program has a direct effect on the instrumentation. For instance, for **out-of-process** analysis, the instrumentation should extract all the necessary information to perform the analysis. Whereas in **in-process** analysis, the analysis typically has access to the program's state and can extract the necessary information itself.

3.3.4 The Instrumentation Language

An instrumentation language should equip users to handle three key considerations: identifying relevant program execution points where events are extracted, which correspond to program code elements; specifying the necessary contextual information to be extracted with these events; and defining the destination of these events, detailing how and where they will be consumed. The instrumentation language considerations are depicted in Figure 3.6.

Identifying relevant program locations can be at the bytecode level or the source code level. At these locations which we refer to as **shadows**, a language should facilitate the extraction of either **static** or **dynamic** contextual information. Static information refers to information that is available at compile time, such as the name of a method or the type of a variable. Dynamic information refers to information that is only available at runtime, such as the value of a variable or the current thread. Finally, the instrumentation language should provide a means to consume the extracted information. This can be done by either adding a hook to a monitor class passing this information or by weaving code to the program itself.

The usability of an instrumentation language is further characterized by two important aspects: its **expressiveness** and the level of **abstraction**. Abstraction relates to the complexity of low-level details that users must deal with in order to specify instrumentation. Instrumentation languages are typically provided either as **external** domain-specific languages or as **internal** API-Based languages. External DSLs are often more accessible to domain experts due to the syntax's inherent focus on domain-specific concerns. In contrast, internal DSLs are implemented within a host language; they integrate more seamlessly with it and are more accessible to developers familiar with the host language. Moreover, the advice to be added to the program can be specified in multiple ways. For instance, in AOP (see 3.6.2) this code is specified by the user as Java syntax. Other approaches would require inserting bytecode instructions, while others might have an API to specify the advice.

On the other hand, expressiveness refers to the language's ability to extract substantial information from the bytecode and modify the program's execution. We distinguish here 3 important factors for expressiveness: code **level**, **guidance**, and **control**, depicted in Figure 3.7.

Instrumentation level. The instrumentation level refers to the scope and granularity of the information that can be extracted. For instance, source code instrumentation can target and capture elements that are visible in the source code like method calls, execution of methods, constructors, field setters, and getters. Whereas, the scope of bytecode level instrumentation is the intermediate language of the program, which is more fine-grained as a single source code instruction can be compiled into multiple bytecode instructions. With bytecode-level instrumentation, a monitor can capture the execution of single bytecode instructions, stack values, and local variables.

Instrumentation guidance. Instrumentation guidance refers to the ability to guide the instrumentation process programmatically. In many cases, the instrumentation can benefit from a compile-time analysis to either refine the selection of program locations or extract complex contextual information. For instance, a residual analysis performed pre-instrumentation on the program can identify instrumentation points that can be ignored at runtime. In other cases, complex contextual information such as the control flow graph can be extracted. Without such ability, a user is limited to the basic analysis provided by the instrumentation language to identify instrumentation points and contextual information.

Instrumentation control. Instrumentation control refers to the ability to modify the original program code using the instrumentation language. Some languages are restricted and can only modify the program using a specific DSL or API. In contrast, an unrestricted language can provide full control over the resulting instrumented program. Here one might skip the execution of an event or modify the program's control flow. However, this level of control requires caution, as incorrect bytecode instructions can compromise the integrity of the bytecode.

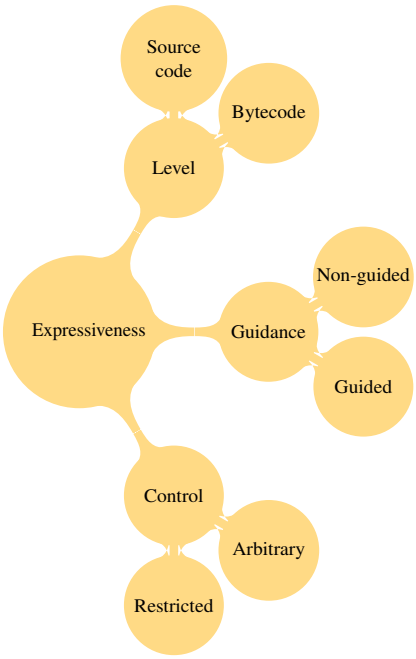


Figure 3.7: Instrumentation language expressiveness considerations.

3.4 Instrumentation Requirements

Correctness and Completeness Monitoring techniques assume the completeness and correctness of instrumentation in capturing events [BFFR18a], however, this assumption is not always valid. For manual instrumentation, it is easy to miss identifying some locations of interest. Also, automated instrumentation can miss some events at runtime due to errors and exceptions raised by the runtime. Some instrumentation techniques wrap the advice with try-catch blocks to avoid system crashes. Although this guarantees the stability of the system, it can lead to missing events without being noticed. It is recommended to disable exception handling when instrumenting the program for the first time.

Non-interference. Ensuring non-interference is crucial to prevent disturbing the program's critical behaviors. Instrumentation should avoid altering aspects such as parallelism, event order, variable values, control flow, and thread scheduling. In a study by [KABM12], the authors identify several interference problems, including deadlocks, state corruption, and JVM crashes, which can be unintended byproducts of instrumentation.

Memory depletion. In-process monitoring makes memory management crucial. Large data storage for analysis risks depleting memory and potentially crashing the application. Hence, an effective data management strategy should be integral to information extraction with instrumentation. Efficient data structures can optimize memory use and prevent interference with memory management. For example, using integer identifiers for event types instead of string descriptions, or extracting unique hash IDs rather than retaining full object references, can be beneficial when applicable.

Environment compatibility. Bytecode verification failures can occur in the JVM for instance due to issues such as incorrect bytecode manipulation, invalid stack or local variable states, control flow problems, or incompatible bytecode versions. In some cases, turning off bytecode verification can be a viable option, but it is not recommended

as it can lead to unexpected behavior and crashes. Moreover, Java enforces a 64KB maximum limit per method, and extensive instrumentation can exceed this limit, leading to compilation errors. These errors can sometimes be avoided by deferring the event construction to a separate method and passing the required object references to it.

3.5 Evaluating Instrumentation

Overhead Evaluating the impact of instrumentation on a program often involves measuring execution time and memory consumption overheads. For precise measurements, using a dedicated machine and repeating the process multiple times is recommended. Profilers like [vis, jprb] can yield the most accurate measurements. Below, we detail some techniques for measuring these overheads. To measure execution time overhead, compare the execution time of the instrumented program with the original one. One method involves inserting timers (via instrumentation) to capture timestamps at the program's entry and exit points. A nonobtrusive alternative is using a command-line benchmarking tool such as [SC], offering features like warmup runs and statistical analysis of results. Memory consumption in the JVM is influenced by multiple factors, including the JVM internals and garbage collection. A good estimate of memory consumption can be obtained by calculating the heap and non-heap memory usage after forcing a garbage collection cycle, measured before the program's exit point. A specialized memory measurement virtual machine like [LBM15] can also be employed.

Affected classes, methods, and instructions. This metric quantifies the impact of instrumentation on the program's codebase. It is measured by counting the number of classes, methods, and instructions that have been instrumented. This metric is useful for evaluating the overall coverage of the instrumentation process and its impact on the codebase. For instance, when capturing method calls, consider all method calls invoked at runtime and compare this to the number of those method calls that have been instrumented.

Instrumentation intensity. This synthetic metric provides an understanding of the extent of the code that has been instrumented. It is a measure of the number of code instructions that have been adjusted for instrumentation purposes, as a proportion of total code instructions. If a larger part of the code has been modified, the instrumentation intensity is higher. This metric is useful for evaluating the overall coverage of the instrumentation process and its impact on the codebase. For instance, when capturing method calls, consider all method calls invoked at runtime and compare this to the number of those method calls that have been instrumented.

Instrumentation code latency. Instrumentation code latency measures exactly the time taken for the execution of the added instrumentation code only. Here the time before and after the advice executes is measured or both timestamps are extracted with the event and the difference is calculated at the monitor side. In concurrent programs, this metric provides insight into the extent to which instrumentation is interfering and affecting the parallelism of the program when compared to the original program and the overall overhead mentioned above provided that it also includes extracting such timestamps.

3.6 Existing Instrumentation Frameworks

Several instrumentation tools and frameworks in different programming languages were developed. In this section, we provide an overview of existing frameworks used for JVM program instrumentation. Nevertheless, there are several tools to instrument programs in different programming languages. For instance, to instrument C/C++ programs AspectC/C++ [CKFS01, SLU05] (high-level) and LLVM [LA04] (low-level) are widely used.

We categorize these approaches into two main types: bytecode manipulation libraries and aspect-oriented approaches. Since expressiveness is a key factor in choosing an instrumentation framework suitable for an instrumentation task at hand, we will discuss these frameworks from the lens of expressiveness (see Figure 3.7).

3.6.1 Bytecode Manipulation Libraries

Bytecode manipulation libraries offer interfaces for the static manipulation of Java bytecode. They enable low-level coverage and facilitate extensive transformations of the bytecode. Such libraries find applications in various domains, including but not limited to, compilers, profilers, obfuscators, and bytecode optimization tools.

These libraries, while offering a lot of expressivity, demand a comprehensive understanding of Java bytecode intricacies, making them less ideal for straightforward instrumentation tasks. For most of these frameworks, users are expected to possess in-depth knowledge of JVM bytecode, including how to interact with local variables and the operand stack. Furthermore, the precise ordering of bytecode instructions is critical to avoid invalid bytecode, yet these libraries offer no guidance in this aspect. A significant challenge also arises in accessing dynamic context, such as the receiver of a method invocation, requiring users to analyze and modify the bytecode to achieve this. The absence of direction in handling dynamic context and the complexity involved in ensuring valid bytecode underscore the limitations of these libraries for simple instrumentation tasks.

ASM [BLC02]. ASM is a Java bytecode manipulation framework utilized by several tools. ASM offers two APIs that can be used interchangeably to parse, load, and modify classes: a visitor-based API and a tree-based API. The visitor-based API is highly efficient, and optimized for quick traversal and manipulation of bytecode. On the other hand, the tree-based API provides a more abstract approach to manipulating classes, as it represents the bytecode structure as a tree structure. This makes it particularly useful for applications where multiple passes or more complex manipulations are required.

```
1  // Necessary imports for ASM classes
2  import org.objectweb.asm.MethodVisitor;
3  import org.objectweb.asm.Opcodes;
4
5  // ... other code to parse bytecode, create a visitor, etc.
6
7  public class CustomMethodVisitor extends MethodVisitor {
8
9     public CustomMethodVisitor(int api, MethodVisitor methodVisitor) {
10       super(api, methodVisitor);
11    }
12
13    @Override
14    public void visitMethodInsn(int op, String owner, String name, String desc, boolean
          itf) {
15    // Check if the method invocation is for creating an iterator from a List
16    if ("java/util/List".equals(owner) && "iterator".equals(name)) {
17       // Duplicate the iterator reference on the operand stack
18       mv.visitInsn(Opcodes.DUP);
19
20       // Load the List object onto the operand stack
21       mv.visitVarInsn(Opcodes.ALOAD, 1);
22
23       // Load the string "create" onto the operand stack
24       mv.visitLdcInsn("create");
25
26       // Invoke the static method receive of the class monitors/SafeListMonitor
27       mv.visitMethodInsn(Opcodes.INVOKESTATIC, "monitors/SafeListMonitor", "receive",
             "(Ljava/lang/Object;Ljava/lang/Object;Ljava/lang/String;)V", false);
28    }
29
30    // Call the original method
31    super.visitMethodInsn(op, owner, name, desc, itf);
32    }
33 }
```

Listing 3.1: Simplified ASM code to capture an event on iterator creation and invoke a monitor.

EXAMPLE 15 (INSTRUMENTATION WITH ASM) Listing 3.1 demonstrates how to use the ASM framework to capture the event of iterator creation from a Java `List` object from Example 1. The code extends the `MethodVisitor` class from the ASM library and overrides the `visitMethodInsn` method. ASM ships with several utility methods to parse the bytecode either from compiled java files (`.class`) or from byte arrays that can be obtained from the JVM. We omit the code for parsing the bytecode and creating a visitor for brevity. The `visitMethodInsn` method is invoked for each method invocation instruction in the bytecode. We specifically look for the invocation of the `iterator` method on a `List` object. When such an invocation is detected, additional bytecode instructions are inserted to capture this event. In ASM, these instructions are added by visiting the correct visitor for each instruction. For instance, `mv.visitVarInsn(Opcodes.ALOAD, 1);` at Line 21, will insert `LOAD 1` instruction into the bytecode. The aim is to correctly invoke a static `receive` method from a hypothetical `SafeListMonitor` class and pass the needed information, which can then handle or log the event as required.

One can see from the example that using ASM requires a good understanding of Java bytecode. Moreover, the index of the local variable containing the `List` object may be different in other methods, for instance, if the method where the invocation is captured is static. This necessitates additional logic to determine the correct index. This example is also simplified for brevity and does not include the necessary code for parsing the bytecode, creating a visitor, or writing the modified bytecode to a file.

Javassist [Chi00]. Javassist is a versatile library for bytecode manipulation that offers two distinct levels of APIs catering to different developer needs. The source-level API is designed for ease of use and allows modifications to be made without a deep understanding of Java bytecode. This is particularly beneficial for developers who prefer working at a higher level of abstraction and wish to avoid the complexities associated with low-level bytecode manipulation. On the other hand, the bytecode-level API offers granular control over class modifications, giving experienced developers the flexibility to implement advanced optimizations or transformations. Despite its robust capabilities, it's worth noting that using Javassist's bytecode-level API requires a good grasp of Java bytecode intricacies. Overall, Javassist provides a comprehensive set of tools suitable for a wide range of bytecode manipulation tasks, from simple code injections to complex program transformations.

Soot [VRCG+99]. Soot is a multifaceted framework designed for Java application analysis as well as bytecode optimization. One of its standout features is the provision of multiple intermediate representations (IRs) of the code, such as Jimple and Shimple, which facilitate different types of analyses and transformations. These IRs make it adaptable for various research and practical applications. Moreover, Soot comes with a rich set of built-in static analyses, making it a powerful tool for complex tasks such as call-graph construction, data-flow analysis, and points-to analysis. These built-in capabilities allow researchers and developers to focus more on custom analysis logic rather than low-level implementation details. Despite its powerful features, Soot has a steep learning curve and is best suited for those who have a deep understanding of program analysis concepts. It is widely used in academic research for its versatility and is considered a go-to framework for those looking to perform in-depth static analysis of Java applications.

BCEL [Apa]. The Byte Code Engineering Library (BCEL) provides an API for both static analysis and dynamic manipulation of Java classes at runtime. It is particularly useful for developers building compilers, profilers, and bytecode optimization tools. BCEL's API comprises packages for analyzing Java classes even when source code is not available, as well as for dynamically generating or modifying class objects. Additionally, it offers tools for displaying target classes and for converting them into various formats such as HTML and assembly language. BCEL does require users to have a solid understanding of Java bytecode.

Expressiviness of Bytecode Manipulation Libraries The presented frameworks are highly expressive instrumentation languages, offering the following key capabilities:

- **Bytecode level instrumentation:** They enable the extraction of events at the bytecode level, where individual instructions can be targeted. This allows for the extraction of low-level information, such as local variables and stack values.
- **Guided instrumentation:** The frameworks support guided instrumentation. Users have the flexibility to write custom code that executes at weave time, guiding the instrumentation process. For example, in Example 15, before adding instructions to the program at Line 18, users can implement static analyzers to determine the necessity of specific instrumentation points.
- **Arbitrary control:** As demonstrated in Example 15, users have the freedom to insert any bytecode instruction, offering arbitrary control over the instrumentation process. Moreover, users can traverse the program's bytecode in any order, allowing for more complex instrumentation scenarios. However, this level of control requires caution, as incorrect bytecode instructions can compromise the integrity of the bytecode.

3.6.2 Aspect-Oriented Approaches

Aspect-oriented approaches simplify the specification of instrumentation directives by employing a *pointcut/advice model*. In this model, users provide Java code snippets, known as advice, that are to be inserted into the code. These

insertion points are specified by pointcuts, which capture the specific locations in the code where the advice should be applied.

Aspect-Oriented Programming

Aspect-oriented programming (AOP) is a broader programming paradigm that aims to increase modularity by allowing the separation of what is called *cross-cutting concerns*. A cross-cutting concern is a concern that affects multiple parts of an application but cannot be cleanly decomposed from the rest of the system in both design and implementation. Examples of cross-cutting concerns include logging, authorization, and persisting data to a database. Runtime verification is also a typical example of a cross-cutting concern that can benefit from AOP. These concerns are often tangled with the business logic of the application, making it difficult to maintain and extend the system. The primary motivation behind AOP is to mitigate problems induced by non-modular code, specifically:

- **Code Scattering**: Occurs when similar code is distributed throughout many program modules. Changing the implementation requires finding and editing all affected code.
- **Code Tangling**: Occurs when two or more concerns are implemented in the same body of code or component. This makes the code more difficult to understand and maintain.

AOP programs consist of the base program and a list of aspects. These aspects are composed with the program using the join point model. Below we discuss the key concepts of this model.

- **Join point**: A well-defined point in program execution that can be identified, such as method invocation or variable modification.
- **Pointcut**: An expression denoting a set of join points.
- **Advice**: The additional behavior, i.e. the crosscutting concern, that needs to be added at a particular join point captured by a pointcut.
- **Aspect**: A construct that encapsulates pointcuts and advice.

An aspect weaver is responsible for composing the base program with the aspects. This process is called *weaving*. The weaver traverses the base program and the aspects, looking for lexical points in the program that match the pointcut designators. The matched lexical points are called *join point shadows*, and advice code will be injected into those places. At run time, the advice will be executed at the run time instances of the lexical shadows which are the join points. This weaving process can be performed at compile time, load time, or run time.

AspectJ

AspectJ [KHH+01b] is the standard implementation of aspect-oriented programming (AOP) for Java. AspectJ ships with a fixed set of join points and pointcut designators that can be used to specify the locations where advice should be applied. These join points include method calls, method executions, field access, object construction, and exception handling. A rich pointcut expression language that includes statically and dynamically evaluated pointcuts allows users to pick out join points and exposes data from the execution context of those join points. For instance, users can specify that advice should be applied to all method calls within a particular package or to all method calls that take a specific parameter. Moreover, dynamic pointcuts can be used to adapt the behavior of the program based on runtime conditions. For example, users can specify that advice should be applied to all method calls that take a parameter of type `String` and whose value is `"foo"`. For dynamically evaluated pointcuts, the weaver injects reflective code into the program. In the previous example, the reflective checks the value of the parameter at runtime and executes the advice only if the condition is satisfied. Control flow pointcuts can also be used to specify that advice should be applied to all method calls that are invoked from a particular method. For example, the `cflow` pointcut can specify that advice should be applied to all method calls that are invoked from a particular method.

AspectJ offers three types of advice: *before* advice, which executes before the join point, and *after* advice, which executes after the join point. It also supports *around* advice, which allows users to override, delay, or alter the execution of the join point. Users can implement aspects using a domain-specific language by writing `.aj` files, or by using its internal API which uses annotations to decorate advice methods. In both cases, the advice code is written in Java and is weaved verbatim into the program.

EXAMPLE 16 (INSTRUMENTATION WITH ASPECTJ) In Listing 3.2 the `SafeListAspect` aspect defines a pointcut named `createIteratorCall`. This pointcut captures the event of calling the `iterator()` method on a `List` object. The `after` advice is where the user specifies the code to be executed when the pointcut is matched. In this case, the advice invokes the `receive` method of a hypothetical `SafeListMonitor` class to handle the event.

```
1 public aspect SafeListAspect {
2
3    pointcut createIteratorCall(List list): call(* List.iterator()) && target(list);
4
5    after(List list) returning(Iterator it) : createIteratorCall(list) {
6      monitors.SafeListMonitor.receive("create", list, it);
7    }
8
9 }
```

Listing 3.2: AspectJ instrumentation to send an event on iterator creation to a monitor.

AspectJ provides the following expressiveness features:

- **Source code level instrumentation:** AspectJ offers join points and access to contextual information only at the source code level. This limits the ability to extract low-level information such as the execution of specific bytecode instructions and the values of local variables and stack values, restricting the scope of instrumentation and analysis. For instance, concurrency primitives such as `synchronized` blocks cannot be targeted with AspectJ. Some extensions have been proposed to handle some of these primitives such as in [BH10], however, these are not part of the standard AspectJ language. Moreover, in monitoring scenarios where the user wants to capture the event of a specific bytecode instruction, such as the execution of a conditional jump i.e. an `if` statement, AspectJ is not suitable.
- **Non-guided instrumentation:** AspectJ does not support custom analyses to guide the instrumentation. The user is restricted to specifying only the code to be inserted. As seen in Example 3.2, users specify only the code that will be weaved into the program such as in Line 6. Any additional code that might be required to execute at weave-time to decide whether to insert the advice or not is not possible with AspectJ. That is why implementing any sort of static analysis within AspectJ required cumbersome compiler customizations, as seen in extending the AspectBench Compiler (*abc*) in [ACH+05, AM07, BLH10]. However, this compiler is not maintained anymore.
- **Restricted control:** Modifying the base the program is restricted in AspectJ. Except for the *around* advice, an advice is not allowed to modify the control flow of the base program. The *around* advice however offers some control over the execution flow as it allows for overriding, delaying, or altering the execution of the join point, primarily through the *proceed()* method. One can skip the execution of a join point, or execute it multiple times.

DiSL

DiSL [MVZ+12] is an instrumentation framework targetting JVM programs. DiSL adopts the pointcut/advice model from AspectJ while also providing low-level access to the bytecode. One of its distinguishing features is an open join point model, which allows the user to create custom pointcuts by writing custom *markers*. Hence, users are not restricted to the provided join point model, and can effectively mark any sequence of bytecode instructions as a join point shadow. DiSL also offers a rich set of static and dynamic context objects that can be used to extract information from the program which can also be extended by the user to facilitate the extraction of additional customized information from the program. It also allows the insertion of thread and local variables into the program.

EXAMPLE 17 (INSTRUMENTATION WITH DISL) Listing 3.3 shows how to use DiSL to capture the event of iterator creation from a Java `List` object. The code defines a custom marker named `IteratorMarker` that captures the event of calling the `iterator()` method on a `List` object. Listing 3.4, shows the advice method `beforeCreateIterator` which instruments the program to invoke the `receive` method of a hypothetical `SafeListMonitor` class to handle the event.

DiSL offers only an internal API for specifying instrumentation where instrumentation specifications, custom markers, guards, and context objects are fully written in Java. The advice code as seen in Listing 3.4 is mixed with

Java syntax and DiSL API calls such as `pc.getReceiver` and `dc.getStackValue`. DiSL then compiles the full body of the advice to produce the bytecode that will be weaved into the program. This differs from AspectJ where the advice code is written in Java and is weaved verbatim into the program.

```
1  public class IteratorMarker extends AbstractDWRMarker {
2
3     public List<MarkedRegion> markWithDefaultWeavingReg(MethodNode method) {
4
5        List<MarkedRegion> regions = new LinkedList<MarkedRegion>();
6
7        // traverse all instructions
8        InsnList instructions = method.instructions;
9
10       for (AbstractInsnNode instruction : instructions.toArray()) {
11          // check for method invocation instructions
12          if (instruction instanceof MethodInsnNode) {
13             // add region containing one instruction (method invocation)
14             MethodInsnNode min = (MethodInsnNode) instruction;
15             if(min.name.contains("iterator") && min.owner.contains ("List"))
16                regions.add(new MarkedRegion(instruction, instruction));
17          }
18       }
19       return regions;
20    }
21 }
```

Listing 3.3: A DiSL custom *marker* to find iterator creation.

```
1 @Before(marker = CreateIteratorMarker.class)
2 public static void beforeCreateIterator(ArgumentProcessorContext pc, DynamicContext dc)
3 {
4
5    Object it = pc.getReceiver(ArgumentProcessorMode.CALLSITE_ARGS);
6    Object list = dc.getStackValue (0, java.util.List.class);
7
8    monitors.SafeListMonitor.receive("create", list, it);
9 }
```

Listing 3.4: A DiSL snippet that invokes a monitor.

DiSL is designed for instrumentation-intensive dynamic analysis scenarios. It implements advanced program composition features such as polymorphic bytecode instrumentation [BMTA16] allowing multiple independent instrumentations to be applied to the same program. It also runs the instrumentation process in a separate JVM process, thereby minimizing perturbation in the observed program. Utilizing a native JVMTI agent, it captures all class loading events in the observed JVM and forwards the classes as byte arrays to the DiSL framework for instrumentation, which is performed using ASM. As such, DiSL requires executing the program in order to instrument it.

DiSL provides the following expressiveness features:

- **Bytecode level instrumentation:** DiSL operates at the bytecode level, where it can target the execution of any single bytecode instruction as a join point. This also allows it to extract high and low-level information from the program, such as local variables and stack values as well as class fields and method parameters. As such it offers a superset of the join points offered by AspectJ, in addition to the ability to extend those. Moreover, since DiSL instruments from the native space by using a JVMTI agent, it can instrument all classes loaded by the JVM, including system classes and classes loaded by other agents. These classes cannot be instrumented using the Java agent approach used by AspectJ. As such, DiSL offers complete bytecode coverage, ensuring that all methods with bytecode can be woven.
- **Guided instrumentation:** With the open join point model, DiSL allows users to define custom markers and context objects, thereby providing more control and guidance for the instrumentation process. This model overcomes the restrictions imposed by predefined join points such as in AspectJ, allowing for more flexibility in instrumentation. As such, users can incorporate within these markers and context objects custom analyses to guide the instrumentation process.
- **Restricted control:** DiSL does not allow users to freely insert arbitrary code and modify the control flow of the program. This is to ensure that the instrumentation does not interfere with the program's execution.

However, it does allow users to specify the order in which snippets are inlined into the program. Unlike AspectJ, DiSL does not support *around* advice, thereby avoiding control-flow modifications with the *proceed()* method. As such, one cannot skip a join point from executing, or execute it multiple times. However, with its support for adding local variables, the user can pass data around a join point. However, DiSL includes a mechanism to implement custom transformation passes utilizing ASM to perform arbitrary bytecode manipulation before applying the DiSL specifications.

3.7 The Need for a Comprehensive Instrumentation Framework

While bytecode manipulation libraries offer a high degree of control over program traversal, allowing for complex manipulations that may be impractical or even impossible to achieve using AOP frameworks, they can be verbose and necessitate deep expertise in low-level bytecode manipulation. On the other hand, AOP frameworks like AspectJ and DiSL simplify the process of specifying what code to insert and where, they also impose limitations on the kinds of transformations that can be achieved. While many scenarios covered in this thesis require the ability to traverse a program's bytecode and insert arbitrary code, this is not possible with AOP frameworks.

Having a comprehensive instrumentation framework that can be used for a wide range of scenarios is highly desirable. As such, there is a pressing need for a more comprehensive approach to JVM code instrumentation. This approach should integrate the ease of use found in AOP languages with the detailed control provided by bytecode manipulation libraries. It should also offer high expressiveness in terms of bytecode coverage, instrumentation guidance, and unrestricted program control. By addressing these gaps, the new approach could offer a balanced and efficient solution for JVM code instrumentation. We sketch below the main challenges that such a framework should address.

Expressiveness and abstraction. The current landscape presents a significant gap: the need for a framework that merges the usability of AOP languages with the granular control of bytecode manipulation libraries. Such a framework would ideally offer granular control with some high-level abstraction, facilitating both ease of use and detailed manipulation capabilities. Ideally, this approach would combine the simplicity of the point-cut/advice model from AOP languages with the flexibility and control of bytecode manipulation libraries.

Ability to guide the instrumentation. With AspectJ, users cannot conduct weave-time analysis as the code snippets in aspects will be weaved with the program verbatim. The fixed pointcut/advice model does not allow for such analysis, as it only provides the ability to specify the advice that will execute at runtime. This restricts the ability to perform any pre-instrumentation analysis that incorporates syntactic or semantic reasoning. In the case of DiSL, although it is possible to implement such analysis through the creation of custom markers and custom context objects, these extensions ultimately rely on low-level bytecode manipulation and do not benefit from DiSL's high-level abstractions provided by its language.

Unrestricted control over the program. Moreover, AOP frameworks like AspectJ and DiSL offer restricted control over the program transformation. This can be a significant limitation in certain scenarios where the user needs to modify the control flow of the program. For instance, in concurrent programs to ensure the atomicity of the program action and the surrounding advice, the user needs flexibility in targeting before and after the location of interest in the program at the same time. Without freely inserting arbitrary code, this is not possible. Another limitation is the ability to insert inline monitors into the program. For instance, to detect a test inversion attack, the user might opt to duplicate the *if* statement in the program and inject a small monitor that detects the attack. This is not possible with AspectJ and DiSL as they only allow the user to specify the advice that will execute at runtime.

Optimized performance. Minimizing the overhead of instrumentation is critical for many applications. For instance, being able to reduce the number of instructions inserted into the bytecode can significantly improve the performance of the instrumented program. For instance in AspectJ, advice is added as an external method, and an invocation to this method is inlined next to the join point of interest. This can lead to a significant increase in the number of instructions to be executed at runtime, producing more context switches and missing compiler optimization opportunities. A better approach for runtime verification is inlining the advice inside the targeted method, as the added code is seldom a complicated aspect added to the program. This is not possible with AspectJ. This can lead to a significant increase in the number of instructions in the bytecode. A comprehensive

approach should aim to minimize performance overhead, making it suitable for real-time or resource-constrained environments.

3.8 Conclusion

In this chapter, we overviewed the instrumentation process, its considerations and challenges. We also overviewed the different instrumentation frameworks and libraries available for JVM instrumentation. We provided examples of how to use these frameworks to instrument a Java program. We discussed the strengths and limitations of each approach. We highlighted the gap between bytecode manipulation libraries and AOP frameworks, highlighting the need for a more comprehensive approach to instrumentation. We also discussed the challenges that such an approach should address. In the next chapter, we will introduce an instrumentation model that will be used to implement a framework that addresses such a need.

CHAPTER 4

A Comprehensive Instrumentation Model

Contents

4.1 Introduction **45**
4.2 Instrumentation Model **45**
4.2.1 Context Objects 45
4.2.2 Join points 46
4.2.3 Advice 46
4.2.4 Shadows 46
4.2.5 Selectors 49
4.2.6 Instruction Visibility 50
4.2.7 Transformers 50
4.2.8 Instrumentation Process 50
4.3 Transformer Composition **50**
4.3.1 Motivations for Composition 50
4.3.2 Composition of Transformers 51
4.3.3 Transformer Collision 51
4.3.4 Order Matters 51
4.4 Conclusion **52**

Chapter abstract

This chapter introduces the instrumentation model for our proposed comprehensive instrumentation framework. Drawing inspiration and borrowing terminology from Aspect-Oriented Programming (AOP), the model serves as the theoretical underpinning for the tool implementation that follows. The model is designed to facilitate precise and context-aware bytecode transformations, offering a robust mechanism for program instrumentation. The model defines key constructs such as join points, shadows, selectors, and transformers. The model incorporates the ability to perform compile-time analysis, providing additional data for making more informed selections. The model also introduces the concept of *composition* of transformers, allowing for complex, chained transformations. Through this model, we aim to provide a rigorous and extensible foundation that can accommodate a wide range of instrumentation needs, from simple code insertions to complex behavioral modifications.

4.1 Introduction

Motivation. Given the challenges, we highlighted in Chapter 1 and Chapter 3, there is a need for a comprehensive instrumentation framework that can accommodate a wide range of instrumentation scenarios. The framework should be *comprehensive* in the sense that it should encompass the capabilities of existing instrumentation frameworks and add additional features to address the limitations we discussed. Such a framework should provide the appropriate abstractions for the instrumentation process, also enabling the incorporation of compile-time analysis which can be used to guide and refine the instrumentation process. Furthermore, it should be based on a foundation facilitating its implementation and extension.

Contributions. This chapter serves as the theoretical underpinning for our proposed comprehensive instrumentation framework. The model is inspired by Aspect-Oriented Programming (AOP) and entails constructs such as join points, shadows, selectors, and transformers. It also incorporates the ability to execute weave-time static analyzers, providing additional data for making more informed and guided instrumentation. Furthermore, it introduces the concept of *composition* of transformers, allowing for complex, chained transformations. We discuss the need for composition and considerations that need to be taken into account when composing transformers.

Chapter organization. The remainder of this chapter is organized as follows: Section 4.2 introduces the main elements of the instrumentation model namely the concepts of context objects, join points, shadows, selectors, and transformers. In Section 4.3, we discuss the composition of transformers and how they can be used to implement complex instrumentation scenarios in a modular fashion. We also cover how to detect collisions between transformers. In Section 4.4, we conclude the chapter.

4.2 Instrumentation Model

In this section, we present the instrumentation model where each of the components corresponds to a specific aspect of the instrumentation process. Recall from Section 2.1, that a program is represented as a set of methods, where each method is represented as a control flow graph (CFG). We defined *Methods* be the set of all methods that have bytecode representation, *Blocks* be the set of all basic blocks, and *Instrs* be the set of all instructions in the program.

4.2.1 Context Objects

We differentiate between two types of contextual information that are available in the program: static and dynamic.

Static context. The static context is a set of attributes available at compile-time. These attributes are extracted from the source code. These include but are not limited to method names, class names, line numbers of instructions, number of method parameters and types of method parameters. Additional contextual information can also be generated as a result of some analysis performed at compile-time such as the control flow graph of a method. We denote the set of all static contexts in the program as C_S.

Dynamic context. The dynamic context is concerned with runtime attributes. These include but are not limited to attributes like the thread executing an instruction, values of variables, or the state of the stack and heap at the time an instruction is reached. They represent the dynamic state of the program at a specific point in time. For instance, the values on the stack and heap are dynamic attributes. These values are often not known at compile-time and can only be determined at runtime. However, we can know their addresses at compile-time. We denote the set of dynamic attributes as C_D.

Together, the static and dynamic contexts provide a comprehensive view of a single point in the program's execution. As such it is important to be capable of capturing as much information as possible about the program's state at that point. This is especially true for runtime verification, where the dynamic context can be used to make decisions about the program's behavior.

EXAMPLE 18 (STATIC AND DYNAMIC CONTEXT) In Example 1, we displayed the bytecode for a method `m` that creates a `List l` with an associated `Iterator i`. In Section 3.6, we also showed several ways to capture the method call that creates the iterator at Line 7 in Listing 2.1. Relevant information at this point of the execution

can be extracted from the static and dynamic contexts. For instance the method name `m`, the class name, and the line number 7 are all static attributes. The dynamic context includes the objects that variables `l` and `i` point to in memory at the time of the method call. For instance, Listing 3.4 shows in lines 5 and 6 how the `DynamicContext dc` object can be used to access the dynamic context; the variables `l` and `i` point to.

4.2.2 Join points

A *join point* is essentially a configuration of the base program traversed during its execution. We denote the set of all possible join points as *Joinpoints*. Join points represent specific points in the program where additional behavior can be added or existing behavior can be modified.

DEFINITION 15 (JOIN POINT) A *join point* represents a specific point in the control flow of a program where additional behavior can be inserted or existing behavior can be modified. A join point $j \in Joinpoints$ is represented as a tuple:

$$j = (C_S, C_D)$$

EXAMPLE 19 (JOIN POINTS) Consider the method `m` in Listing 2.1. The method call at Line 7 is a join point. Other join points could be the entry and exit points of the method. Join points can be captured at different levels of granularity. For instance, the execution of a single bytecode instruction can be a join point. It represents the smallest observable unit of execution in the program.

4.2.3 Advice

The *advice* encapsulates the additional behavior that is to be inserted into the base program at specific join points. The set of all possible advice functions is denoted as *AD*. Formally, an advice is the function adv defined as:

DEFINITION 16 (ADVICE FUNCTION) The advice function is defined as:

$$\text{adv} : Joinpoints \rightarrow Instrs$$

The advice function takes a join point and its associated static and dynamic contexts as inputs and returns a set of bytecode instructions that are to be inserted at that join point. These instructions can either modify existing behavior or introduce new functionality.

4.2.4 Shadows

Each join point has a corresponding part in the program code, which is called a *shadow*. Shadows are constructs used to mark the code regions in the base program. It is at these regions that transformations are applied. A shadow consists of a lexical element and a direction attribute. Lexical elements serve as the point of reference for transformations. They correspond to the actual bytecode instructions, basic blocks, or methods in the program. A lexical element *element* in the program is defined as:

$$element \in Instrs \cup Methods \cup Blocks$$

A direction attribute indicates the relative position of the transformation concerning the lexical element. Let *Direction* be a direction defined as:

$$Direction \in \{\text{before}, \text{after}, \text{enter}, \text{exit}\}$$

Shadows are pairs consisting of bytecode instructions along with a specified direction (before and after), or basic blocks and methods along with an enter or exit direction.

DEFINITION 17 (SHADOW) A shadow serves as a marker in the bytecode, indicating where a transformation can be applied. A shadow $s \in Shadows$ is defined as a tuple:

$$s = (Direction, element)$$

The set *Shadows* represents all the shadows that are identified in a program.

$$\begin{aligned} Shadows = \quad & (\{\text{before}, \text{after}\} \times Instrs) \\ \cup \ & (\{\text{enter}, \text{exit}\} \times Blocks) \\ \cup \ & (\{\text{enter}, \text{exit}\} \times Methods) \end{aligned}$$

For a method $m \in Methods$, $m.\text{blocks} \subseteq Blocks$ denote the basic blocks in CFG_m, and $m.\text{instrs} \subseteq Instrs$ denote the indexed list of instructions in m.

DEFINITION 18 (METHOD SHADOWS) $Shadows_m$ denotes the set of all shadows in method m.

$$\begin{aligned} Shadows_m = \quad & (\{\text{before}, \text{after}\} \times m.\text{instrs}) \\ \cup \ & (\{\text{enter}, \text{exit}\} \times m.\text{blocks}) \\ \cup \ & \{\langle \text{enter}, m\rangle, \langle \text{exit}, m\rangle\} \end{aligned}$$

The shadows of a method are restricted to its instructions and basic blocks. We give an example of the shadows in a method.

```
1  m() {
2      ⟨enter, m0⟩
3      ⟨enter, b0⟩
4      ⟨before, i0⟩
5      _new ArrayList
6      ⟨after, i0⟩
7      ⟨before, i1⟩
8      dup
9      invokespecial ArrayList.init ()V
10     astore 1
11     aload 1
12     ldc A
13     invoke List.add (Object;)Z
14     pop
15     aload 1
16     ⟨before, i9⟩
17     invokeinterface List.iterator ()Iterator;
18     ⟨after, i9⟩
19     astore 2
20     aload 2
21     invokeinterface Iterator.hasNext ()Z
22     ⟨exit, b0⟩
23     ifeq L0
24     ⟨enter, b1⟩
25     aload 2
26     invokeinterface Iterator.next ()Object;
27     pop
28     ⟨exit, b1⟩
29     L0
30     ⟨enter, b2⟩
31     getstatic System.out, PrintStream;
32     ldc done
33     invokevirtual PrintStream.print (String;)V
34     ⟨exit, b2⟩
35     ⟨exit, m0⟩
36     return
37 }
```

Listing 4.1: JVM Bytecode and associated shadows for the method in Listing 4.2.

EXAMPLE 20 (METHOD SHADOWS) Listing 4.2 contains a Java method `m` that creates a `List l` with an associated `Iterator i`. The method checks if `i.hasNext()` and calls `i.next()`. Listing 4.1 shows the (simplified) bytecode for method *m* in black font. We show some of the shadows highlighted in the colors blue, red, and olive green. We have two shadows for the method entry and exit points. Two shadows for each basic block (the if-statement results in having three basic blocks), and for each instruction, two shadows to delimit the region before it and after it (we omitted many of the instruction shadows for brevity). *

```
1    public void m() {
2       //Initialize a list of strings
3       List<String> l = new ArrayList<>();
4       l.add("A");
5       //Create iterator
6       Iterator<String> i = l.iterator();
7       //Call next if iterator has next
8       if (i.hasNext())
9          i.next();
10
11      System.out.print("done");
12    }
```

Listing 4.2: A method calling an Iterator.

Equivalence Between Shadows

A code region in the program can be targeted by transformations. Since shadows mark the bytecode regions, we define the notion of equivalence between shadows which allows us to detect transformations that target the same bytecode regions.

DEFINITION 19 (EQUIVALENCE RELATION OVER SHADOWS) The equivalence relation over shadows in a method m is denoted by $\Leftrightarrow_s^m$ and defined as follows:

$$\Leftrightarrow_s^m \subseteq Shadows_m \times Shadows_m$$

$$\begin{aligned}
&\stackrel{\text{def}}{=} \{ \langle \text{enter}, m\rangle, \langle \text{enter}, m.\text{entryBlock}\rangle \} && (1)\\
&\cup \{ \langle \text{exit}, b\rangle, \langle \text{exit}, m\rangle \mid b \in m.\text{exitBlocks} \} && (2)\\
&\cup \{ \langle \text{enter}, b\rangle, \langle \text{before}, i\rangle \mid b \in m.\text{blocks} \wedge i \in m.\text{instrs} \\
&\qquad \wedge\ i.\text{index} = b.\text{first.index}\} && (3)\\
&\cup \{ \langle \text{after}, i\rangle, \langle \text{exit}, b\rangle \mid b \in m.\text{blocks} \wedge i \in m.\text{instrs} \\
&\qquad \wedge\ i.\text{index} = b.\text{last.index} \} && (4)\\
&\cup \{ \langle \text{after}, i\rangle, \langle \text{before}, i'\rangle \mid i, i' \in m.\text{instrs} \\
&\qquad \wedge \exists k \in [1, \text{size}(m.\text{instrs})] : \\
&\qquad i.\text{index} = k\ \wedge\ i'.\text{index} = k+1 \} && (5)
\end{aligned}$$

In Definition 19, line (1) states that the region on method enter is equivalent to the region on basic block enter if the block is the entry block of the method. Line (2) states that the region at a method exit is equivalent to the region at the exits of all basic blocks that are exit blocks in the method. Line (3) states that the region at block entry is equivalent to the region before the first instruction in the block defined by b.first. Line (4) states that the region at the exit of a block is equivalent to the region after the last instruction of the block defined by b.last. Line (5) states that the region after an instruction and before its consecutive are equivalent. We omit for brevity that pairs in the relation are reflexive, symmetric, and transitive.

EXAMPLE 21 (EQUIVALENT SHADOWS IN A METHOD) Figure 4.1, depicts the CFG of a method m with 4 basic blocks (b_1, b_2, b_3, b_4) where b_1 is the entry block, b_2 and b_4 are both exit blocks. In basic block b_2, we show two consecutive instructions i and j. In basic block b_3, we show instruction k as the first instruction in the block and instruction l as the last instruction. The filled grey boxes in the figure illustrate the equivalent shadows,

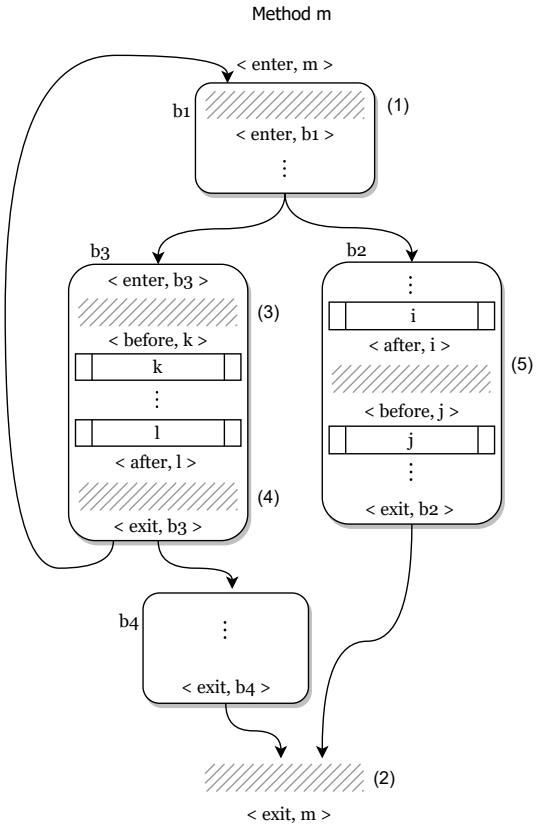


Figure 4.1: Illustration of shadows and their equivalence relation.

numbered as their corresponding line in Definition 19. From (1), we have $\langle \text{enter}, m\rangle \Leftrightarrow_s^m \langle \text{enter}, b_1\rangle$. From (2), we have $\langle \text{exit}, b_4\rangle \Leftrightarrow_s^m \langle \text{exit}, b_2\rangle \Leftrightarrow_s^m \langle \text{exit}, m\rangle$. From (3), we have $\langle \text{enter}, b_3\rangle \Leftrightarrow_s^m \langle \text{before}, k\rangle$. From (4), we have $\langle \text{after}, l\rangle \Leftrightarrow_s^m \langle \text{exit}, b\rangle$. From (5), we have $\langle \text{after}, i\rangle \Leftrightarrow_s^m \langle \text{before}, j\rangle$.

4.2.5 Selectors

The selector function takes a shadow, a static context C_S, and a dynamic context C_D as inputs and returns a set of join points where transformations can be applied.

DEFINITION 20 (SELECTOR FUNCTION) A selector $l \in \textit{Selectors}$ is a function defined as:

$$l : \textit{Shadows} \times C_S \times C_D \rightarrow 2^{\textit{Joinpoints}}$$

Selectors facilitate the selection of join points. They are enabled for well-defined regions in the bytecode. The associated compile-time analysis provides additional data for making more informed selections. Each selector is associated with a compile-time analysis function that takes static context as input and optional domain-specific data and returns static context that can be used for making more informed selections of join points.

DEFINITION 21 (COMPILE-TIME ANALYSIS FUNCTION) Each selector l is associated with a compile-time analysis function analyse, defined as:

$$\text{analyse} : C_S \times D_d \rightarrow C_S$$

where D_d is a domain-specific data structure that can be used in the analysis.

4.2.6 Instruction Visibility

Each instruction in the program is assigned a visibility status that determines whether the instruction is a candidate to be part of a shadow. This feature comes in handy later when we discuss the composition of transformers (Section 5.3.7) since it allows us to control the visibility of instructions in the base program.

DEFINITION 22 (INSTRUCTION-LEVEL VISIBILITY) Let visible : $Instrs \rightarrow \{\text{visible}, \text{invisible}\}$ be a function that maps each instruction to its visibility status.

An instruction with "visible" status is a candidate for transformation, while one with "invisible" status is ignored.

4.2.7 Transformers

Transformers are the building blocks of the instrumentation process. They encapsulate the logic for selecting join points and applying transformations. A transformer consists of a set of selector functions, a compile-time analysis, a set of advice functions, and a visibility function.

DEFINITION 23 (TRANSFORMER) A transformer $t \in Transformers$ is a tuple defined as:

$$t = (Selectors, \text{analyse}, AD, \text{visible})$$

The transformer function is responsible for selecting join points and applying transformations.

DEFINITION 24 (TRANSFORMER FUNCTION) The transformer function is defined as:

$$\text{transform} : Joinpoints \times C_S \times \text{visible} \rightarrow 2^{Instrs}$$

The transformer function focuses solely on the task of applying transformations to selected join points, thereby adhering to the principle of separation of concerns. This design choice allows for greater modularity, as each component—analysis, join point selection, and transformation—can be independently extended or modified. By performing the analysis as a preprocessing step, the model gains efficiency and context-awareness, enabling more precise transformations.

4.2.8 Instrumentation Process

The instrumentation process takes a program P, applies a composition of transformers, and produces a new program P'. The instrumentation process is defined as:

$$\text{instrument} : P \times Transformers \times Shadows \times C_S \times C_D \rightarrow P'$$

4.3 Transformer Composition

We allow more than one transformer in the instrumentation process. The transformers are applied sequentially to the base program in the order specified by the user. We refer to applying multiple transformers in a single run as *transformer composition*. In this section, we discuss the motivation for composing transformers (Section 4.3.1) and address some concerns that may arise when multiple transformers target the exact program bytecode regions (Section 4.3.3).

4.3.1 Motivations for Composition

Composition is needed in some cases and optional in others. Transformer composition is obligatory when it is impossible to merge the code of two transformers in one transformer. This situation arises when there is a dependency between transformers, and more than one pass is required to instrument the program. In other cases, we may want to implement separate transformers based on their functionality for a cleaner code. We discuss the two cases.

More than one pass. In many cases, transformations might require multiple passes on the same class. Since our model can be used to implement static analyzers, this enables plenty of scenarios where static analysis can be leveraged in combination with runtime verification. In such cases, a transformer can be implemented to perform the analysis before the transformer is responsible for instrumenting the code for monitoring. Another example is assuming that we are implementing a simple obfuscator that randomly changes the names of all methods in a program. In this case, one pass is not enough; we need one pass to map the original names to the obfuscated names and then another pass to change the classes and method names.

Modularity of transformers. At the core of aspect-oriented programming is achieving modularity to the cross-cutting concerns of an application. Hence we inspire and encourage separating transformers based on their functionality. This allows different team members to implement different transformers separately, where a single transformer should logically handle one concern. Let us say we want to instrument a program to monitor at runtime multiple safety properties. Implementing a single transformer for each property is more readable and favors reuse.

4.3.2 Composition of Transformers

The composition of transformers allows for chaining multiple transformations in a specific order, enabling complex bytecode modifications. The output of a composition is a transformer that encapsulates the combined behavior of all the individual transformers in the sequence.

DEFINITION 25 (COMPOSITION (c)) A composition $c \in \mathit{Composition}$ is a sequence of transformers $t_1, t_2, \ldots, t_n$ such that:

$$c : \mathit{Transformers} \times \mathit{Transformers} \times \ldots \times \mathit{Transformers} \rightarrow \mathit{Transformers}$$

4.3.3 Transformer Collision

Transformers impose new “aspects" into the base program by inserting advice. When two transformers insert advice that targets the same program bytecode regions, we say that the two transformers collide. BISM detects and reports transformer collisions, which makes the composition more transparent to the user.

BISM detects collision after weaving the advice of multiple transformers into the base program. Recall Definition18 of the shadows of a method. Let $\mathit{Shadows}^t_m$ represent all the shadows used by BISM to insert the advice for transformer t, in a method m, we have:

$$\mathit{Shadows}^t_m \subseteq \mathit{Shadows}_m$$

To detect collision in a method, we check whether two transformers insert advice at equivalent shadows (Section 4.2.4).

DEFINITION 26 (TRANSFORMER COLLISION) Transformer t collides with transformer t' in method m, iff

$$\exists\ s \in \mathit{Shadows}^t_m\ ,\ \exists\ s' \in \mathit{Shadows}^{t'}_m\ :\ s \Leftrightarrow^m_s s'$$

Notice that the collision between transformers is symmetric, which means that the order of applying two transformers is irrelevant to detect collision. Also, collision is reflexive, which means that collision is also detected when applying the same transformer twice.

Several concerns may arise from collisions, such as determining the order of execution and the visibility among aspects. We discuss these problems in the rest of this section.

4.3.4 Order Matters

The sequence in which transformations are applied can have a significant impact on the behavior of the final instrumented program. This is particularly true when multiple transformers target the same join points, leading to what we term as a *collision*.

DEFINITION 27 (ORDER-SENSITIVE TRANSFORMERS) Two transformers t_1 and t_2 are said to be order-sensitive at a join point j in method m if and only if:

$$t_1(j, SC, \text{visible}) \not\equiv t_2(j, SC, \text{visible})$$

Two transformers are considered equivalent at a join point if their transformations yield the same functional outcome, meaning that they produce identical changes in the program's control flow, state, and observable behavior. However, we will not delve into the full semantics of this equivalence relation, as it is beyond the scope of this thesis.

EXAMPLE 22 (ORDER MATTERS) Consider two hypothetical transformers: one for logging method entries and exits, denoted as t_{log}, and another for timing the execution of methods, denoted as t_{time}. Both transformers target the same join points, specifically the entry and exit points of methods. This results in a collision.

If t_{log} is applied before t_{time}, the timing data will include the time consumed by the logging operations. Conversely, if t_{time} is applied first, the timing data will only reflect the original method's execution time.

This example illustrates that t_{log} and t_{time} are order-sensitive transformers according to Definition 27. *

In some scenarios, the sequence of transformations may not significantly impact the final instrumented program. Therefore, it is crucial to understand the nature of each transformer and its interactions with others when determining the sequence of transformations.

4.4 Conclusion

In this chapter, we presented a formal underpinning for our instrumentation framework. We introduced the main concepts of the model, namely context objects, join points, shadows, selectors, advice, and transformers. Transformers can incorporate compile-time analysis, providing additional data for making more informed and guided instrumentation. We also discussed the composition of transformers and how they can be used to implement complex instrumentation scenarios in a modular fashion. Transformers in composition can control the visibility of program elements, allowing the transformers to filter out irrelevant elements based on some compile-time analysis. We also covered how to detect collisions between transformers.

CHAPTER 5

BISM: Bytecode Instrumentation for Software Monitoring

Contents

5.1 Introduction **55**
5.2 BISM in a Nutshell **55**
5.2.1 Overview 55
5.2.2 Design Goals and Features 56
5.3 BISM Instrumentation Language **57**
5.3.1 Selectors 57
5.3.2 Static Context 59
5.3.3 Dynamic Contexts 61
5.3.4 Advice Methods 61
5.3.5 Instrumentation Scoping 62
5.3.6 User Configuration 62
5.3.7 Transformer Composition 63
5.4 The External DSL for BISM **64**
5.4.1 Design Considerations 64
5.4.2 Pointcuts 64
5.4.3 Events 65
5.4.4 Monitors 65
5.4.5 Code Generation 66
5.5 Implementation **66**
5.5.1 The DSL 68
5.6 An Observation Layer for BISM **69**
5.7 Discussion **69**
5.8 Conclusion **71**

Chapter abstract

In this chapter, we present BISM (Bytecode-Level Instrumentation for Software Monitoring), a lightweight JVM bytecode instrumentation tool that features an expressive high-level control-flow-aware instrumentation language. The instrumentation language is inspired by the aspect-oriented programming paradigm in modularizing instrumentation however offering more flexibility in the instrumentation process. BISM allows capturing join points ranging from bytecode instructions to methods execution and provides comprehensive static and dynamic context information. However, it enables users to incorporate weave-time static analyzers that can be executed at the time of instrumentation along with advice code. Hence both the instrumentation and the analysis are specified at the same place using the same abstractions provided by the framework. We present the core API-based language of BISM and its design choices. We then present an external DSL that we developed to provide a more user-friendly interface for BISM targeted towards instrumentation for runtime verification. We then discuss its implementation detailing the instrumentation process before presenting an observation layer that can be used to observe the execution of the instrumented program.

5.1 Introduction

Motivation. In the previous chapters, we highlighted various challenges in runtime verification that can be tackled with a comprehensive instrumentation framework. Ensuring software reliability via runtime verification entails overcoming a variety of challenges, such as soundly capturing fine-grained events (C1), and the ability to incorporate weave-time analysis within the instrumentation process (C2).

Approach. We implement the instrumentation model we introduced in the previous chapter in the tool BISM (Bytecode-level Instrumentation for Software Monitoring) targeting JVM-based languages. We design BISM to provide the best of both worlds: the expressiveness of bytecode manipulation frameworks and the abstraction of aspect-oriented programming (AOP) frameworks. This is done by having a different instrumentation mechanism than the one used in AOP frameworks. To achieve this, writing instrumentation with BISM is achieved through *transformers*, which are classes that encapsulate join point selection and advice inlining. However, users specify advice using helper methods rather than directly writing code snippets. This approach enables users to write weave-time analyses that can be executed at the time of instrumentation along with advice code. Hence both the instrumentation and the analysis are performed at the same place using the same abstractions provided by the framework.

Contributions. BISM contributes to the field of runtime verification in several ways. First, it provides a dedicated, versatile, and expressive framework for collecting accurate traces (C1). Its design is such that it allows seamless compile-time pre-instrumentation analyses, thereby enabling more advanced monitoring techniques (C2).

- We present BISM, a dedicated bytecode instrumentation tool for runtime verification, that is both flexible and efficient.
- We balance expressiveness and abstraction in the BISM instrumentation language, inspired by AOP but enabling the capabilities of bytecode manipulation frameworks.
- We present an external DSL for BISM that allows for a more straightforward, declarative specification of instrumentation requirements.
- We demonstrate the architecture of BISM and its instrumentation workflow, along with the DSL implementation.
- We present an observation layer for BISM that allows for integrating various dynamic analyses with the instrumentation process.
- We discuss and compare the difference between BISM and other instrumentation frameworks that we previously presented in Chapter 3.

Chapter organization. Section 5.2 presents an overview of BISM, its design goals, and features. Section 5.3 introduces the language of BISM and the design choices taken. Section 5.4 presents the external DSL for BISM. Section 5.5 discusses the implementation of BISM. Section 5.6 presents an observation layer for BISM. Section 5.7 compares BISM to other instrumentation frameworks. Finally, Section 5.8 concludes the chapter.

5.2 BISM in a Nutshell

In this section, we overview BISM and discuss the design goals and its objectives.

5.2.1 Overview

BISM is a bytecode instrumentation tool for JVM languages such as Java, Scala, Kotlin, and Groovy programs. It is implemented on top of the bytecode manipulation library ASM [BLC02]. Figure 5.1 shows a high-level overview of BISM. The user provides: the base program, and the instrumentation logic written in transformers with BISM language (see Section 5.3). BISM encapsulates and performs 3 main steps that, in general, can describe any instrumentation task. It parses the bytecode to obtain a representation of the base program, then generates the needed transformations that are specified by the user in transformers, and finally weaves those transformations

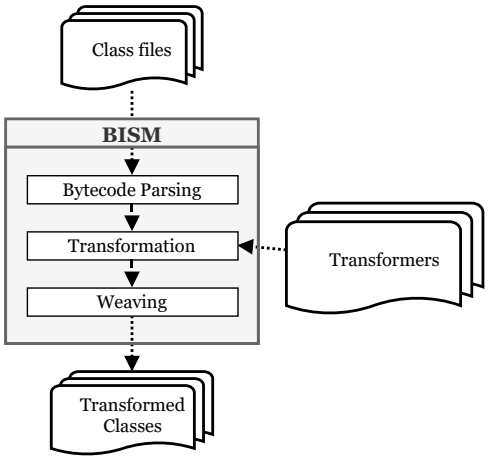


Figure 5.1: BISM overview.

into the base program to obtain an instrumented program. In Section 5.5, we provide more details of the BISM instrumentation workflow.

5.2.2 Design Goals and Features

BISM is a tool on which RV tools can rely to perform efficient and expressive instrumentation. In this section, we describe the design goals and features of BISM.

Instrumentation mechanism. BISM provides a mechanism to write separate instrumentation classes in standard Java. An instrumentation class in BISM, which we refer to as a transformer, encapsulates the instrumentation logic that is the join point selection and the advice to be injected into the base program. Advice is specified using dedicated *advice methods* provided by the BISM language that allows bytecode insertion, method invocation, and printing. Rather than directly writing code snippets, this approach enables users to write weave-time analyses that can be executed at the time of instrumentation along with advice code. Hence both the instrumentation and the analysis are performed at the same place using the same abstractions provided by the framework. Unlike aspect-oriented programming tools that adopt the pointcut/advice model where the user is only capable of specifying the code that should be injected, users can execute arbitrary code in the transformer class to perform compile-time analyses on the base program to guide the instrumentation.

Access to program context. BISM offers access to complete static information about instructions, basic blocks, methods, and classes. It also offers dynamic context objects that provide access to values that are only available at runtime, such as local variables, stack values, method arguments, and results. Moreover, BISM allows accessing instances and static fields of these objects. Furthermore, new local variables and arrays can be created within the scope of a method to pass values needed for instrumentation.

Control-flow context. BISM generates the CFGs of target methods out-of-the-box and offers this information to the user. In addition to basic block entry and exit selectors, BISM provides specific control-flow related selectors to capture conditional jump branches. Moreover, it provides a variety of control-flow properties within the static context objects. For example, it is possible to traverse the CFG of a method to retrieve the successors and the predecessors of basic blocks. Edges in CFGs are labeled to distinguish between the True and False branches of a conditional jump. Furthermore, BISM provides an option to display the CFGs of methods before and after instrumentation, which provides developers with visual assistance for analysis and insights on how to instrument the code and optimize it.

Compatibility with ASM. BISM uses ASM extensively and relays all its generated class representations within the static context objects. Furthermore, it allows inserting raw bytecode instructions by using the ASM data types. When inserting instructions, it is the user responsibility to write a code free from errors. If the user unintentionally

inserts faulty instructions, the instrumentation may fail. The ability to insert ASM instructions provides highly expressive instrumentation capabilities, especially when it comes to inlining the monitor code into the base program, but comes with the risk of producing unwanted behavior.

Instrumentation modes. BISM can run in two modes: *build-time* and *load-time*. In build-time, BISM acts as a standalone application capable of instrumenting all the compiled classes and methods[1]. In load-time, BISM acts as an agent (using JVM instrumentation capability[2]) that intercepts all classes loaded by the JVM and instruments before the linking phase. The load-time mode permits to instrument additional classes, including classes from the Java class library that are flagged as modifiable[3]. Instrumentation modes are complementary. BISM produces a new statically instrumented standalone program in build-time mode, whereas, in the load-time mode, BISM acts as an interface between the program and the JVM (keeping the base program unmodified).

Portability and ease of use. BISM is a lightweight tool written in Java and fitting in a single jar of less than 1MB. It is hardware-agnostic and only relies on the presence of a JVM in the host software. The user only needs to add the jar to the *classpath* (Java Runtime Environment variables) to compile new custom transformers. The tool has been successfully tested on various operating systems and even embedded devices such as the Raspberry Pi.

5.3 BISM Instrumentation Language

In this section, we present the core instrumentation language of BISM or what we refer to as the *BISM API*. Full details about the API can be found at [SFb]. The language allows the user to select *join points* (points in the program execution), retrieve relevant context information, and inject advice (i.e., extra code) that can extract information from these points or alter the behavior of the program.

Instrumentation in BISM is specified in Java classes named *transformers*. BISM language provides *selectors* (Section 5.3.1) to select join points of interest, static and dynamic context objects (Section 5.3.2 and Section 5.3.3) which retrieve relevant information from these points, and *advice* methods (Section 5.3.4) to specify the code to be injected into the base program.

5.3.1 Selectors

Selectors provide a mechanism to select join points and specify the advice. They are implementable methods where the user writes the instrumentation logic. BISM provides a fixed set of selectors classified into four categories: instruction, basic block, method, and meta selectors. We list below the set of available selectors and specify the execution they capture.

Instruction. BISM provides instruction-related selectors:

- `BeforeInstruction` captures the execution before a bytecode instruction.
- `AfterInstruction` captures the execution after a bytecode instruction. If the instruction is the exit point, i.e. the last instruction of a basic block, it behaves the same way as the `BeforeInstruction` selector; that is it captures the execution before that last instruction.
- `BeforeMethodCall` captures the execution just before a method call instruction and after loading all needed values on the stack.
- `AfterMethodCall` captures the execution immediately after a method call instruction and before storing the return value of the method call, if any, from the stack into a variable.

Basic block. In addition to the previous selectors, BISM provides basic block-related selectors that ease capturing control-flow related execution points:

[1] Excluding the native and abstract methods, as they do not have bytecode representation.
[2] The `java.lang.instrument` package.
[3] The modifiable flag keeps certain core classes outside the scope of instrumentation. To the best of our knowledge, there is no exhaustive list of classes with the before-mentioned flag.

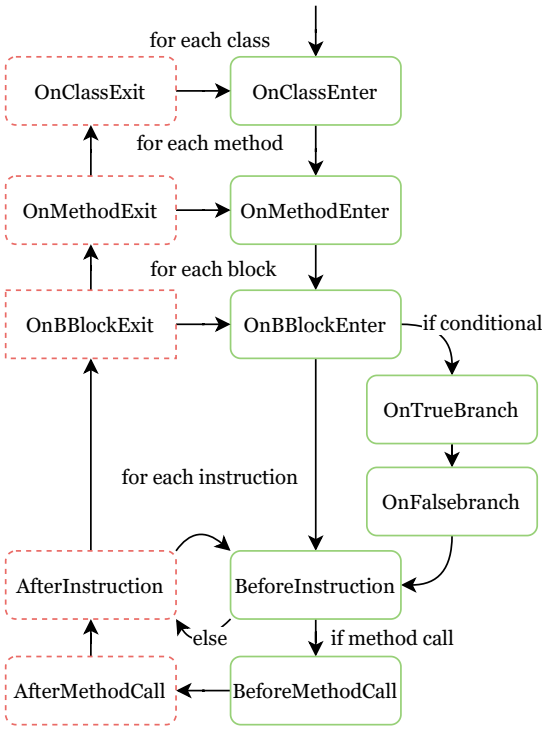


Figure 5.2: Instrumentation loop of BISM.

- `OnBasicBlockEnter` captures the execution when entering the block, before the first real instruction[4].
- `OnBasicBlockExit` captures the execution after the last instruction of a basic block; except when the last instruction is a JUMP/RETURN/THROW instruction, then it captures the execution before that last instruction.
- `OnTrueBranchEnter` captures the execution on the entry of a successor block after a conditional jump on True evaluation.
- `OnFalseBranchEnter` captures the execution on the entry of a successor block after a conditional jump on False evaluation.

Method. BISM also provides two method-related selectors:

- `OnMethodEnter` captures the execution on a method entry.
- `OnMethodExit` captures the execution on all exit blocks of a method before the return instruction.

Meta selectors. Finally, BISM provides two class related meta-selectors: `OnClassEnter` and `OnClassExit`. These selectors do not capture execution points but can be used for introductions, such as adding new members to a class. They have no semantic for the execution but are instead related to BISM execution. Selector `OnClassEnter` is invoked when a class is *loaded* and `OnClassExit` after all methods have been instrumented. They are not related to the `static {...}` block which is a function in Java classes. They can also be used to optionally initialize and finalize the transformer execution for each instrumented class.

The order at which selectors are visited when applying a transformer is depicted in Figure 5.2. Knowing this traversal flow helps the developer know in which order the advice weaving happens.

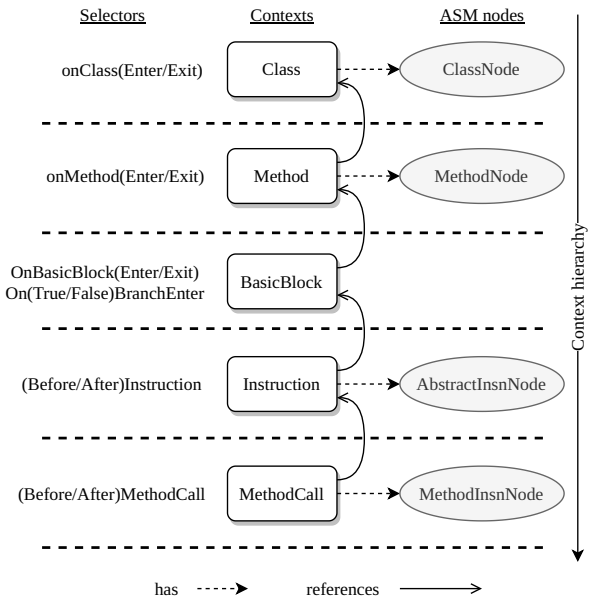


Figure 5.3: The static context tree related to selectors and ASM nodes.

5.3.2 Static Context

Static context objects provide access to relevant static information for captured join points in selectors. Each selector has a specific static context object based on its category. These objects can be used to retrieve information about bytecode instructions, method calls, basic blocks, methods, and classes. BISM performs static analysis on the base program and provides additional control-flow-related static information such as basic block successors and predecessors. The rich set of context information allows the user to have an expressive join point selection mechanism from within selectors. Unlike AspectJ, BISM does not offer yet regular expressions to select join points, but from the context objects, one can retrieve the method signature and therefore make the selection manually. It is even possible to be more selective as BISM offers directly the static context of each bytecode instruction, which is not accessible in AspectJ. The static context hierarchy and accessibility are summarized in Figure 5.3. Each context provides access to the corresponding ASM node object and is accessible from the corresponding selector or by traversing the hierarchy bottom-up as demonstrated in Listing 5.1.

```
1 public void beforeMethodCall(MethodCall mc, ...){
2   Instruction i = mc.ins; //Access the surrounding Instruction static context
3   BasicBlock b = i.basicBlock; //Access the surrounding BasicBlock context
4   Method m = b.method; //Access the surrounding Method context
5   ClassContext c = m.classContext; //Access the surrounding Class context
6 }
```

Listing 5.1: Hierarchy of Static context objects.

In the following, we detail commonly used properties for each context.

Common properties. At each selector, we need to identify the currently instrumented object and its location. Static context objects contain some identifiers:

- a reference to its parent context, named after the parent type;
- multiple string identifiers for class and methods, such as name and signature;
- a unique (method-wise) integer identifier for basic blocks and instructions.

[4]Real instructions are instructions that actually get executed, as opposed to some special Java bytecode instructions such as labels or line number instructions.

Instruction context. The `Instruction` context provides relevant information about a single instruction:

- `opcode`: its opcode in the JVM instruction set;
- `next/previous`: neighbor instructions in the current basic block if they exist;
- `isBranchingInstruction()`: indicator of whether it is a branching instruction (multiple successors). BISM takes care of comparing with the right opcodes and ASM node types.

It is also possible to retrieve information about the stack context at an instruction, as the information is embedded into the class file :

- `getBasicValueFrame()`: returns a list of all local variables, stack items, and their types at the stack frame before executing the current instruction;
- `getSourceValueFrame()`: returns a list of all local variables and stack items and their source i.e. which instruction created/manipulated them.

Method Call context. This is a special type of `Instruction` context (only available in `*MethodCall` selectors). In addition to its `Instruction` context, some specific information is provided, such as the caller method name or the called method class and name.

Basic Block context. The `BasicBlock` context provides information about a basic block and its neighborhood in the CFG:

- `blockType`: a type to easily identify its role (entry, exit, conditional, or normal block);
- `getSuccessor/PredecessorBlocks()`: all successors and predecessors of the basic block as per the CFG;
- `getTrue/FalseBranch()`: the target block after this conditional block evaluates to true (false);
- `getFirst/LastRealInstruction()`: the first and last executable instructions (not labels) of this block.

Method context. The `Method` context provides information about the currently visited method:

- `name`: the name of the method (not fully qualified);
- `getEntryBlock/getExitBlocks()`: the first and last blocks of the method;
- `isAnnotated(String)`: checks for the existence of an annotation on the method;
- some signature information about the method such as its return type and list of formal arguments.

Class context. The `Class` context provides the name and ASM node of the currently instrumented class.

```
1  public class BasicBlockTransformer extends Transformer {
2
3    @Override
4    public void onBasicBlockEnter(BasicBlock bb){
5      String blockId = bb.method.className + "." + bb.method.name + "." + bb.id;
6      print("Entered block:" + blockId)
7    }
8
9    @Override
10   public void onBasicBlockExit(BasicBlock bb){
11     String blockId = bb.method.className + "." + bb.method.name + "." + bb.id;
12     print("Exited block:" + blockId)
13   }
14 }
```

Listing 5.2: A transformer for intercepting basic block executions.

In Listing 5.2, the transformer uses two selectors to intercept all basic block executions (`onBasicBlockEnter` and `onBasicBlockExit`). `BasicBlock bb` is used to get the block id, the method name, and the class

name. The advice method `print` inserts a print invocation in the base program before and after every basic block execution.

5.3.3 Dynamic Contexts

BISM also provides dynamic context objects at selectors to extract join point dynamic information. These objects can access dynamic values from captured join points that are possibly only known during the base program execution. BISM gathers this information from local variables and operand stack, then weaves the necessary code to extract this information. In some cases (e.g., when accessing stack values), BISM might instrument additional local variables to store them for later use. For brevity, we list some of the dynamic context methods and omit the return type of the methods which is always a `DynamicValue`:

- `getThis()`: returns a reference to the class owner of the method being instrumented, and null if the class or method is static;
- `getThreadName()`: returns a reference to the name of the thread executing the method being instrumented;
- `getLocalVariable(int)`: returns a reference to a local variable by index;
- `getStackValue(int)`: returns a reference to a value on the stack by index;
- `getStatic/InstanceField(String)`: returns a reference to an instance/static field in the class being instrumented.

BISM gives access to method-relative information. The runtime arguments passed to a method can be retrieved using `getMethodArgs(int)`. The method result (return value) can be retrieved using `getMethodResult()`. The object on which the method is called can be retrieved using `getMethodReceiver()`.

It is also possible to add new local variables of primitive types with a call to `addLocalVariable(...)`. The scope of the added variables is the method where they are created. This is useful for different purposes like to pass data across selectors. Local arrays can also be added with the `createLocalArray(Method, Class)` method. BISM weaves the necessary bytecode and returns a dynamic value to query and update freely, such as clearing and appending elements. It is particularly useful when there is a need to pass objects between selectors in a method without knowing how much space will be needed at runtime.

Listing 5.3 presents a transformer that uses the selector `afterMethodCall` to capture the return of an `Iterator` created from a `List` object. It uses the dynamic context object using `MethodCallDynamicContext dc` provided to the selector to retrieve the dynamic data. The example also shows how to limit the scope to a specific method using an if-statement on the static context.

```
1  @Override
2  public void afterMethodCall(MethodCall mc, MethodCallDynamicContext dc){
3    if (mc.methodName.equals("iterator") && mc.methodOwner.endsWith("List")) {
4      //Access to dynamic data
5      DynamicValue callingClass = dc.getThis(mc);
6      DynamicValue list = dc.getMethodTarget(mc);
7      DynamicValue iterator = dc.getMethodResult(mc);
8
9      //Invoking a monitor
10     StaticInvocation sti = new StaticInvocation("IteratorMonitor", "iteratorCreation");
11     sti.addParameter(callingClass);
12     sti.addParameter(list);
13     sti.addParameter(iterator);
14     invoke(sti);
15   }
16 }
```

Listing 5.3: A transformer that intercepts the creation of an iterator from a `List`.

5.3.4 Advice Methods

A user inserts advice into the base program at the captured join points using the advice instrumentation methods. Advice methods allow the user to extract needed static and dynamic information from within join points, also allowing arbitrary bytecode insertion. These methods are invoked within selectors. BISM provides `print` methods

with multiple options to invoke a print command. It also provides (i) `invoke` methods for static method invocation; (ii) `annotate` methods for adding annotations to class, methods or fields and (iii) `insert` methods for inserting bytecode instructions. These methods are compiled by BISM into bytecode instructions and inlined at the referenced bytecode location. We list below the advice methods available in BISM.

Printing on the console. Instrumenting print statements in the base program can be achieved via method `print`, which permits to write both on the standard and error output of the base program. These methods take either static values or dynamic values retrieved in selectors. Listing 5.2 shows an example of using one of the print helper methods to instrument the base program to print the basic block constructed id.

Invoking methods. Invoking external static methods can be achieved using the advice method `invoke`. An object of type `StaticInvocation` or `MethodInvocation` should be constructed and provided with the external class name, the method name, and parameters. Listing 5.3 depicts a transformer that instruments the base program to call the external static method `iteratorCreation`. The constructor of the invocation takes the class and method names as input in case of static methods. In the case of instance methods, the constructor takes as dynamic value the object on which the method is called. Parameters can be added using `addParameter()`. It supports either `DynamicValue` type or any primitive type in Java, including `String` type (any other type will be ignored). After that, `invoke` weaves the method call in the base program.

Annotating the bytecode. Annotating elements of a class file allows the user to add some meta-data to the code. It is possible to use the BISM advice `annotate(...)` to add *runtime-visible* annotations to either class, fields, or methods. These annotations can be accessed via Java reflection, and they provide extra information for the input program or some third-party API.

Raw bytecode instruction insertion. Inserting raw bytecode instructions can be achieved with `insert` methods. When used, it is the developer's responsibility to write correct instructions that respect the JVM static and structural constraints. Errors can be introduced by ignoring the stack requirements and altering local variables. For Java 8 and above programs, using the insert methods to push new values on the stack or create local variables requires modifying the *maxStack* and *maxLocals* values. All static contexts give access to the needed ASM object `MethodNode` to increment the values maxLocals and maxStack from within the join point.

5.3.5 Instrumentation Scoping

BISM provides many configuration features, such as limiting the scope of the instrumentation or passing arguments to the transformers to modify their behavior. For example, the *scope* global argument permits matching classes and methods by their names.

Specifying *(scope=java.util.List.*, java.util.Iterator.next)* will instrument all methods in the `List` class and only the next method in the `Iterator` class. Moreover, static context objects can also be used to limit the scope of instrumentation from inside selectors; they can provide more precise scoping information demonstrated in Listing 5.3. It is recommended using the *scope* argument when possible to avoid analyzing unwanted classes enhancing instrumentation performance.

5.3.6 User Configuration

To favor usability, BISM execution accepts arguments both from the command line (which has higher priority) or through a configuration file. Configurable settings such as printing the CFG files and dumping the instrumented bytecode can be specified. The configuration file is more expressive as it also permits passing arguments to transformers. A transformer may need arguments to modify its internal behavior, e.g., a flag for logging.

EXAMPLE 23 (BISM CONFIGURATION FILE) Listing 5.4 shows an example of a BISM configuration file. The `config` tag contains the global configuration arguments. The `transformer` tag contains the list of transformers to be executed. The `arg` tag is used to pass arguments to transformers. The `scope` argument specifies the scope of the instrumentation. The `blacklist` argument specifies the classes to be excluded from the instrumentation. The `output` argument specifies the output directory for the instrumented classes. The `visualize` argument specifies that the CFGs of the instrumented methods should be dumped. *

```
<bism>
    <config>
        <transformer value="inst/IndirectCoverageTransformer.class,inst/OnExit.class"/>
        <scope value="com.*"/>
        <blacklist value="com.sun.*"/>
        <output value="./out/instrumentation"/>
        <visualize/>
    </config>

    <transformer></transformer>
    <transformer>
        <arg key="owners" value="com/Main"/>
        <arg key="methods" value="main"/>
    </transformer>
</bism>

```

Listing 5.4: BISM XML configuration file.

5.3.7 Transformer Composition

We here detail some features of BISM that ease the composition of transformers.

Collision report.

To detect a collision (Section 4.3.3), we compute equivalent shadows used for instrumentation by transformers. BISM records the used shadows after weaving each transformer and reports all collisions after each run. The report shows the exact locations of the collision along with the colliding transformers to the user.

Several concerns may arise from collisions, such as determining the order of execution and the visibility among aspects. We discuss these problems in the rest of this section.

Controlling Visibility

When composing transformers, each transformer introduces a new set of instructions to the base program. The newly added instructions are then part of the base program and become visible to the second transformer to target. In many cases, we may want to hide these newly instrumented instructions in a composition. BISM provides the attribute `@Hidden` that can be placed as an annotation on a transformer. When used, the newly added instructions are not intercepted anymore by the selectors by following transformers. A prevalent scenario is to avoid instrumenting previously added code. Let us look at the following example.

```
@Hidden
class CountMethodCalls extends Transformer {
   onMethodCall(..) {
      //Invoke methodCounterIncrement
   }
}
@Hidden
class LogMethodCalls extends Transformer {
   onMethodCall(..) {
      //Invoke logger
   }
}
```

Listing 5.5: `@Hidden` attribute on transformers.

EXAMPLE 24 (HIDDEN TRANSFORMERS) Listing 5.5 demonstrates two transformers: `CountMethodCalls` which counts the number of method calls and `LogMethodCalls` which logs all method calls. We can see that the join points captured by `onMethodCall` are shared between both transformers. Hence, the advice will be inserted at the same bytecode regions, and we have a *collision*. A user might be only interested in logging the method calls of the base program and does not want the logger to log counting calls introduced by `LogMethodCalls`. Alternatively, the user might be interested in counting the method calls of the base program and not the log calls introduced by `CountMethodCalls`. Adding the `@Hidden` attribute to the transformers hides the newly added instructions from other transformers in a composition.

Hidden instructions. BISM also allows transformers to hide arbitrary instructions of the base program from other transformers by providing a mechanism to mark instructions as hidden. When an instruction is marked as hidden, it is excluded from the available shadows and thus not exposed to selectors. Hence, it will not be intercepted by the transformers that follow. This feature can be used for optimizing instrumentation by having one transformer implement a static analyzer that hides particular instructions from the instrumentation transformer.

In general, to avoid instrumenting previously added advice, the user is encouraged to check the collision report and use the `@Hidden` when needed.

5.4 The External DSL for BISM

In order to facilitate the use of BISM, we designed a new external DSL for BISM. This DSL is a textual specification language that allows users to specify instrumentation logic in a more intuitive and concise manner. Targeting a wider range of users, the DSL is designed with intuitive syntax, maps directly to the key requirements of program instrumentation, and offers comprehensive coverage of program aspects while reducing specification complexity.

5.4.1 Design Considerations

In this section, we discuss the main design considerations for the new BISM DSL.

Intuitive syntax. Instrumenting a program generally involves specifying three key requirements: (1) program points to capture (*join points*), (2) the information needed from these join points, and (3) the destination of these events. The DSL addresses these requirements by providing exactly these three main constructs: (1) *pointcuts*, (2) *events*, and (3) *monitors* to consume the captured events.

Expressiveness & abstraction. BISM API offers remarkable expressiveness for covering all program aspects and extracting information from the executing program. We designed the DSL to retain this expressiveness while considerably simplifying the specification and providing a higher abstraction level. However, the DSL's high-level abstraction may limit its ability to address certain low-level details or complex scenarios that require more fine-grained control over the instrumentation process.

Efficiency and performance. Textual specifications for instrumentation often require parsing and compiling transformations. We optimized the DSL implementation with a focus on performance, striving to achieve reasonable execution times for parsing and applying instrumentation compared to using the native BISM API.

Usability simplification. Crafting a BISM transformer requires implementing a Java class using the provided instrumentation API. The DSL is tailored to accommodate users with diverse expertise levels by eliminating boilerplate code and low-level implementation details, ultimately enhancing productivity, minimizing errors, and simplifying code maintenance.

Code Generation. The DSL creates API-based transformer code from user-defined textual specifications, allowing further user customization for complex instrumentation. It also generates monitor interfaces, promoting collaboration between developers and monitoring experts.

In the next section, we present the main abstractions and constructs provided by the DSL to address the user requirements of runtime verification and instrumentation.

5.4.2 Pointcuts

Pointcuts enable users to specify the join points to capture from program execution. They can be denoted as follows: a pointcut *name*, a BISM selector with a *pattern*, and an optional *guard*. Multiple selectors are chainable using the `||` operator. The pattern restricts the scope of the selector to specific fields or methods, applying filters such as types, method signatures, or field names. Matching can be achieved with wildcards; for example, "`*.set*(..)`" matches any method that starts with "set". The optional guard allows the specification of a condition, essentially a boolean expression using static context objects from the join point. The guard conditions may use

comparisons of booleans, numerics, and strings, and can be chained with the conjunction operator (&&) to create complex conditions. In each DSL transformer, the definition of at least one pointcut is necessary.

```
pointcut pc1 after MethodCall(* BankAccount.*(..)) with (getNumberOfArgs = 3 &&
    currentClassName = "Main")
          || after MethodCall(* *.*(..)) with (instruction.linenumber = 42)

pointcut pc2 before Instruction(* *.*(..))
                    with (isConditionalJump = true)
```

Listing 5.6: Example of pointcuts definition.

Listing 5.6 shows a composite pointcut `pc1` that uses two selectors. The first selector captures calls to methods defined by the class`BankAccount`, but only captures calls that are invoked from the `"Main"` class and the called method has exactly three arguments. The second selector captures any method call occurring at line `42`. This second guard showcases BISM's hierarchical context objects and how they can be accessed using dot notation. Pointcut `pc2` captures any instruction that is a conditional jump.

5.4.3 Events

Events encapsulate the information that needs to be extracted from a pointcut. Each event must be associated with a single pointcut along with its arguments. Multiple events can be defined for each pointcut. An event includes a name and zero or more arguments. Arguments may comprise single values or lists of values, which can include BISM static or dynamic context objects, string literals, numbers, or lists. Lists are denoted as sequences of values, separated by commas, and enclosed in brackets.

```
event e1("call", [getMethodReceiver, getMethodResult]) on pc1

event e2([opcode,getStackValues]) on pc2 to console(List)
```

Listing 5.7: Example of events definition.

Listing 5.7 shows an event named `e1`, associated with the pointcut `pc1`, that is defined with the string literal `"call"` and a list of dynamic context objects which extract the callee object and the result of the method. Event `e2` is defined with a list of dynamic context objects (`opcode, getStackValues`) and is associated with the pointcut `pc2`. The DSL also provides a construct to print an event to the console, which is particularly useful during the debugging or profiling of a program. This event is associated with the output `console(List)` which prints the event information to the console.

5.4.4 Monitors

Monitors define the extraction points of one or several events during program execution. Each monitor is identified by a unique name, a class name, and the events it is set to listen to. Typically, the events are passed as parameters during the invocation of a monitor method. A monitor can be defined as follows: a monitor name, a class name, and the events it listens to. Events are mapped to the monitor method name with its argument types. Multiple monitors can be defined and events can be sent to more than one monitor.

Listing 5.8 shows a monitor `m1` corresponding to a class named `com.MonitorX`. Event `e1` is mapped to the method `receive` with the argument types `String` and `List`. The specification can be simplified by directly associating the monitor with the event. However, this restricts the use of the event to only one monitor. Here is an equivalent specification without the explicit definition of a monitor.

```
monitor m1{
    class: com.MonitorX,
    events: [e1 to receive(String, List)]
}
```

Listing 5.8: Example of a monitor definition.

```
event e1("create",[getMethodReceiver,getMethodResult])
                    on pc1 to com.MonitorX.receive(String,List)
```

Listing 5.9: Simple monitor definition.

5.4.5 Code Generation

We here present the code generation facilities that bridge the gap between new and expert users of BISM, also the gap between monitoring experts and program developers.

Transformers for Complex Instrumentation Tasks

The Java-based API transformers enable users to develop intricate analysis logic beyond the capabilities of the DSL. For example, BISM utilized this approach for static analysis prior to program instrumentation in [SF22b]. Starting with a simple text-based DSL specification, users can gradually move to complex instrumentation logic using a full-fledged Java transformer. To facilitate this transition, we generate and compile Java transformers equivalent to the provided textual specification files. Consequently, these specification files can act as bootstrappers for implementing more complex logic.

Monitor Interfaces

As discussed in Section 5.4.4, the process of creating an instrumentation specification often requires declaring a monitor to listen for events. In collaborative scenarios, a monitoring expert consults the developer to identify relevant events and their runtime context. Subsequently, the programmer creates a specification file capable of locating and extracting the necessary events, leaving the monitoring expert solely responsible for implementing the monitors. To optimize this collaboration, the DSL incorporates a feature to generate the monitor interfaces, that need to be implemented to listen to events. This capability substantially improves interoperability between different teams, fostering a more efficient and productive workflow when addressing complex instrumentation tasks.

```
package com;
import java.util.List;

public class MonitorX{
    public static void receive(String a1, List a2) {
    }
}
```

Listing 5.10: A program with a test inversion attack.

Listing 5.10 shows the generated interface for the monitor in Listing 5.8. The user can then implement the `receive` method to perform the desired analysis logic.

5.5 Implementation

In this section, we provide some details about the BISM implementation. BISM is implemented in Java using about 7,000 LOC and 55 classes distributed in separate modules that interact and achieve its functionality. Below are the main modules of BISM depicted in Figure 5.4 and their responsibilities.

- **Core.** This module handles the user input and the output. It is responsible for orchestrating the instrumentation process, which we will detail in the remainder of this section.
- **CFG.** This module generates and stores the control flow graphs for the methods of the target program. It also contains the basic data structures that represent and iterate methods in classes that are built on top of ASM data structures.
- **Transformers.** This module exposes BISM instrumentation language for writing transformers. It also handles the iteration of program classes for the generation of join points, static and dynamic context, and applying the `Transformer` classes.

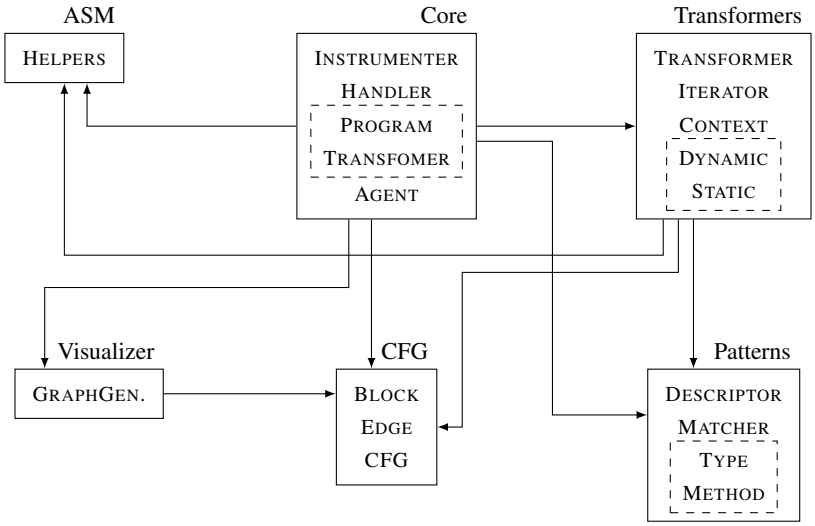


Figure 5.4: BISM modules with arrows indicating dependencies.

- **GraphVisualizer.** This module is responsible for generating CFG visualization. It uses the CFG module to generate the CFG of the input program before and after instrumentation, and it outputs HTML files that allow us to display the generated CFGs.
- **Patterns.** This module provides the pattern-matching logic for types, methods, and their signatures.
- **ASM.** This module provides the ASM library and its API to parse and generate bytecode.

BISM uses ASM under the hood for bytecode parsing, analysis, and weaving. It is provided as a runnable JAR file that does not require any installation from the user except for having Java 8 or above installed. It can run in two modes: build-time mode, as a standalone application to statically instrument a program, and load-time mode, attached to a program as a Java agent. Fig. 5.5 shows a detailed view of the internal workflow.

(1) User Input. In build-time mode, arguments consist of a base program bytecode (*.class* or *.jar*) to be instrumented and a list of transformers that specifies the instrumentation logic. In load-time mode, only the transformers are passed as arguments and all classes loaded by the JVM are instrumented. BISM provides several built-in transformers that can be directly used. Moreover, users can specify various runtime arguments to BISM or even the transformers, from the console or through a configuration file.

(2) Parse Bytecode. For each class in the base program, BISM uses ASM to parse the bytecode and generate a tree object containing all the class details, such as fields, methods, and instructions. The following three steps will be performed on each class for every transformer specified in a run.

(3) Build CFG. BISM constructs the CFGs for all methods in the target class. If the transformer utilizes control-flow join points (`onTrueBranch` and `onFalseBranch`), BISM eliminates all *critical edges* from the CFGs to avoid instrumentation errors. This is done by inserting empty basic blocks in the middle of critical edges, which is only applied if used while keeping copies of the original CFGs. Also, if the transformer uses join point `onMethodExit`, all the exit blocks (which terminate with a return opcode) are merged into a single one to avoid duplication and errors. This is done by adding a new block that contains a return of a suitable type; then, all other returns are replaced by unconditional jumps to the added one. Moreover, if the users opted for the *visualizer*, the CFGs are printed into HTML files on the disk.

(4) Generate Shadows and Context Objects. BISM iterates over the target classes to identify all shadows utilizing the created CFGs. The relevant static and dynamic context objects are created and initialized using the static information available and BISM analysis at each shadow.

(5) Transformer Weaving. The transformer is notified of each shadow and passed the static and dynamic objects. The weaving loop is illustrated in Figure 5.2. BISM evaluates the transformations applied by a transformer using the advice methods. After that, it accordingly weaves the necessary bytecode instructions into the target class.

(6) Output. The instrumented bytecode is then output back as a *.class* file in build-time mode or passed as raw

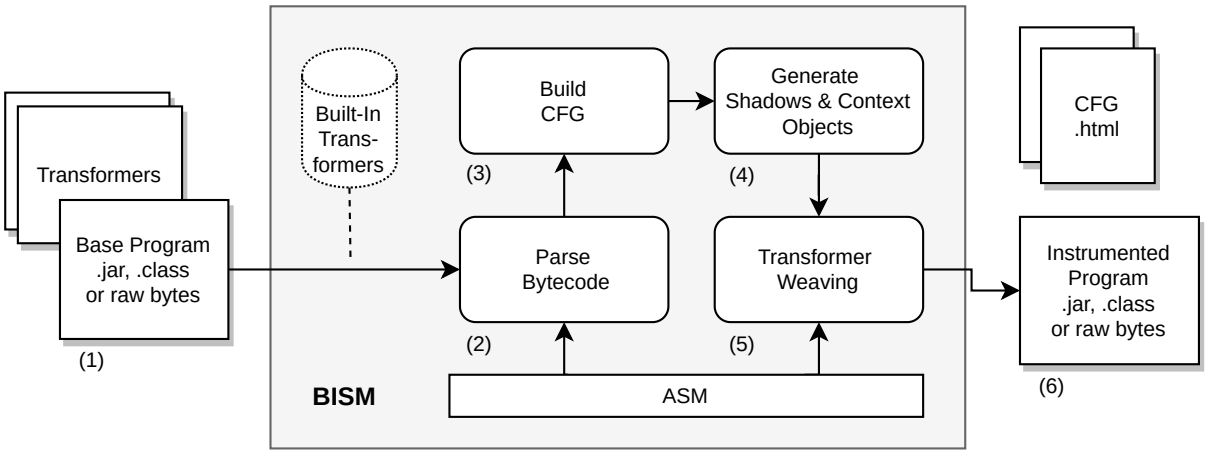


Figure 5.5: Instrumentation process in BISM.

bytes to the JVM in load-time mode. In case of instrumentation errors, e.g., due to adding manual faulty ASM instructions by the user, a weaving error is emitted. If the *visualizer* is enabled, the instrumented CFGs are also printed into HTML files on the disk.

5.5.1 The DSL

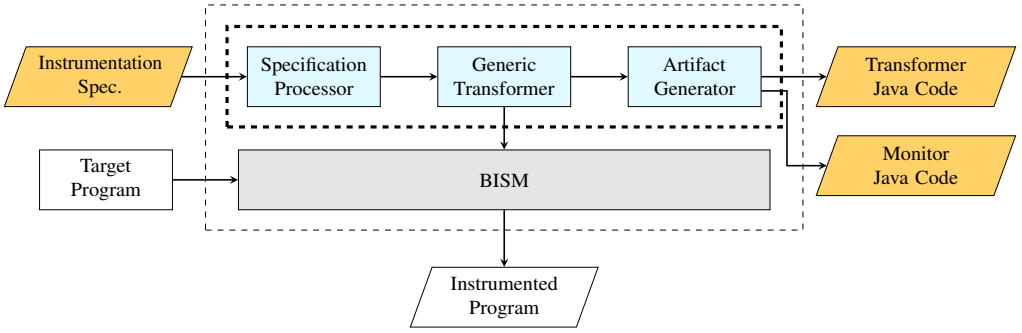


Figure 5.6: Added modules on BISM (in blue) to support the external DSL.

We implemented the DSL as an extension to BISM with 6 KLOC of Java code [5]. Figure 5.6 shows the three main modules we added to BISM. The following sections describe each module in detail.

Specification Processormodule. This module takes the textual specification written by the user as input and parses it to construct a specification object which is an intermediate representation of the needed transformations. It also performs checks to ensure that the specification is valid such as removing duplicate rules and ensuring that the specification is well-formed.

Generic Transformermodule. This module handles applying the transformations specified by the user to the target program. It provides a transformer template that extends the `Transformer` type provided by BISM and automatically generates the transformations. These transformations are then turned into bytecode instructions by BISM that are then weaved into the target program.

Artifact Generatormodule. This module is activated whenever the user requests code generation facilities. It is responsible for generating an equivalent transformer class that uses BISM API. It also generates a monitor class that can be implemented and used to monitor the instrumented program's behavior during runtime.

[5] Available at https://gitlab.inria.fr/bism/bism-dsl.

5.6 An Observation Layer for BISM

In this section, we present an observation layer that bridges BISM and the instrumented program with any analysis tool. This layer allows for incorporating various kinds of analyses into the instrumented program such as monitoring.

Users specify the instrumentation requirements using the BISM language and the analysis using their implementation of choice. We implemented an integration layer that bridges the communication between the running instrumented program and the analysis logic. This layer orchestrates a flexible, scalable pipeline whereby the program is the producer of events that are then dispatched for analysis. It is composed of 4 main modules that work together to receive, collect, and send events. Figure 5.7 shows the integration layer and its modules along with their dependencies. A dotted edge in the graph indicates that the source module uses the destination module. The base program is first passed to BISM for instrumentation along with `Transformer` files that specify the instrumentation requirements (Step 1). BISM analyses the program and injects instrumentation code into the program (Step 2). When the instrumented program is run, events are emitted and passed to the `Observer` module (Step 3).

The `Observer` module routes events into the `Event` module where custom events can be defined and created (Step 4). In its most generic form, an event consists of a map of key-value pairs. Next, the `Observation` module, receives the events (Step 5). An observation can be specified to operate in either *synchronous* or *asynchronous* modes, depending on the requirements of the user. In synchronous mode, the events are pushed immediately to the `Event Processor`. To ensure thread safety, event reception is synchronized using a lock. When operating in asynchronous mode, events are placed in a non-blocking queue. A separate process running on the distinct thread then removes the events from the queue. It creates a fork and delivers events simultaneously to subscribed event processors. The fourth module, the `Event Processor`, provides an interface for observations to publish events (Step 6). The module is also responsible for initializing the monitor specified by the user. Several processors can be created to handle events, subscribing to events that the observer publishes. Event processors can subscribe to specific events with the `subscribe` method which is `true` by default.

5.7 Discussion

In this section, we compare BISM with other instrumentation tools that we covered in Section 3.6 and discuss the differences and similarities. Table 5.1 summarizes this comparison and separates between features that concern expressiveness (in red) and features that concern abstraction (in blue).

Comparison with bytecode manipulation libraries. BISM is implemented on top of ASM to provide a superset of its features and offer the user a higher level of abstraction. The choice of ASM over alternatives like Soot for BISM was influenced by ASM's faster performance and smaller size. Essential tasks that need to be handled by the user with bytecode manipulation libraries such as program traversal, are already provided by BISM. Moreover, complex tasks such as accessing dynamic contexts and extracting their values from a running program require analyzing the code and adding bytecode instructions such as stack duplication and local variables assignments at the correct locations. These tasks are handled by BISM, allowing the user to retrieve dynamic context via its dynamic context objects. The same reasoning applies to the advice methods where the user inserts advice into the

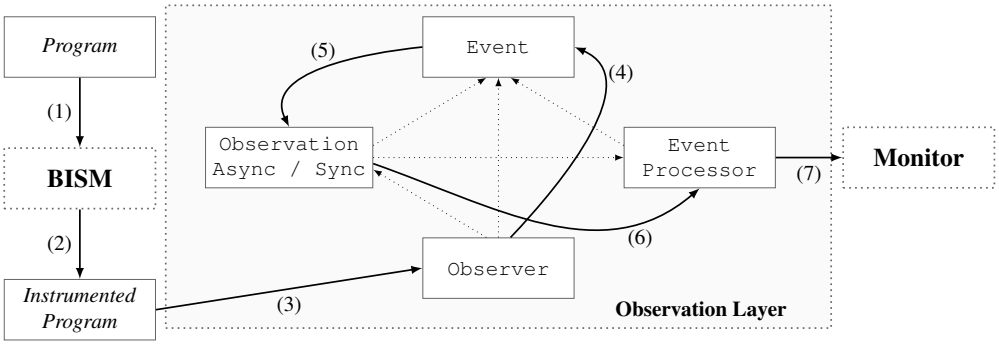


Figure 5.7: High-level design diagram for the observation layer. Solid lines show the flow of information. The dotted lines show module dependencies.

Feature	BCEL [Apa]	ASM [BLC02]	Javassist [Chi00]	Soot [VRCG+99]	DiSL [MVZ+12]	AspectJ [KHH+01b]	BISM
Bytecode Level Coverage	✓	✓	✓	✓	✓	✗	✓
Unrestricted Control	✓	✓	✓	✓	✗	✗	✓
Writing Guided Instrumentation	✓	✓	✓	✓	✓	✗	✓
No Bytecode Proficiency	✗	✗	✓̸	✓	✓	✓	✓
High-Level Abstractions	✗	✓̸	✓	✓	✓	✓	✓
Pointcut/Advice Model	✗	✗	✗	✗	✓	✓	✓̸

Table 5.1: Comparison of the tools. In red are features concerning expressiveness, and in blue are features concerning abstraction. ✓- Tool provides the feature, ✗- Tool does not provide the feature, ✓̸- Tool partially provides the feature.

base program and BISM handles the generation and weaving of the needed bytecode instructions.

Comparison with AspectJ. AspectJ offers the pointcut/advice model where the user specifies the advice using Java syntax. Whereas with BISM, advice is specified with the advice methods (Section 5.3.4). Any other code associated with selectors is executed statically during instrumentation time. As such, BISM can be used to implement guided instrumentation, where the user can specify the instrumentation points and the contextual information using custom static analyzers. Both BISM and AspectJ, provide two DSLs for specifying instrumentation, a Java-based API and an external DSL.

AspectJ does not inline advice next to the join points. Instead, it always inlines advice in external classes, and the base program is instrumented to call these external classes at the join points. This approach introduces significant overhead and makes it difficult to inline advice next to the join points. In BISM, the advice methods are weaved with minimal bytecode instructions and are always inlined next to the targeted regions.

AspectJ provides a rich set of pointcuts, static and dynamic, that can be used to target join points. BISM provides a fixed set of statically evaluated selectors that can be used to target join points and does not provide dynamic pointcuts such as *cflow*, *this*, *args* and *if* from AspectJ. However, AspectJ cannot capture all the join points that BISM can capture. For example, AspectJ cannot capture the execution of an *if* statement. Moreover, AspectJ does not provide a mechanism to access the full static context of the join point. Instead, it provides a limited set of predefined static context objects. BISM, on the other hand, provides a richer set of static context objects that can be accessed from any selector. Moreover, custom static context objects can be defined by the user. Users of AspectJ are not required to have any knowledge of bytecode semantics, whereas BISM requires basic knowledge about bytecode semantics from its users such as locating the location of an argument on the stack.

Comparison with DiSL. For writing advice, DiSL follows the pointcut/advice model where the user specifies the advice using Java syntax. Whereas with BISM, advice is specified with the advice methods (Section 5.3.4). Any other code associated with selectors is executed statically during instrumentation time. BISM, in addition to its API-based instrumentation language, provides also an external DSL for specifying instrumentation, whereas DiSL provides only an API-based instrumentation language. Both tools inline the advice code next to the join point shadows. BISM provides a fixed set of selectors, whereas DiSL's open join point model allows the marking of any arbitrary sequence of bytecode instructions as a pointcut. Such functionality can still be achieved in BISM using the *instruction* selectors. BISM provides built-in control-flow context from selectors to provide access to the control-flow graph of the base program. These can still be achieved in DiSL by writing custom markers and context objects. As for dynamic context objects, both BISM and DiSL provide equal access.

Both tools do not offer dynamic pointcuts such as *cflow*, *this*, *args* and *if* from AspectJ. However, implementing these within a DiSL advice is straightforward since it provides typed dynamic context objects, whereas, with BISM, it requires writing intricate bytecode instructions. Also, both tools are capable of inserting synthetic local variables. Both BISM and DiSL require basic knowledge about bytecode semantics from their users such as locating the location of an argument on the stack. However, since instrumentation with DiSL will probably require writing custom markers and context objects, this knowledge becomes more essential and also requires additional ASM syntax knowledge. DiSL allows only restricted modifications to the base program. It does not allow the insertion of arbitrary bytecode instructions. However, it provides a mechanism to write custom transformers in ASM that can be executed before DiSL's pass. Whereas BISM allows the direct insertion of arbitrary bytecode instructions, and

hence unrestricted modifications to the base program.

All in all, DiSL provides more features (mainly targeted for writing dynamic analysis tools) and enables dynamic dispatch amongst multiple instrumentations and analysis without interference [BMTA16], while BISM is more lightweight, as will be shown by our evaluation. DiSL provides full bytecode coverage when performing load-time instrumentation, as opposed to BISM which is restricted to the classes that can instrumented as per the `java.lang.instrument` API. Hence, it can target any class that is loaded by the JVM, including core classes such as the `Thread` class. Moreover, DiSL runs a separate virtual machine for instrumentation, while BISM runs as a standalone tool and requires no installation.

Transformer composition. Composition and interference problems in aspect-oriented programming have been studied in the literature. Interference between different aspects is commonly addressed as *aspect interactions* and *aspect interference*. The main objective is to detect places of interaction between different aspects (collision of transformers in BISM) and provide mechanisms to resolve conflicts. In [DFS02], a framework for the detection and resolution of aspect interactions is presented. The work provides a formal model for aspect weaving and a framework for detecting and resolving conflicts between aspects using static analysis. In [TC10], the work focuses on unexpected behavior of combined advice (advice interference). They show that controlling the order of execution of advice is not enough in some instances. They propose an AspectJ extension with a new resolver *around* advice for resolving interference where there is a conflict. The introduced resolver can be implemented separately and composed to resolve interference between other resolvers. BISM provides a built-in feature to capture transformer collision after a run. However, we do not provide a mechanism for resolving conflicts, which can be addressed in our future work. Composition conflicts are also studied in the literature concerning the base program and a single aspect. In [HNB06] and [HNBA07], composition conflicts related to introductions to the base program are modeled and detected using a graph-based approach. Introductions are constructs that affect the structure of a class, such as changing the inheritance structure, and adding and removing methods. In BISM, such introductions are possible since the user is free to use the ASM structure and modify the class structure. However, we do not address such conflicts and keep the user responsible for avoiding them.

5.8 Conclusion

In this chapter, we presented, BISM, a new bytecode instrumentation tool for JVM programs. We detailed the implementation choices we took for the BISM which combines elements from bytecode manipulation frameworks and AOP frameworks. BISM capabilities subsume the expressiveness capabilities of bytecode manipulation frameworks we covered in Section 3.3.4 and Section 3.6.1. However, it lacks the pointcut/advice model from AOP frameworks as well as the dynamic pointcuts from AspectJ. BISM instrumentation mechanism allows for the specification of weave-time analysis code and advice code together within the same constructs and using the same abstractions. This enables guiding the instrumentation process to refine the instrumentation points. We presented a DSL for BISM that allows for a more straightforward, declarative specification of instrumentation requirements for runtime verification. We presented the architecture of BISM and its instrumentation workflow, along with the DSL implementation. We also presented an observation layer for BISM that allows for integrating various dynamic analyses with the instrumentation process.

Part II

Guiding Instrumentation with Residual Analysis

CHAPTER 6

Residual Runtime Verification of Parametric Properties

Contents

6.1 Introduction **77**
6.2 Residual Analysis of Parametric Properties **77**
6.3 Residual Analysis via Intraprocedural Reachability Analysis **79**
6.3.1 Motivating with an Example 79
6.3.2 Capturing a Program Model 80
6.3.3 Extending the Automaton of Bad Prefixes 82
6.3.4 Cutting the Behavior 83
6.3.5 Scope and Soundness of the Analysis 84
6.4 Implementation **85**
6.5 Related Approaches **86**
6.6 Conclusion **86**

Chapter abstract

In this chapter, we present a novel method for residual runtime verification of parametric properties, incorporating the semantics of these properties into the instrumentation process with BISM. We introduce an approach that utilizes reachability analysis on control flow graphs of methods, specifying lightweight weave-time static analyses to identify safe execution paths at the intra-procedural level of programs. Such paths, guaranteed to preserve the monitored property, can thus be ignored at runtime. This guides an instrumentation tool to select only the necessary program points for observation. Our method, designed to be independent of external static analysis frameworks, can be fully performed within the instrumentation process, allowing for modular and adaptable integration into various runtime verification scenarios. This approach minimizes instrumentation points by integrating both property semantics and program behavior, and is applicable to a broad range of safety and co-safety properties.

6.1 Introduction

Motivation. One way to tackle the overhead challenges in runtime verification is to combine static and runtime verification to offload the runtime monitor from verifying parts of the program that can be statically guaranteed to preserve a property. While monitors typically depend on runtime information accompanying the events to decide a verdict, such information is generally not available statically. However, some static guarantees can be provided by relying on sound over-approximations of the behavior of the program. Here is where pre-instrumentation analysis comes into play. By capturing a model of the program that abstracts and over-approximates its behavior, we can perform such an analysis to find safe execution paths in the control flow at the intra-procedural level of programs. Such paths are guaranteed to preserve the monitored property and thus can be ignored at runtime. This analysis guides an instrumentation tool to select program points that should be observed at runtime. Eventually, the monitor is left to perform runtime verification for the *residual* parts of the program that the analysis could not statically prove safe. Moreover, we require that this analysis be lightweight and not depend on dataflow analysis, thus separating the task of residual analysis from static analysis approaches; allowing for seamless integration with many RV frameworks and development pipelines.

Methodology. We focus on properties that can be expressed by finite-state automata, such as typestate [SY86a] errors, supporting different formalisms and monitoring approaches that allow specifications with data. These monitoring approaches typically rely on the detection of *bad* and *good prefixes*, which are intuitively the witnessing sequences allowing a monitor to conclude about monitoring the program based on the trace observed so far (see Section 2.2). In such approaches, a parametric monitor receives a parametric trace with events carrying data and spawns multiple monitors for different trace slices corresponding to sets of related objects [HRTZ18b].

Given a property and a program, addressing challenge C2 we are interested in answering the following questions:

- **Q1**: Can we fully verify the program statically? If yes, then there is no need to instrument and runtime monitor it.
- **Q2**: If not, can we verify some parts of it? How can we find them so that we only monitor the *residual* parts?

Since **Q2** is the general case of **Q1**, we focus on answering **Q2**. To achieve this, for each program method we construct the CFG Automaton, that is capable of exploring all possible traces. Moreover, we over-approximate the aliasing relations between objects in the program by assuming that the variables generating events within one method *may-alias*. To deal with different trace slices, we extend the monitor representing the language of bad/good prefixes to handle the overapproximations. After that, we perform a reachability analysis on the CFG Automaton and the extended bad-prefix automaton to find safe execution paths that are guaranteed to preserve the monitored property. Finally, we construct the *residual instrumentation* function which will be used by the instrumentation tool to select program points that should be observed at runtime and ignore the safe ones.

Contributions. The contributions of this chapter can be summarized as follows:

1. We define the residual runtime verification for parametric properties.
2. We present the control-flow graph automaton, a behavioral model extracted from methods to abstract its behavior.
3. We instantiate residual analysis at the interprocedural level that does not depend on data-flow analysis.
4. We separate the problem of static analysis from the residual analysis, allowing for seamless integration with the RV workflow.

Chapter organization. The chapter is organized as follows. Section 6.2 defines residual runtime verification and its requirements. Section 6.3 describes our instantiation of residual analysis at the interprocedural level. Section 6.4 briefly presents the implementation of our approach. Section 6.5 reviews related research focusing on residual analysis. Section 6.6 concludes the chapter.

6.2 Residual Analysis of Parametric Properties

Given a parametric property $\Lambda X.\varphi$ (see Section 2.3) and a program P, we want to statically verify that $P \models \Lambda X.\varphi$. The behavior of a program can be abstracted by the set of parametric event traces that it can produce at runtime. Let

$[P] \subseteq \Sigma\langle X\rangle^*$ be such set for a program P. The verification problem can be stated as checking if all the parametric traces of the program satisfy the property:

$$P \models \Lambda X.\varphi \stackrel{\text{def}}{=} \forall \tau \in [P] : \tau \models \Lambda X.\varphi$$

Recall that a parametric trace is extracted from the program at runtime and is then sliced by the monitor into a set of *projected traces* (see Section 2.3). Any static verification technique aimed at verifying the program should then be capable of exploring two key elements. First, it should be able to explore all the parametric traces in $[P]$ that the program can generate.

```
1  void m()
2  {
3     List l1 = ... ;
4     List l2 = ... ;
5
6     l1.add(..); // event u
7     Iterator it = l1.iterator();// event c
8
9     if(someflag) {
10       l2.add(..); // event u
11       l2.add(..); // event u
12    }
13
14    Object o = it.next(); // event n
15    it = l1.iterator(); // event c
16    l2.add(..); // event u
17 }
```

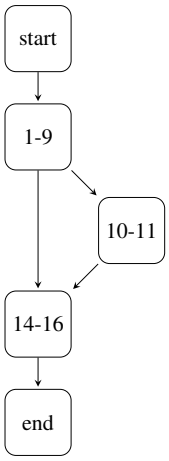


Figure 6.1: A method using Iterators in Java, and its CFG.

EXAMPLE 25 (PARAMETRIC TRACES) Figure 6.1, shows a Java method *m* along with its control-flow graph (CFG). It retrieves 2 lists (lines 3,4), updates them (lines 6,10,11), creates iterators (lines 7,15), and calls the "next" method on the iterator (line 14). Let us say we are interested in monitoring the **SafeIterator** property which specifies that "*A collection should not be updated when an iterator associated with it is created and being used*". By ignoring other methods in the program for now, this method depending on the evaluation of the condition on Line 9 at runtime may produce the following $[P]$ with two different parametric traces:

- $\tau = (u, [l \mapsto o(l1)])\ (c, [l \mapsto o(l1), i \mapsto o(it)])\ (u, [l \mapsto o(l2)])\ (u, [l \mapsto o(l2)])\ (n, [l \mapsto o(l1), i \mapsto o(it)])\ (c, [l \mapsto o(l1), i \mapsto o(it)])\ (u, [l \mapsto o(l2)])$.
- $\tau' = (u, [l \mapsto o(l1)])\ (c, [l \mapsto o(l1), i \mapsto o(it)])\ (n, [l \mapsto o(l1), i \mapsto o(it)])\ (c, [l \mapsto o(l1), i \mapsto o(it)])\ (u, [l \mapsto o(l2)])$. *

As such the analysis needs to explore two parametric traces in total.

Second, from Definition 7, we know that a parametric trace τ is verified if all its projected traces are verified. As such, for each parametric trace, the technique should also be capable of exploring the set of projected traces, that is the different ways of slicing the parametric trace into a set of projected traces.

EXAMPLE 26 (PROJECTED TRACES) Consider τ from Ex. 25. Here depending on the aliasing relation between $o(l1)$ and $o(l2)$, we can have three projected traces:

- If $o(l1) = o(l2)$, then $Proj(\tau) = \{ucuuncu\}$
- If $o(l1) \neq o(l2)$, then $Proj(\tau) = \{ucnc, uuu\}$

As such, the analysis needs to verify three projected traces in total for this parametric trace.

Exploring the parametric traces statically requires knowledge of the call graph of the program, whereas verifying the projected traces requires knowledge of the aliasing relations between objects producing them. We know that

obtaining such information is generally undecidable statically. Pointer analysis may not always conclude with a result, especially for Java programs. [1] Meanwhile, at runtime, this information is completely available. Yet, runtime verification incurs overhead on the execution of the program where this overhead is typically positively correlated with the size of traces. Our interest is then to statically verify parts of the program and leave a residual part for runtime verification.

The residual analysis, we propose, *statically* identifies a set of instructions in the program, $\mathcal{S}_P$, that can be safely silenced/ignored at runtime from the monitor side without affecting verification. Ignoring an instruction means that there is no need to produce an event when it executes. As such, we want to construct the *residual instrumentation* function. Let us denote the set of all instructions in the program by instrs. An execution of the program can be abstracted by a sequence of instructions, which we denote by instrs*.

DEFINITION 28 (RESIDUAL INSTRUMENTATION FUNCTION) The residual instrumentation function residual : instrs* $\rightarrow$ ($\mathcal{S}_P$ $\rightarrow$ $\Sigma\langle X\rangle^*$) maps a run in the program to a set of instructions $\mathcal{S}_P$ that can be safely ignored during runtime without affecting the verification process. Ignoring an instruction implies that no event is produced when it executes.

Let us note *Runs* $\subseteq$ instrs* the set of all the possible (feasible) runs of a program P. Consider the function instrument that maps a run in the program to a set of events produced by the monitor. Instrumenting the program with residual should ideally produce shorter traces than instrument, however, for both, we should get the same monitoring verdict.

DEFINITION 29 (RESIDUAL ANALYSIS CONDITION) We can state the condition that should be met by the residual analysis as follows:

$$\begin{aligned} \forall r \in \mathit{Runs} : \quad & |\text{residual}(r)| \leq |\text{instrument}(r)| \\ \wedge \quad & \text{residual}(r) \models \Lambda X.\varphi \iff \text{instrument}(r) \models \Lambda X.\varphi \end{aligned}$$

To perform the residual analysis statically and produce the set $\mathcal{S}_P$, we can over-approximate the program behavior by constructing a set $[\widehat{P}] \supseteq [P]$. This allows us to explore all the parametric traces that the program can produce but also traces that the program might never produce. A residual analysis should then check whether silencing some instructions does not affect the verification verdict of any trace in $[\widehat{P}]$, and safely assumes the same effect in $[P]$. Yet, given that $[\widehat{P}]$ is an over-approximation, the analysis may suffer from false positives, which are instructions that can indeed be silenced however the analysis found the opposite. In what follows, we consider a subset $[\widehat{P_m}] \subseteq [\widehat{P}]$ for our residual analysis, these are traces that are fully produced in single methods.

6.3 Residual Analysis via Intraprocedural Reachability Analysis

We demonstrate our instantiation of the residual analysis at the intraprocedural level using reachability analysis. In Section 6.3.2, we capture the behavior of a method by using its control-flow graph to construct a representative model that allows us to explore the parametric traces a method can generate. In Section 6.3.3, we deal with the over-approximations by extending the bad-prefix automaton to handle different projections that might be produced by a parametric trace. In Section 6.3.4, we then present the reachability analysis algorithm that finds *safe* and *violating* paths in the control-flow graph; by cutting the behavior in a model-based checking approach. Finally, in Section 6.3.5, we discuss the soundness of our analysis.

6.3.1 Motivating with an Example

Recall the program in Fig. 6.1 and **Q1** and **Q2** from Section 6.1. By manually inspecting the program and its control-flow graph we see that, at runtime, it may violate the property if the execution enters the `if` block, labeled (10-11) in the graph. More precisely, a violation can occur if both of the following conditions are met: **(1)** `someflag` evaluates to *true*; and **(2)** if the variables `l1` and `l2` alias each other i.e., they refer to the same object in memory. Let us consider that Condition **(1)** is only decidable at runtime. To generally decide Condition **(2)**

[1] In addition to the inability to construct the full static call graph of the program, Java allows for dynamic class loading and reflection which often cause additional problems to pointer analysis.

statically, we need to perform pointer analysis on the program that checks all calling contexts `m` and return whether `l1` and `l2` alias. In practice, we may get one of the following results about our query: the two objects *must-alias*, *may-alias*, or *must-not-alias*. Moreover, pointer analysis often times out and never returns a result. However, to answer **Q1**, we need to get the result that `l1` and `l2` *must-not-alias*, i.e., they refer to different objects in memory. This is a sufficient condition to statically ensure that *m* will behave correctly at runtime regardless of the control flow since the update actions on Lines 10-11 are not on the list iterated by iterator `it`. To answer **Q2**, by observing Lines 15-16, we can see that, regardless of what happens at execution, these two instructions are safe and their execution does not need to be monitored. Also, in Line 6, the instruction is safe since it updates the list before the creation of the iterator.

Pointer analysis may not always conclude with a result, especially for Java programs. In addition to the inability to construct the full static call graph of the program, Java allows for dynamic class loading and reflection which often cause additional problems to pointer analysis. Our work relies on the idea that when statically analyzing cases such as the one of Condition (2), one can safely assume that such two variables *may-alias*, even without performing pointer-analysis. Also, we analyze methods separately and thus need to handle escaping references. Objects in the program that are relevant to the property may escape from the method to a subroutine or a return statement and produce events there. As such, we handle all instructions that may allow references to escape, such as method calls, with special *escape events*.

Our over-approximation might miss some positive answers to **Q1**, therefore missing some optimization opportunities. However, based on our observation (such as the experiments in Section 11.5), cases where one needs to perform pointer analysis such as in Condition (2) are less frequent in many Java programs. As such, our approach mainly addresses **Q2** while it is also capable of answering **Q1** but, in certain cases, less effectively.

6.3.2 Capturing a Program Model

Our analysis treats methods separately, however, we need to be careful. If a method receives as an argument an object which is a type that is capable of producing events in the alphabet of the property, then we cannot assume any previous behavior. As such, we exclude such methods from the analysis. This is easy to check given the static context available to our analysis. For the same reason, we exclude all methods that operate on static instances of the types involved.

For each method *m*, we map two types of instructions to events and discard all other instructions as they are irrelevant to our analysis. We keep instructions that produce events in Σ, given by the property specification. We also keep instructions that may allow any object reference to escape from the context of method *m*; we introduce the new *escape* event (#) for such instructions. Escape events are assignments to class fields, method calls that pass objects by references, in addition, to return statements that return objects [CGS+99]. However, our analysis allows the user to specify a safe list of instructions, denoted by the set *SafeList*, defined over the compile type information such as method names, package and type names, and opcodes. For instance, calling `System.out.print(l1.toString())` is a safe instruction. All instructions that are escape events and are not in *SafeList* are added to the set Esc_m.

Given the alphabet of a property Σ and the control-flow graph $CFG_m = \langle B_m, E_m \rangle$, then for all b in B_m, we replace it with b' and map the instructions to events as follows.

$$b'.instr \;=\; b.instr.\text{map}\left(i \mapsto \begin{cases} i & \text{if } instrument(i) \in \Sigma, \\ \# & \text{else if } i \in Esc_m \\ \epsilon & \text{otherwise} \end{cases} \right)$$

That is, we erase all instructions we are not interested in and replace them with ϵ, and keep the others intact. Method calls that are not in *SafeList* are replaced by *escape* events #.

After removing the irrelevant instructions, we now proceed in splitting the nodes of CFG_m such that each remaining instruction is represented in a unique node containing its mapped event. We modify the control-flow graph, by constructing a new graph as follows.

DEFINITION 30 (SPLIT CFG) Given the $CFG_m = \langle B_m, E_m \rangle$ with instructions mapped to events, we construct a new modified graph $SCFG_m = \langle B'_m, E'_m \rangle$ as follows:

$$B'_m = \bigcup_{b \in B_m} \text{split}(b) \quad , \tag{1}$$

$$\begin{aligned} E'_m = \; & \{ \langle b', b_i \rangle \mid \langle b', b \rangle \in E_m, b_i \in \text{split}(b) : \\ & \qquad \text{idx}^b(i) = 0 \} \qquad (2) \\ & \cup \{ \langle b_i, b_j \rangle \mid b_i, b_j \in \text{split}(b) : \\ & \qquad 0 \leq \text{idx}^b(i) < |b.instr| - 1 \\ & \qquad \wedge \; \text{idx}^b(j) = \text{idx}^b(i) + 1 \} \qquad (3) \\ & \cup \{ \langle b_i, b' \rangle \mid \langle b, b' \rangle \in E_m, b_i \in \text{split}(b) : \\ & \qquad \text{idx}^b(i) = |b.instr| - 1 \} \qquad (4) \end{aligned}$$

where:

- $\text{idx}^b : b.instr \to \mathbb{N}$ returns the index of an instruction in a block b.
- $\text{split} : B_m \to 2^{B_m}$ defined as $\text{split}(b) = \{b_i \mid i \in b.instr\}$ is a function that splits a basic block into multiple nodes, such each one contains an instruction, ϵ or the escape event #. The first node of an entry block is set as the entry node and the last node of an exit block is set as the exit node.

The newly created graph, $SCFG_m$, is constructed as follows. All basic blocks in CFG_m are split into multiple nodes, one node per instruction (1). All incoming edges to an original block, are connected to the node representing the first instruction of the original block (2). All nodes that are created from a single original block are connected sequentially (3). All outgoing edges from the original block are now outgoing from the last split node (4).

A node in the new graph $SCFG_m$ contains one letter which represents: either an instruction that generates events in Σ, or an escape event #, or an ϵ. The entry block in the new graph contains the first event of interest having $b.entry$ equals $true$. We merge two consecutive nodes containing ϵ, and move their edges accordingly.

The CFG Automaton is then constructed from the Split CFG $SCFG_m$.

Definition 31 (CFG Automaton) Given the mapped $SCFG_m$, the CFG Automaton is the non-deterministic finite-state automaton $\mathcal{A}^c_m = (\Sigma \cup \{\#\}, Q, \delta, q_0, F)$ constructed as follows:

$$\begin{aligned} Q &= \{q_b \mid b \in B'_m\} & q_0 &= \{q_b \mid b \in B'_m \wedge b.entry = true\} \\ F &= Q & \delta &= \{\langle q_b, s, q_{b'} \rangle \mid \langle b, b' \rangle \in E'_m \wedge b.\text{instr} = s\} \end{aligned}$$

Each node in the control-flow graph is now represented as a state in the CFG Automaton. We make all states accepting states and merge states connected with ϵ transitions. Now, by traversing the CFG Automaton, we can explore the paths that method m can take at runtime and thus the parametric traces it can produce.

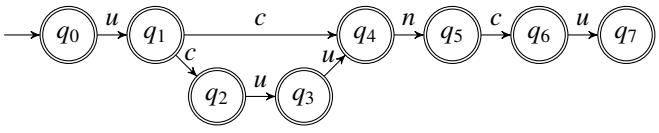


Figure 6.2: The constructed CFG Automaton $\mathcal{A}^c_m$

Example 27 (CFG Automaton) Figure 6.2, shows the CFG Automaton constructed from the method in Figure 6.1. Each state corresponds to an instruction that we are interested in the program. Two traces can be explored from the automaton in Figure 6.2, $t_1 = ucuuncu$, which corresponds to the parametric trace τ from Ex. 25, and $t_2 = ucncu$. *

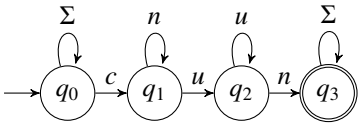


Figure 6.3: Automaton $\mathcal{A}^{bad_\varphi}$ recognizing the language of bad prefixes for the *SafeIterator* property.

6.3.3 Extending the Automaton of Bad Prefixes

We first recall the definition of bad prefixes from Section 2.2. A bad prefix is a finite trace $u \in \Sigma^*$ such that $\forall w \in \Sigma^* : uw \notin L$. As such a monitor is constructed to recognize the bad prefixes of a property. It suffices to find a bad prefix in a trace to conclude that the whole trace violates the property.

EXAMPLE 28 (SAFEITERATOR BAD PREFIXES AUTOMATON) Figure 6.3 shows the monitor that checks for the violation of the **SafeIterator** property. Note that the monitor reaches the accepting state when seeing the pattern $c.n^*.u^+.n$, as it suffices to conclude that the whole run violates the property. ∗

We now describe how our analysis handles the over-approximations and extends the bad prefix automaton.

Handling Variables May-alias

Recall from Section 2.3, that a parametric trace τ in $\Sigma\langle X\rangle^*$ at runtime is projected into $Proj(\tau)$ to possibly multiple traces in Σ^*, depending on the aliasing relationship between the objects carried in the events. At runtime, this aliasing relationship is available for the parametric monitor to do the projection. However, statically for our residual analysis this information is not available. Our central idea in this paper is to avoid performing data-flow analysis and assume that the objects producing events in a method may-alias. For two events, our analysis should then consider the case when the objects bound to them *must-alias* and the case when they *must-not-alias*. In the former case, both events will be projected into the same trace, and in the latter, they will be projected into different traces.

EXAMPLE 29 (PROJECTED TRACES APPROXIMATION WITH MAY-ALIAS) Consider the trace t_1 which can be explored with the CFG Automaton from Ex. 27. At runtime, if the program takes such a control flow path, it emits a parametric trace that produces either of the projected traces from Ex. 26 depending on whether $l1$ and $l2$ alias. Since we avoid producing the aliasing relation statically and assume that $l1$ and $l2$ may alias, we should then consider in our residual analysis the disjunction of both cases. Thus the traces $pt_1 = \{ucuuncu, ucnc, uuu\}$ should be checked by our residual analysis. As for t_2 from Ex. 27, by the same reasoning, the traces to be checked are $pt_2 = \{ucncu, ucnc, u\}$. Hence for method m, the set of traces that should be checked is $pt_1 \cup pt_2$. ∗

The CFG Automaton allows us to explore the different paths that the program can take at runtime, however, its traces are too coarse. They may be *polluted* with events that do not correspond to the same trace at runtime. We notice from above that this is equivalent to generating and considering all the subwords of a trace, where the real trace can be any subword of a trace that can be explored with the automaton. Thus, to safely handle the different projections, we use the upward closure, from Section 2.4, of the language of bad prefixes L. By using the upward closure $\uparrow L$, we can recognize a bad prefix in a full trace or any subword of it since $L \subseteq \uparrow L$, allowing us to find bad prefixes in all possible projected traces. However, we restrict the closure by removing the Σ self-loops from the initial and final states as we want to find the shortest paths that match a bad prefix.

Handling Escape Events

In Section 6.3.2, when constructing the CFG automaton, we introduced the escape # events. Since our analysis analyzes each method separately, we are oblivious to what might be happening in #-transitions. We have to assume that they might produce events untracked by the method under analysis. To handle them safely, we add a #-transition in the bad prefixes automaton from each state to all of its reachable states. Intuitively, this means when # event is encountered in a path, we assume that the path is not safe anymore and that it might match a bad prefix.

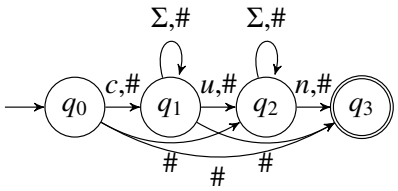


Figure 6.4: The constructed automaton $\mathcal{A}^{\uparrow bad_\varphi}$.

Extending the Bad Prefixes Automaton

We proceed to show how we extend the automaton of bad prefixes to handle the multiple projected traces and the escape events.

DEFINITION 32 (EXTENDED AUTOMATON OF BAD PREFIXES) Given the language of bad prefixes $\mathcal{L}(bad_\varphi)$ recognized by automaton $\mathcal{A}^{bad_\varphi} = (\Sigma, Q, \delta, Q_0, F)$ with its extended transition function $\hat{\delta}$. The extended automaton of bad prefixes is defined as $\mathcal{A}^{\uparrow bad_\varphi} = (\Sigma \cup \{\#\}, Q, \delta', Q_0, F)$ where:

$$\begin{aligned} \delta' = \quad & \delta \setminus \{ \langle q, s, q \rangle \mid s \in \Sigma \wedge (q \in F \vee q \in Q_0) \} && (1) \\ \cup \quad & \{ \langle q, s, q \rangle \mid s \in \Sigma \cup \{\#\} \wedge q \in Q \wedge q \notin Q_0 \wedge q \notin F \} && (2) \\ \cup \quad & \{ \langle q, \#, q' \rangle \mid q, q' \in Q \wedge \exists w \in \Sigma^* : \hat{\delta}(q, w) = q' \wedge q' \notin F \} && (3) \end{aligned}$$

The extended automaton has the same states. We remove the self-loops from initial and final states, as we want to find the shortest paths that match a bad prefix (1). We add the upward closure by adding Σ and # self-loops on all other states (2). We add #-transitions from each state to the reachable states from it (3).

EXAMPLE 30 ($\mathcal{A}^{\uparrow bad_\varphi}$ FOR THE SAFEITERATOR PROPERTY) Figure 6.4, shows a construction of automaton $\mathcal{A}^{\uparrow bad_\varphi}$. Recall the pattern $c.n^*.u^+.n$ from Ex. 28, the new automaton will now recognize such a pattern while also handling the two over-approximations above.

6.3.4 Cutting the Behavior

We now proceed to describe how we find violating paths in the method. The idea is to traverse the constructed CFG automaton $\mathcal{A}^c_m$ state by state and check whether there is a path, starting from the visited state, that makes the extended bad prefixes automaton reach a final state. We limit the discussion here to matching bad prefixes, nevertheless, the same analysis works for matching good prefixes. However, when finding paths that match good prefixes, these will be the safe paths.

Given an automaton, $\mathcal{A}$, $\mathcal{A}(q)$ denotes $\mathcal{A}$ where q is set to be the initial state. Recall that, given a finite state machine $\mathcal{A}(q)$ with its extended transition function $\hat{\delta}$ [HMU06], a state q is coreachable if there exists a word $s \in \Sigma^*$ such that $\hat{\delta}(q, s) \in F$. State q is reachable if there exists a word $s \in \Sigma^*$ such that $\hat{\delta}(q_0, s) = q$ and q_0 is an initial state.

Algorithm 1, shows how to mark all states in $\mathcal{A}^c_m$ as either safe (in $\mathcal{P}_m$) or violating (in $\mathcal{V}_m$). The algorithm implements a depth-first search starting from the initial node of $\mathcal{A}^c_m$. We maintain a *work* stack and *visited* set, in lines (4,5,7,9,15), to hold automaton states to be visited and states that were already visited, respectively. For each state q we visit, we set q as the initial node and find the intersection with the $\mathcal{A}^{\uparrow bad_\varphi}$, line (11). If the intersection is not empty (line 12), we find the set of all co-reachable states in the intersection automaton. Each state in the intersection automaton $\hat{\mathcal{A}}$ corresponds to a state in $\mathcal{A}^c_m$ and $\mathcal{A}^{\uparrow bad_\varphi}$. For each coreachable state in $\hat{\mathcal{A}}$, we add its corresponding state in $\mathcal{A}^c_m$ to the set $\mathcal{V}_m$, (line 13). We do not revisit states that are already in $\mathcal{V}_m$ (line 10) since paths leading to a final state in $\mathcal{A}^{\uparrow bad_\varphi}$ are already explored by the intersection.

EXAMPLE 31 (PROPERTY VIOLATING STATES) Figure 6.5, shows the CFG Automaton constructed from the program from Figure 6.1, where states marked in red exist in a property-violating path. The red states in the

Algorithm 1: Marking violating and safe paths

Given $\mathcal{A}^c_m = (\Sigma \cup \{\#\}, Q, \delta, q_0, F)$, $\mathcal{A}^{\uparrow \text{bad}_\varphi}$
$\mathcal{V}_m = \emptyset$ // represents all states in a violating path
$\mathcal{P}_m = Q$ // represents all states in a safe path
work := q_0 // represents a worklist stack
visited = $\emptyset$
while *work not empty* **do**
  q = *work*.pop()
  if $q \notin$ *visited* **then**
    visited = *visited* $\cup\, q$
    if $q \notin \mathcal{V}_m$ **then**
      $\hat{\mathcal{A}} = \mathcal{A}^c_m(q) \times \mathcal{A}^{\uparrow \text{bad}_\varphi}$
      if $\mathcal{L}(\hat{\mathcal{A}}) \neq \emptyset$ **then**
        $\mathcal{V}_m = \mathcal{V}_m \cup \{\{q' \mid (q', -) \in \text{coreachable}(\hat{\mathcal{A}})\} \cap \text{reachable}(\mathcal{A}^c_m(q))\}$
    foreach q'' *in* $\{q'' \mid \langle q, s, q'' \rangle \in \delta\}$ **do**
      work.push(q'')
    end
end
$\mathcal{P}_m = \mathcal{P}_m \setminus \mathcal{V}_m$

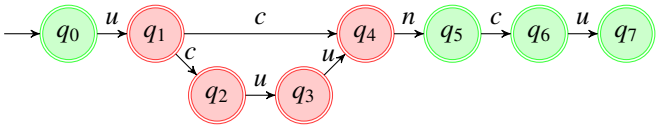


Figure 6.5: Marking property violating paths in red, and safe in green.

automaton are the states that we need to instrument, and the green states are hidden from instrumentation. We can see that instead of instrumenting at 8 different locations, we only have to instrument at 4 locations.

For our residual analysis, for each method m we analyze, we add the instructions corresponding to the states in $\mathcal{P}_m$ to the set $\mathcal{S}_P$. As for the other states in $\mathcal{V}_m$, their corresponding instructions will be instrumented for runtime monitoring.

6.3.5 Scope and Soundness of the Analysis

We first argue that our analysis only affects the traces that are fully produced in one method. Recall from Section 6.3.2, that the nodes of CFG automaton $\mathcal{A}^c_m$ correspond to instructions in method m that produce events. We use the notation ev(q) to denote the corresponding event from an automaton state, and events(t) to denote all events from a trace t. If some trace t contains events produced by instructions outside of m, then no instruction in m that produced events in t was marked safe.

PROPOSITION 1 (SCOPE OF THE ANALYSIS) Given a parametric trace τ in $[P]$:

$$\forall t \in \mathit{Proj}(\tau), \forall m \in \mathit{Methods} :$$
$$(\exists i \in \text{instrs} \backslash \mathit{Instructions}_m : \text{instrument}(i) \in t)$$
$$\implies \{ \text{ev}(q) \mid q \in \mathcal{P}_m \} \cap \text{events}(t) = \emptyset$$

Proof. Assume that there exists some trace t that has events produced outside of m i.e. $\exists i \in \text{instrs} \backslash \mathit{Instructions}_m$: instrument(i) $\in t$ is true. Such traces can be split into two types. Traces that contained events before the execution of m at runtime (1), and traces that start from m but have some events that are produced outside of m at runtime

(2). We will show that for both types of traces, the analysis would result in $\mathcal{P}_m = \emptyset$. Since we exclude from the analysis any method that receives a parameter of a type that generates events. Traces from (1), will not be affected by analysis. For that to happen, method m should receive the objects generating the events. Therefore, m will be excluded from the analysis resulting in $\mathcal{P}_m = \emptyset$. As for (2), any escape of an object, which might produce events outside m, is captured by the # transitions. From the construction of the bad-prefixes automaton and Algorithm 1, such transitions will result in reaching a final state from any state in the CFG automaton, resulting in $\mathcal{P}_m = \emptyset$. Hence for both types of traces we have $\mathcal{P}_m = \emptyset$, therefore $\{\text{ev}(q) \mid q \in \mathcal{P}_m\} \cap \text{events}(t) = \emptyset$ holds, and the proposition holds. □

Proposition 1 in fact depends on the specification of the *SafeList* from Section 6.3.2. If some method was added by the user that is not safe, i.e. allows references to escape, then the proposition will not hold. From the above, we also see that our analysis only affects instructions that produce events only in traces that are collected fully in the method itself since otherwise $\mathcal{P}_m = \emptyset$. For soundness, we need to guarantee that at any run of the program, an event that we marked safe in our residual analysis does not have any effect on deciding the violation/satisfaction of the property for any projected trace at runtime. As we showed that the analysis only affects projected traces that are fully produced in one method, we only reason about single methods when discussing soundness.

PROPOSITION 2 (SOUNDNESS OF THE ANALYSIS) Given a language $L \subseteq \Sigma^*$ and $\mathcal{P}_m$ resulting from the analysis on method m, the analysis is sound iff

$$\begin{aligned} &\forall a_1 \cdots a_i \cdots a_n \in \Sigma^+, \forall i \in \mathbb{N} : \\ &\quad match_L(a_1 \cdots a_i \cdots a_n) \neq match_L(a_1 \cdots a_{i-1} a_{i+1} \cdots a_n) \\ &\qquad \implies a_i \notin \{\, \text{ev}(q) \mid q \in \mathcal{P}_m \} \end{aligned}$$

The condition states that given a projected trace at runtime, if we remove an event a_i from it and get a different match from the new trace i.e. $match_L(a_1 \cdots a_i \cdots a_n) \neq match_L(a_1 \cdots a_{i-1} a_{i+1} \cdots a_n)$, then our analysis must have not statically marked a_i as safe ($a_i \notin \{\, \text{ev}(q) \mid q \in \mathcal{P}_m$).

Proof. The proof follows from the definition of Algorithm 1. Assume that when our analysis removes a_i, then $match_L(a_1 \cdots a_i \cdots a_n) \neq match_L(a_1 \cdots a_{i-1} a_{i+1} \cdots a_n)$. This means that a_i is in an extension of $a_1 \cdots a_{i-1}$ that leads to a final state in the monitor of the bad-prefixes of L, or else $match_L(a_1 \cdots a_i \cdots a_n) = match_L(a_1 \cdots a_{i-1} a_{i+1} \cdots a_n)$. However, if a_i is in such a path, then it will be added to $\mathcal{V}_m$ as per Line 7 of Algorithm 1 since the algorithm finds any path from a state in the CFG that reaches the final state of the automaton of bad-prefixes. Then a_i is not in $\mathcal{P}_m$ and $a_i \notin \{\, \text{ev}(q) \mid q \in \mathcal{P}_m\}$ holds. □

6.4 Implementation

Our implementation is integrated into BISM. We augment BISM with specialized modules to facilitate residual runtime verification. Specifically, we introduce a new transformer that performs the static verification and an additional module for automata operations. For each property, we employ two transformers for each property: the static analyzer for pre-instrumentation analysis and a second transformer for instrumenting the residual code.

Static analyzer transformer. The static analyzer is not a separate component but rather implemented as one of the transformers in BISM. This transformer performs a one-pass pre-instrumentation analysis at the intra-procedural level. It takes as input the property specification and identifies and hides the visibility of instructions that can be statically verified as safe, thereby leaving only the residual, or unverified, portions of the program code for a subsequent transformer that is responsible for instrumenting the residual code for runtime monitoring.

Automata operations module. To enable the static analyzer, we extend BISM with a specialized module that enables automata-based analyses. This module is responsible for generating the Control Flow Graph (CFG) automata for methods. It also extends these automata to capture bad prefixes and identifies execution paths that could potentially lead to property violations.

6.5 Related Approaches

Many research approaches combine static and runtime verification. We focus here on some influential and most recent tools devised for verifying general behavioral parametric properties in sequential programs via residual analysis.

CLARA [BLH12, BLH10] handles properties that can be expressed by finite-state automata by partially evaluating the runtime monitors at compile time and reducing the instrumentation points. It performs three-staged phases of analysis with increasing precision. The more precise phase uses a demand-driven pointer analysis and handles intra-procedural analysis. The first two phases of its analysis can be easily applied within our framework. However, unfortunately, CLARA is no longer maintained and so is its underlying instrumentation tool the abc compiler [Fit00]. In [WCM13], the authors present two optimizations for [BLH12]. One optimization identifies changeless configurations during the backward analysis; the other one uses local object information to refine the forward analysis and backward analysis of the *nop-shadow* analysis. CLARVA [ACP20] extends CLARA [BLH12] to handle properties expressed by DATEs (Dynamic Automata with Events and Timers [CPS09a]) where events are guarded by runtime conditions and timers. Similar to our approach, it transforms Java code into an automaton-based model and allows for the incorporation of control-flow analyses. CLARVA is capable of reducing the instrumentation points as well as reasoning about and pruning the property itself. However, the analysis relies on constructing the callgraph of the full program and on pointer analysis using Soot [VRCG+99]. Our approach is still capable of producing optimizations with a single pass on the program and without any dependence on static analysis, separating the limitations of static analysis from the residual analysis.

STARVOORS [CAPS15] combines deductive theorem proving with control-flow reachability analysis allowing to target control and data-oriented properties. The formalism used for property specification is ppDATE (an extension of DATE) where the automaton states are extended with pre/post-conditions (*Hoare triples*). The property is reduced by pruning the transitions based on solving the triples with Java theorem prover KEY [ABH+06]. STARVOORS is capable of handling control and data-oriented properties, however, it focuses on pruning the property and does not reduce the instrumentation points.

In [Leu12, ZLD12], the authors present Predictive Semantics for runtime monitoring at the intra-procedural level. In this setup, the program is analyzed, using the control flow graph (CFG) and program dependence graph (PDG), to find predictive words. Predictive words are events that will occur in sequence in a control-flow path. Then, the monitor at runtime will either receive a single event or a predictive word. This approach does not reduce the instrumentation points, hence does not reduce the overhead of instrumentation, however, it emits predictive words which may produce faster verdicts.

6.6 Conclusion

We introduce an analysis supporting *residual runtime verification* for parametric properties that can be expressed by finite-state automata. Our approach over-approximates the behavior of the program and analyzes its methods separately relying only on their control-flow graphs to statically identify safe regions. We have demonstrated the effectiveness of our approach in monitoring the bad prefixes of a property, however, our approach can also be used with good prefixes (when monitoring co-safety properties for instance). Our approach is capable of producing overhead optimizations without any dependence on a specific type of static analysis, separating the task of static analysis from the residual analysis and allowing for seamless integration with many RV frameworks. However, this separation comes at the cost of over-approximation, which can be reduced by integrating static analysis results into the residual analysis. It is fully implemented and integrated within the BISM instrumentation tool, which is the state-of-the-art instrumentation tool for Java programs. We also demonstrated the significant performance benefits at runtime.

Part III

Monitoring Concurrent Programs

CHAPTER 7

Representative Traces for Concurrent Programs

Contents

7.1 Introduction **91**
7.2 Trace Collection for Concurrent Programs **92**
7.2.1 Issues with Linear Traces 93
7.3 Concurrent Traces **94**
7.4 Sound and Faithful Concurrent Traces **94**
7.5 Obtaining Sound Concurrent Traces **95**
7.5.1 Atomicity and Instrumentation Requirements 95
7.5.2 The Reordering Algorithm 96
7.5.3 Algorithm Cost 96
7.5.4 Algorithm Correctness 96
7.6 Criteria For Monitorability **98**
7.6.1 Monitor Causal Dependence 98
7.6.2 Trace Monitorability of Concurrent Executions 100
7.6.3 Optimal Faithfulness 100
7.7 Implementation **101**
7.8 Related Approaches **101**
7.9 Conclusion **102**

Chapter abstract

This chapter focuses on collecting *representative* traces from concurrent executions. We define a representative trace as a trace that is both sound and faithful. A sound trace is a trace that does not provide false information about the order of events. A faithful trace is a trace that includes all ordering information between events. To collect representative traces, we present a reordering algorithm based on vector clocks that can operate either synchronously or asynchronously with the program to establish event causality on the fly. We then present new criteria for assessing the validity of the collected trace. We redefine the concept of trace monitorability based on automata-based formalisms. Our refined definition integrates a causal dependence relation extracted from a given property to identify non-permutable events within a trace. This allows us to evaluate whether a trace contains sufficient order information to yield a sound monitoring verdict. With such information, existing monitoring frameworks relying on total order formalisms can soundly monitor concurrent programs. We implement the trace collection and assessment in a tool called FACTS implemented on top of BISM. The collected representative traces can be used by existing monitoring frameworks to soundly monitor concurrent programs.

7.1 Introduction

Motivation. In the context of concurrent programs, traces serve as a model for property-based *detection* and *prediction* techniques which choose their trace collection approaches differently based on the class of targeted properties. Some approaches target generic classic concurrency errors such as data-races [HMR14a, FF09], deadlocks [BH05], and atomicity violations [FF04, WS06, MV20]. Other techniques target general behavioral properties; those are typically order violations such as null-pointer dereferences [FPRS12], and typestate violations [JS08, HLR15, SCR12] and more generally runtime verification [LS09, FHR13b]. The need for establishing causality in traces is particularly evident for properties involving concurrency [AG96, ANB+95, MPA05]. Existing techniques either rely on vector clock algorithms, which are computationally expensive and not optimized for online use [RGB20], or assume a sequentially consistent execution model [HS12a], limiting their applicability. Classical monitoring techniques, initially designed for single-threaded programs, have been adapted for multithreaded contexts but often produce unsound verdicts due to incorrect assumptions about event ordering [EF18]. We aim to address these limitations by collecting representative traces suitable for sound online monitoring of concurrent programs, particularly those involving general behavioral properties [MP90, Pat]. The objective is to improve the expressiveness and reliability of existing monitoring techniques in concurrent settings.

Methodology. We first consider partially ordered traces of concurrent executions and qualify two properties that determine if they are good representatives of an execution: *soundness* and *faithfulness* (Section 7.4). Soundness holds when the trace does not provide false information about the order of events. Faithfulness holds when the trace contains all the information about the order of events from the execution. We then address two research questions:

- **RQ1** addressing C1 : Can we collect a representative trace of a concurrent execution on the fly with minimal interference on the program?

Vector clock algorithms have been employed and refined for several decades now [CL02, MV20, AG05, RS04]. However, very few (we know of [RGB20]) are directed towards online monitoring of behavioral properties; where a final trace consists of property-related events only. These algorithms typically require blocking the execution; by synchronizing the instrumentation, program actions, and algorithm's processing to avoid data races [CL02]. With the quadratic bound on their runtime complexity and the coarse-grained synchronization they introduce, one might want to run such an algorithm off the critical path of the program. We present a vector clock algorithm that does not require blocking the execution and can run either synchronously or asynchronously with the program. This algorithm requires instrumenting and capturing all synchronization actions in the program. Asynchronous trace collection is ideal for scenarios where the monitoring overhead cannot be afforded and a small delay in the verdict can be tolerated. For instance, in real-time systems where the system is expected to produce a result within a defined strict deadline [SR94]. Our algorithm constructs representative concurrent traces that are sound and very often faithful. The algorithm might miss, in some marginal cases, ordering information resulting in sound but unfaithful traces (see Section 7.5). As far as we know, it is unique in the context of monitoring behavioral properties that can run off the critical path of the execution.

- **RQ2 addressing C3**: Is the collected concurrent trace good enough to soundly monitor a property?

Monitoring single-threaded programs depend on instrumentation, which is assumed to be correct, to provide all relevant events [BFFR18a]. For concurrent programs, we notice that monitorability with classical approaches depends also on the ordering information available in the trace (resp. the execution). Firstly, consider for instance the precedence property seen previously. If events r and g happen to execute concurrently in the program, a sound trace would have them as unordered events and a sound monitor should report a violation of the property. However, in practice, when monitoring such property with an automaton, the partial order must be linearized before passing it to the monitor. The linearization will produce an arbitrary order between r and g, for instance, $r.g$ which would make the oblivious monitor miss the violation. In such a case, the trace is not fit for monitoring the property and the monitor should be warned. Secondly, in certain scenarios due to some partial instrumentation or logging failure, some synchronization actions might be missing from the trace, resulting in unfaithful traces where some orderings cannot be established. This also poses soundness problems to the monitor similar to the problems with lossy traces [JTF17]. To handle the mentioned problems, we extract a *causal dependence relation* from a given property to know which events cannot permute in a trace and check whether a trace contains enough order information (Section 7.6). We then redefine trace *monitorability* for concurrent executions with a necessary condition on the trace to guarantee a sound verdict when monitoring. If the condition is not met, we produce warnings for the monitor. Figure 7.1 shows an overview of our methodology where we reconstruct a partial order trace of the

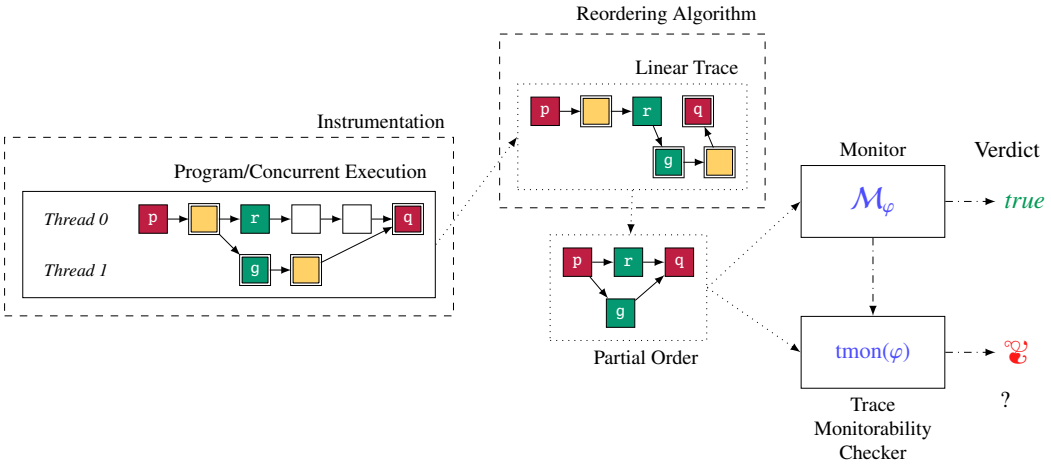


Figure 7.1: A concurrent execution with a partial trace.

execution and check its monitorability.

Contributions. Here is a summary of our contributions:

- We introduce the properties of soundness and faithfulness to qualify representative traces, and show how a representative trace correctly abstracts the execution and preserves the property of interest.
- We present a vector clock-based reordering algorithm that does not require blocking the execution and can run either synchronously or asynchronously with the program.
- We redefine trace monitorability for concurrent executions and provide a necessary condition for sound monitoring.
- We implement our contributions in a tool, FACTS, which attaches to programs running on the JVM.

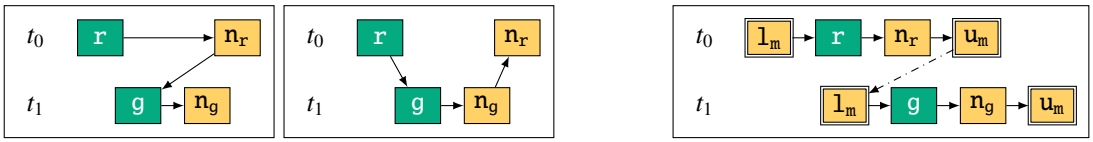
Chapter organization. The rest of this chapter is organized as follows. Section 7.2 discusses trace collection and the issue of using the linear traces as is. Section 7.3 define a concurrent trace. Section 7.4 introduces the properties of soundness and faithfulness. Section 7.5 describes our vector clock algorithm for collecting representative traces. Section 7.6 discusses the concept of trace monitorability and provides a necessary condition for sound monitoring. Section 7.7 presents our implementation of the tool FACTS. Section 7.8 reviews some related work. Section 7.9 concludes the chapter.

7.2 Trace Collection for Concurrent Programs

Any monitor of a concurrent execution needs to first collect a trace of the execution. Recall the definition of a concurrent execution in Def. 14. Trace collection is achieved by instrumenting the program. Additional code needs to be instrumented into the program and executed. We refer to such actions that notify the monitor as *notification actions*, denoted by the set $\mathbb{NA} \subseteq \mathbb{A}$.

Notification actions are often method invocations that also pass to the monitor needed context from the executing actions. In general, we cannot interrupt the execution of a program action; we can intercept the points in time just before or just after the execution of the instruction responsible for the action. Notification actions are associated with a direction and execute either *before* or *after* the execution of their actions. Instrumentation is given by the partial function $\text{notify} : \mathbb{A} \times \{\text{before}, \text{after}\} \rightharpoonup \mathbb{NA}$.

When the instrumented program executes, a monitor (or any observer) receives the trace as a sequence of notifications. To handle concurrent threads producing notifications, a monitor should implement a locking mechanism to protect itself from concurrent access. This mechanism forces a total order on the notifications. As such a linear trace is a totally ordered set of notifications.



(a) Unsound instrumentation.

(b) Forcing atomicity.

Figure 7.2: Instrumenting concurrent events.

DEFINITION 33 (LINEAR TRACE) A linear trace is a totally ordered set $(\mathbb{NA}, \xrightarrow{\text{to}})$ of notification actions such that $\xrightarrow{\text{to}}$ is an arbitrary linearization of $\xrightarrow{\text{e}} \cap (\mathbb{NA} \times \mathbb{NA})$, the execution order restricted by the set $\mathbb{NA}$.

7.2.1 Issues with Linear Traces

Monitoring approaches relying on total-order formalisms such as LTL and finite state machines require linear traces to be fed to the monitors as their input consists of words. In this section, we discuss certain issues that arise when using linear traces from concurrent executions.

Advice atomicity assumption. General purpose runtime verification frameworks such as Java-MOP [CR05a], Tracematches [AAC+05a], and others [BFB+17] rely on instrumentation for extracting events for traces. As mentioned, to handle concurrent programs, RV tools provide a feature to *synchronize* the monitor to protect it from concurrent access and data races. As such, threads acquire a lock before notifying the monitor and release it afterward. However, when a program action is executed, its advice will not always execute atomically with it unless both are wrapped in a synchronization block.

EXAMPLE 32 (ADVICE ATOMICITY) Consider the traced execution in Fig. 7.2a where green boxes represent program actions and yellow boxes represent their advice, in this case, the advice is a single notification action that notifies the monitor, and arrows indicate the execution order of these actions. A context switch happens after the `r` is executed (but before notifying the monitor), allowing the execution of `g` and its advice before $\mathtt{n}_\mathtt{r}$. It is clear how this would result in unsound results when monitoring the property `r` *precedes* `g`. ∗

This observation problem cannot be solved by simply checking for the absence of data races. One needs to guarantee atomicity between all executing program actions and their advice. The problem will not appear if `r` and `g` are synchronized and executed with their advice within mutually exclusive regions, such as in Fig. 7.2b. Nevertheless, we might sometimes want to capture events from concurrent regions.

Forced atomicity. One way to solve the lack of advice atomicity is to force it. Forced atomicity can be achieved by instrumenting synchronization blocks[1] that wrap the program actions with their advice in mutually exclusive regions. However, forcing atomicity introduces two new problems, one at the program level and the other at the monitor level. First, it forces a total order between concurrent program actions; interfering with the parallelism of the program and changing its behavior. One needs also to minimize the area for which the lock is applied and avoid coarse-grained synchronization. From the monitor side, the verdict will be dependent on the specific scheduling of the execution. Take `r` *precedes* `g` for instance, if these events are concurrent in the program, monitoring will produce a different verdict at each run. Second, any information about parallel actions in the program is lost and one can no longer determine whether two actions execute concurrently initially in the non-instrumented program. In that case, it becomes impossible to express properties on the concurrent parts of the execution. Furthermore, RV tools mentioned above rely on AspectJ [KHH+01a] for a high-level specification of instrumentation which is rather unfitting to instrument synchronization blocks as it cannot instrument at the bytecode level.

[1] Such as `synchronized` in Java.

7.3 Concurrent Traces

Our goal is to capture a representative trace from concurrent executions which we defined in Def. 14. We define a concurrent trace as a subset of the execution actions, and a partial order over these actions.

DEFINITION 34 (CONCURRENT TRACE) A concurrent trace, denoted as t, is a partially ordered set $t = (\mathbb{E}, \xrightarrow{\text{tr}})$ where:

- $\mathbb{E}$ is a set of trace events, such that $\mathbb{E} \subseteq \mathbb{A}$,
- $\xrightarrow{\text{tr}}$ is a partial order that establishes the sequence of these events within the concurrent execution.

We note that the trace actions $\mathbb{E}$ are called *events* in monitoring and verification approaches. The trace events $\mathbb{E}$ are assumed to be a subset of the execution actions, capturing the pertinent actions for monitoring purposes. Furthermore, Σ, which are the events of interest for property monitoring, are considered as a subset of $\mathbb{E}$, thus:

$$\Sigma \subseteq \mathbb{E} \subseteq \mathbb{A}$$

This subset relation simplifies the projection from captured traces to the set of events relevant for monitoring. While different monitoring techniques may necessitate distinct projections, for the purpose of this thesis, we equate the trace events $\mathbb{E}$ with the monitored events Σ. A discussion on projection techniques used in parametric monitoring for instance can be found in Section 2.3.

7.4 Sound and Faithful Concurrent Traces

In this section, we show how a representative trace preserves the properties of an execution.

We first define the notion of *trace soundness*. Informally, a trace is a sound trace if it does not provide false information about the execution.

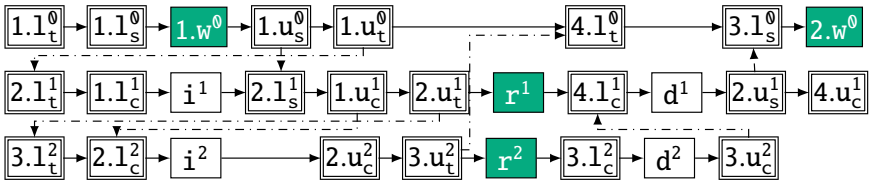
DEFINITION 35 (TRACE SOUNDNESS) A concurrent trace t is said to be a sound trace of a concurrent execution e (noted $\text{snd}(e, t)$) iff (i) $\mathbb{E} \subseteq \mathbb{A}$ and (ii) $\xrightarrow{\text{tr}} \subseteq \xrightarrow{\text{e}}$.

To be sound, a trace (i) should not capture an action not found in the execution, and (ii) should not order unrelated actions.

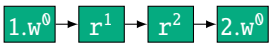
While a trace that does not provide incorrect information about the execution model leads to sound monitoring, a trace can still not provide enough information about the order (for a monitor to determine a verdict). Faithfulness is similar to *completness*; it is expensive as it requires capturing all relevant causality in the program. Informally, a *faithful* trace contains all information on the order of events that occurred in the execution model.

DEFINITION 36 (TRACE FAITHFULNESS) A concurrent trace t is said to be faithful to an execution e (noted $\text{faith}(e, t)$) iff $\xrightarrow{\text{tr}} \supseteq (\xrightarrow{\text{e}} \cap\, \mathbb{E} \times \mathbb{E})$.

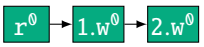
EXAMPLE 33 (SOUNDNESS AND FAITHFULNESS) Recall the execution from Example 13. For a behavioral property such as "no read or write happen concurrently with another read or write", we are interested in the actions `r` and `w`. The order relative to these events in the execution is $\rightarrow_{e_0} = \{\langle 1.\mathtt{w}^0, \mathtt{r}^1\rangle, \langle 1.\mathtt{w}^0, \mathtt{r}^2\rangle, \langle \mathtt{r}^1, 2.\mathtt{w}^0\rangle, \langle \mathtt{r}^2, 2.\mathtt{w}^0\rangle, \langle 1.\mathtt{w}^0, 2.\mathtt{w}^0\rangle\}$. Fig. 7.3b presents a linear trace of the execution t_0 as captured by Java-MOP using advice atomicity assumption, see Section 7.2.1. One can see that the order is $\rightarrow_{t_0} = \rightarrow_{e_0} \cup \{\langle \mathtt{r}^1, \mathtt{r}^2\rangle\}$. We notice that we have faithfulness ($\text{faith}(e_0, t_0)$) as $\rightarrow_{e_0} \subset \rightarrow_{t_0}$. However we do not have soundness ($\neg\text{snd}(e_0, t_0)$) as the pair $\langle \mathtt{r}^1, \mathtt{r}^2\rangle \notin \rightarrow_{e_0}$. Indeed, the reads happened concurrently. Fig. 7.3c shows a trace that is neither sound nor faithful. Fig. 7.3d the trace captures thread order only. It is a sound trace, as it only contains $\langle 1.\mathtt{w}^0, 2.\mathtt{w}^0\rangle$, and therefore no wrong information. However, it is not faithful, as it is missing order information. Fig. 7.3e presents a partial trace of the execution t_3 that is both sound and faithful. Ideally, t_3 is the smallest concurrent trace collected to verify behavioral properties on reads and writes. ∗



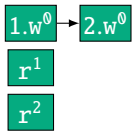
(a) Full Concurrent Execution (e_0)



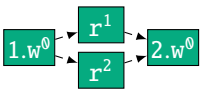
(b) $\overline{\text{S}}$/F Linear (t_0)



(c) $\overline{\text{S}}$/$\overline{\text{F}}$ Linear (t_1)



(d) S/$\overline{\text{F}}$ Partial (t_2)



(e) S/F Partial (t_3)

Figure 7.3: Four different collected traces from the execution of 1-Writer 2-Readers.

A property can be used to define the set of correct concurrent executions. For the verification of the temporal behavior of programs, the semantics of matching properties is applied on traces [MP95]. Effectively, at the semantic level, a property partitions the set of all executions into correct and incorrect ones. Consequently, a property $\varphi(\Sigma)$ defined over $\Sigma \subseteq \mathbb{A}$ is a set of partial orders over Σ.

DEFINITION 37 (PROPERTY SATISFACTION) A concurrent execution $(\mathbb{A}, \overset{e}{\rightarrow})$ satisfies a property $\varphi(\Sigma)$ (noted $(\mathbb{A}, \overset{e}{\rightarrow}) \models \varphi(\Sigma)$) iff $(\overset{e}{\rightarrow} \cap \Sigma \times \Sigma) \in \varphi(\Sigma)$.

To check that a property has been satisfied, we simply "project" the order $\overset{e}{\rightarrow}$ on Σ, that is, we restrict our information about the execution to Σ. We then check if the projection belongs to the set of correct concurrent executions ($\varphi(\Sigma)$).

While our goal is verifying properties on the full execution of a program, we generally gather a subset of it as a trace. As such, we are interested in the fact that verifying a property on the trace holds the same as it would on the full execution. By construction (from Def. 35 and 36), we notice that the projections over Σ (Def. 37) of some execution e and a trace t for a property over variables $\Sigma \subseteq \mathbb{E} \subseteq \mathbb{A}$ where we have soundness (snd(e,t)) and faithfulness (faith(e,t)) are the same. We deduce the following theorem.

PROPOSITION 3 (PROPERTY PRESERVATION) Given a concurrent execution $\mathtt{e} = (\mathbb{A}, \overset{e}{\rightarrow})$, and a trace $\mathtt{t} = (\mathbb{E}, \overset{\text{tr}}{\rightarrow})$, and a property φ over $\Sigma \subseteq \mathbb{E} \subseteq \mathbb{A}$ we have:

$(\text{snd}(\mathtt{e}, \mathtt{t}) \wedge \text{faith}(\mathtt{e}, \mathtt{t})) \implies (\mathtt{e} \models \varphi(\Sigma) \text{ iff } \mathtt{t} \models \varphi(\Sigma))$.

We say that t is an appropriate abstraction of e; e and t can be used interchangeably to verify properties over Σ. Since our notion of a property is simply a set of traces, the presented results apply to any set of monitorable [FFM12, PH18] properties.

7.5 Obtaining Sound Concurrent Traces

We present a vector clock algorithm that constructs sound concurrent traces. The algorithm differs from standard algorithms in that it can run asynchronously, allowing scenarios where a delay can be tolerated. Most, if not all vector clock algorithms in the literature block the execution to process the algorithm at each event.

7.5.1 Atomicity and Instrumentation Requirements

One of the features of the algorithm is its relaxed requirements for atomicity. Specifically, the algorithm does not necessitate advice atomicity for its correct operation. This is a significant point as it reduces both the complexity

and potential overhead associated with implementing the algorithm in a real-world system. Below are two instrumentation requirements that are sufficient for the algorithm to operate correctly.

- **Requirement 1**: When an action a is instrumented with a *before* (or *after*) directive, the corresponding notification action, $\mathtt{n}_a$, is required to occure before a (or after a). Importantly, no other action from the same thread is permitted to execute in between.
- **Requirement 2**: The algorithm requires *before* and *after* directives for instrumenting *release* and *acquire* actions respectively. With this requirement, since the synchronization order is the only relation that orders actions from different threads in the execution. Then, any two ordered actions from different threads must either be themselves notifications for synchronization actions or must have at least one synchronization action between them that has been observed and collected.

7.5.2 The Reordering Algorithm

Algorithm 2 maintains the following data structures. A map $\mathbb{L} : T \to V$ holding the last timestamp seen by each thread. A map $\mathbb{R}$ holding the last release action per resource, a map $\mathbb{W}$ holding the last write of a value on a shared variable, and a set $\mathcal{T}_{\text{ts}}$ which represent the concurrent trace and will be populated with timestamped actions. On each received notification action, the algorithm sets its timestamp to the latest timestamp seen by its thread (line 3). Then, synchronization actions in SA (lines 4-5) are send to `ReleaseOrAcquire`, except for *read* or *write*. Actions *unlock*, *fork*, *end* are represented with *release*, the algorithm puts them in the map entry in $\mathbb{R}$ associated with their resource in R (lines 11-12). Actions *lock*, *begin*, *join* are represented with *acquire*, the algorithm retrieves the last action that released its resource from R; if found [2], their vector clocks are merged (lines 13-15). The *join* is merged with the last action seen by the finished thread. Actions *read* and *write* are handled with `ReadOrWrite`. They are handled with the map $\mathbb{W}$ which is indexed by a shared variable and a value. A write (lines 17-23) is only pushed into $\mathbb{W}$ when it does not conflict with the entry in $\mathbb{W}$ associated with its variable and value (discussed more in the correctness section). A read is merged with the latest write (lines 24-26). After that, the timestamp is incremented (line 6), and map $\mathbb{L}$ is updated (line 7). Events will be stored in the concurrent trace, and all synchronization actions will be discarded (lines 8-9).

EXAMPLE 34 (CONCURRENT EXECUTION REORDERING) Consider the execution depicted in Fig. 7.3a in Ex. 33. Table 7.1 shows the timestamps of events given by the algorithm in Algorithm 2. The final trace $\mathcal{T}_{\text{ts}}$ will contain event e_1 = `1.w` with $e_1.VC$ = [3,0,0], event e_2 = `1.r` with $e_2.VC$ = [5,6,0], event e_3 = `2.r` with $e_3.VC$ = [5,5,5], and event e_4 = `2.w` with $e_4.VC$ [8,8,7]. You can notice that the two reads e_2 and e_3 are concurrent since neither $e_2VC \leq e_3VC$ nor $e_3VC \leq e_2VC$. Whereas reads are ordered with writes. The captured trace corresponds to the sound and faithful trace from Fig. 7.3e.

7.5.3 Algorithm Cost

The *Join*, *Comparison* and *Copy* operations of vector clocks require $\Theta(k)$ time, linear in the number of threads k. *Increment* operations on vector clocks, retrieving and inserting elements to the maps require $O(1)$. The reordering algorithm then requires $O(n \times k)$ time for a trace of n actions.

7.5.4 Algorithm Correctness

We now show how the algorithm always produces sound traces. After the algorithm ends, the collected linear trace is timestamped into a concurrent trace represented by $\mathcal{T}_{\text{ts}}$ that contains ordered pairs $\langle a, b\rangle$, such that $a.VC \leq b.VC$.

Handling Release and Acquire actions. For handling *release* and *acquire* actions (except for reads and writes), these actions already execute in a total order in any execution. The algorithm performs classical operations of timestamp merging for matching actions. However, the correctness of our algorithm is dependent on instrumentation. Since we do not force atomicity between an action and its notification, we must instrument actions that perform vector clock merge operations i.e. *acquire* actions with the *after* directive and *release* actions with the *before*

[2] We omit some null checks to simplify the presentation of the algorithm, however, assume that joining $e.VC$ with a null value does not affect it.

Algorithm 2: Vector Clock Algorithm

```
1  procedure ReceiveAction(e)
2      t = tid(e)
3      e.VC = copy(𝕃(t))
4      if e is a synchronization action then
5          send e to the appropriate procedure
6      inc_t(e.VC)
7      𝕃(t) = e.VC
8      if e is not a synchronization action then
9          𝒯_ts ← e                                   // Add e to trace 𝒯_ts
10 procedure ReleaseOrAcquire(e)
11     if e = release(t, r) then
12         ℝ(r) = e                                   // Update ℝ
13     else if e = acquire(t, r) then
14         e' = ℝ(r)
15         e.VC = e.VC ⊔ e'.VC                        // Merge vector clocks
16 procedure ReadOrWrite(e)
17     if e = write(t, x, v) then
18         e' = 𝕎(x, v)
19         if (e' = null) ∨ (e'.VC ≤ e.VC) then
20             𝕎(x, v) = e
21         else
22             𝕎(x, v) = null                         // Conflicting write
23         end
24     else if e = read(t, x, v) then
25         e' = 𝕎(x, v)
26         e.VC = e.VC ⊔ e'.VC
```

directive. The intuition is that if a *release* action is instrumented with the *after* directive, a context switch between it and its advice can lead to having an unmerged consecutive *acquire*.

For any *release* and *acquire* actions that are instrumented correctly as per Requirements 1 and 2, the algorithm ensures that if a matching *release* exists for an *acquire*, it has already been captured and processed. Below is a sketch of the proof.

1. Given proper instrumentation, a *release* action's vector clock is updated before the action is executed, ensuring that any subsequent *acquire* action on the same resource will have a vector clock that can be safely merged.
2. Therefore, when an *acquire* action is observed, if a matching *release* exists, it has already been captured and processed.

When the program only uses *lock* and *fork* actions for synchronization, then our algorithm guarantees both soundness and faithfulness of the concurrent trace since it captures all orderings, provided that they are all instrumented and captured.

Handling Reads and Writes. Handling *reads* and *writes* is more demanding if we want to run the algorithm asynchronously and do not want to force on them a total order in the execution. Reads and writes are instrumented with *after* and *before* directives, respectively, including volatile and atomic variables [LBM+]. Atomic operations such as *compare-and-swap* are handled differently since we check for the result of the operation before emitting an action.

For any read r and write w actions that are instrumented correctly as per Requirements 1 and 2, the algorithm ensures that r will either match its write and merge, or not. Below is a sketch of the proof.

1. On observing a read r on a shared variable x with a value v (line 24), the algorithm checks $\mathbb{W}(x, v)$ to find its

Table 7.1: Step by step timestamping of events from the execution in Ex. 33 using Algorithm 2.

Step	Event e	Thread	$e.VC$	Step	Event e	Thread	$e.VC$
1	$1.\mathtt{l}_\mathtt{t}$	0	[1,0,0]	13	$2.\mathtt{l}_\mathtt{c}$	2	[5,5,2]
2	$1.\mathtt{l}_\mathtt{s}$	0	[2,0,0]	14	$2.\mathtt{u}_\mathtt{c}$	2	[5,5,3]
3	$1.\mathtt{w}$	0	[3,0,0]	15	$3.\mathtt{u}_\mathtt{t}$	2	[5,5,4]
4	$1.\mathtt{u}_\mathtt{s}$	0	[4,0,0]	16	$2.\mathtt{r}$	2	[5,5,5]
5	$1.\mathtt{u}_\mathtt{t}$	0	[5,0,0]	17	$3.\mathtt{l}_\mathtt{c}$	2	[5,5,6]
6	$2.\mathtt{l}_\mathtt{t}$	1	[5,1,0]	18	$3.\mathtt{u}_\mathtt{c}$	2	[5,5,7]
7	$1.\mathtt{l}_\mathtt{c}$	1	[5,2,0]	19	$4.\mathtt{l}_\mathtt{c}$	1	[5,7,7]
8	$2.\mathtt{l}_\mathtt{s}$	1	[5,3,0]	20	$2.\mathtt{u}_\mathtt{s}$	1	[5,8,7]
9	$1.\mathtt{u}_\mathtt{c}$	1	[5,4,0]	21	$4.\mathtt{u}_\mathtt{c}$	1	[5,9,7]
10	$2.\mathtt{u}_\mathtt{t}$	1	[5,5,0]	22	$4.\mathtt{l}_\mathtt{t}$	0	[6,5,4]
11	$1.\mathtt{r}$	1	[5,6,0]	23	$3.\mathtt{l}_\mathtt{s}$	0	[7,8,7]
12	$3.\mathtt{l}_\mathtt{t}$	2	[5,5,1]	24	$2.\mathtt{w}$	1	[8,8,7]

matching write (line 25).

2. If prior to r, multiple writes $\mathtt{w}(t, x, v)$ and $w'(t', x, v)$ occurred writing on x the same value v, such that they are not ordered i.e. *conflicting* ($w.VC \not\leq w'.VC$ and $w'.VC \not\leq w.VC$), the map entry $\mathbb{W}(x, v)$ is cleared (line 22) to prevent unsafe merging.

3. Therefore, a read r will either find a non-conflicting write to merge with or not find a matching write at all. We say the two writes on x, writing the same value v, are *conflicting* when they are not ordered, i.e. $w.VC \not\leq w'.VC$ and $w'.VC \not\leq w.VC$.

This might have consequences on the faithfulness of the trace and not its soundness. Faithfulness will only be affected if (1) three or more threads are writing to the same variable the same value and in a concurrent region, and (2) the program relies on those writes to synchronize. We reasonably believe that this is an infrequent case in concurrent programs. However, for such programs, reads and writes can be instrumented to have advice atomicity; by instrumenting synchronization blocks that wrap them with their advice in mutually exclusive regions. This will force a total order on reads and writes and there will be no need for the procedure `ReadOrWrite`. They can be treated as *acquire* and *release* and handled with procedure `ReleaseOrAcquire`.

Given both presented arguments, the algorithm generates a sound trace $\mathcal{T}_{\text{ts}}$. Given a concurrent execution denoted by $\langle \mathbb{A}, \xrightarrow{\text{e}} \rangle$, the above steps guarantee that $\mathcal{T}_{\text{ts}}$ satisfies the following condition which establishes the soundness of the algorithm:

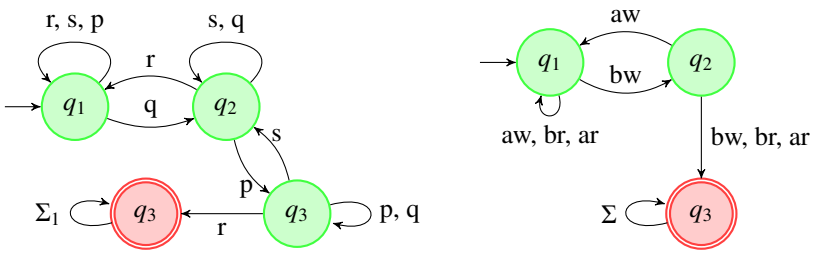
$$\forall \langle a, b \rangle \in \mathcal{T}_{\text{ts}} \times \mathcal{T}_{\text{ts}} \ : \ a.VC \leq b.VC \implies \langle a, b \rangle \in \xrightarrow{\text{e}} \tag{7.1}$$

7.6 Criteria For Monitorability

In this section, we discuss the criteria for sound monitoring with automata and concurrent traces. In Section 7.4, we see that a sound and faithful trace represent a program execution and can be used interchangeably when verifying a property. However, a concurrent trace (equivalently a concurrent execution) will contain unordered events; if they are concurrent. Many monitoring approaches rely on finite state automata as they can be used for most of the specification patterns [DAC99, Pat]. Such monitors expect a total order of events as their input consists of words. Given that a concurrent trace is a partial order, it must be linearized before proceeding with monitoring. A *linearization* of a partial order will impose an arbitrary order between unordered events. However, feeding the monitor with faulty orderings of events might lead to an incorrect verdict. For the remainder of this section, we set Σ as the set of events over which properties are specified. We distinguish it from $\mathbb{E}$, which is the set of runtime events that will possibly be projected to events in Σ.

7.6.1 Monitor Causal Dependence

When observing an automaton, we might find pairs of events whose order is irrelevant to its progress; they can permute without affecting the verdict. The *causal dependence* relation $\mathcal{D} \subseteq \Sigma \times \Sigma$ is a binary relation that is anti-reflexive and symmetric. It contains all pairs of events whose correct order is necessary and their permutation would



(a) Response property P1 (b) Property P2

Figure 7.4: Automata of Bad Prefixes.

lead the automaton to a different state. Hence, for all pairs of events that do not belong to $\mathcal{D}$, their permutation after a linearization (if they occur concurrently) can be safely tolerated. Moreover, $\mathcal{I}_D = (\Sigma \times \Sigma) \setminus \mathcal{D}$ is the independence relation induced by the dependence relation $\mathcal{D}$. For a deterministic automaton, $\mathcal{I}_D$ is a reflexive and symmetric relation. We extract $\mathcal{I}_D$ by finding pairs of actions that would lead to the same state if permuted.

Algorithm 3: Generate $\mathcal{I}_D$ for DFA $\mathcal{A}$

```
1  Given DFA A = {Σ, Q, δ*, Q0, F}
2  I_D : Σ × Σ → {tt, ff, ?} a map, initially all set to ?.
3  for (a, b) ∈ Σ × Σ do
4      if I_D(a, b) ≠ ? then continue
5      if a = b then I_D(a, a) = tt; continue
6      for q in Q do
7          if δ*(q, a.b) ≠ δ*(q, b.a) then
8              I_D(a, b) = I_D(b, a) = ff
9              continue to main loop
10         end
11     end
12     I_D(a, b) = I_D(b, a) = tt
13 end
```

Algorithm 3 extracts $\mathcal{I}_D$ from a property specified as a DFA. The algorithm checks for all pairs (a, b) in $\Sigma \times \Sigma$ if starting from any state in the automaton $a.b$ would lead to a different state from $b.a$ (lines 8-11). If that is the case, then the automaton depends on receiving both symbols in order, and the pair is added to the dependency relation. For pairs of the same symbol, it adds them into $\mathcal{I}_D$ (line 5).

Note that, the *causal dependence* relation resembles trace equivalence from [Maz87] which constrains the allowed linearizations of a partial order. However, here we extract the relation from the monitor itself, and it defines *word* equivalence concerning an automaton, that is, the allowed permutations of letters in a word that would eventually lead to the same verdict.

Example 35 (Monitor Causal Dependence) We demonstrate with two example properties. **P1** states that event `s` responds to `p` between `q` and `r`. **P2** is a mutual exclusion property that states that no read or write should happen concurrently with another write. To monitor with a finite state machine monitor, the read and writes are instrumented and delimited with events *bw* and *aw* represent before and after a write, *br* and *ar* for before and after a read. Fig. 7.4 shows the violation automata of bad prefixes to monitor the properties. For each automaton, we have:

- $\mathcal{I}_{D_1} = \{\langle s, q \rangle\}$
- $\mathcal{I}_{D_2} = \{\langle br, ar \rangle\}$ *

From $\mathcal{I}_{D_1}$, we see that the monitor does not depend on the order between s and q. From $\mathcal{I}_{D_2}$, we see that the monitor depends on the order between writes themselves, reads, and writes and reads.

7.6.2 Trace Monitorability of Concurrent Executions

We first define the notion of necessary order for a concurrent trace, which indicates whether the trace has the needed order based on the dependence relation.

DEFINITION 38 (TRACE NECESSARY ORDER) We say that a concurrent trace $t = (\mathbb{E}, \xrightarrow{\text{tr}})$ has the necessary orderings w.r.t. a causal dependence relation $\mathcal{D}$, noted $\text{tno}(t, \mathcal{D})$ when:

$$\forall\, e, e' \in \mathbb{E} : \langle e, e' \rangle \in \xrightarrow{\text{tr}} \;\vee\; \langle e, e' \rangle \notin \mathcal{D}$$

EXAMPLE 36 (TRACE NECESSARY ORDER) Back to property **P2** from Example 35 and the traces depicted in Fig. 7.3. Trace t_3 is ordered enough for monitoring the property, whereas t_2, a trace collected considering the thread order only, does not capture the order between reads and writes. As such we have:

$\text{tno}(t_3, \mathcal{D}_2) = \top$ $\qquad$ $\text{tno}(t_2, \mathcal{D}_2) = \bot$ $\qquad$ $*$

Let us recall the notion of monitorability from [KYV01, BLS11b]. A property φ is *monitorable*, denoted by $\text{Mon}(\varphi)$, if every prefix of every trace has a finite extension that allows its monitor to reach a verdict, be it positive or negative. Monitoring with unsound traces leads to unsound verdicts. We redefine monitorability for concurrent programs by adding necessary conditions on the traces.

DEFINITION 39 (TRACE MONITORABILITY OF CONCURRENT EXECUTIONS) Given a property φ with its dependency relation $\mathcal{D}$, and a trace t collected from a concurrent execution e. Property φ is monitorable with t, noted **t-Mon**(φ) when $\text{Mon}(\varphi) \;\wedge\; \text{snd}(e, t) \;\wedge\; \text{tno}(t, \mathcal{D})$.

First, the property φ should be monitorable in the classical sense, $\text{Mon}(\varphi)$, or else we will not reach a verdict. Second, the trace should be sound, $\text{snd}(e, t)$, or else we will have an unsound verdict. Third, the trace should have all the ordering information needed by the property as per its dependency relation, $\text{tno}(t, \mathcal{D})$, or else a linearization would produce an unsound trace. The above indicates that a concurrent trace does not need to contain all the ordering information between events and that the notion of faithfulness can be relaxed when monitoring. Now, if there is missing information in a sound and faithful concurrent trace, this means that the execution itself does not contain the needed causality for monitoring the property. As such, the user is warned about the missing order to address the problem, and a tradeoff is presented between concurrency and monitorability. On one hand, they can synchronize the concurrent actions in the program to have them ordered, or they can force linearization of unordered actions via instrumentation as discussed in Section 7.2.1. On the other hand, they can leave the actions executing concurrently and afford inconsistent verdicts.

7.6.3 Optimal Faithfulness

Since an ordering is a set of pairs, we can now define a ratio to faithfulness as the number of existing ordered pairs in a trace compared to the ordered pairs in an execution. For a given sound trace $t = (\mathbb{E}, \xrightarrow{\text{tr}})$, we define the trace faithfulness ratio as $R = |\xrightarrow{\text{tr}}|/|(\xrightarrow{\text{e}} \cap\, \mathbb{E} \times \mathbb{E})|$. The faithfulness ratio cannot be greater than 1 for sound traces. If all of the collected actions execute in synchronized regions in the program, then we get a faithful trace with $R = 1$.

Ideally, we want to instrument programs to capture traces with the smallest optimal faithfulness ratio, denoted as R_φ. Such traces contain only necessary orderings for monitoring some property φ. Moreover, R_φ delimits the boundary of monitorability. Obtaining lower faithfulness ratios leads to a non-monitorable execution. Obtaining more than the optimal ratio means there might be a chance of optimizing instrumentation to lower the overhead on the executing program.

Property 1 (Degrees of Faithfulness) *Given some faithfulness ratios* R' *and* R'' *for a collected trace from an execution* e *to monitor some property* φ*. If* $\text{snd}(e, t)$ *and* $\text{Mon}(\varphi)$ *hold, we have:*

(1) $\text{R}' < \text{R}_\varphi \implies \neg$**t-Mon**$(\varphi)$

(2) $\text{R}'' \geq \text{R}_\varphi \implies$ **t-Mon**(φ)

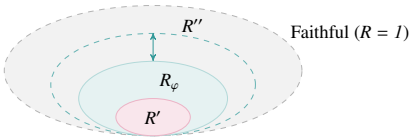


Figure 7.5: Approaching optimal faithfulness.

One can check statement (1) by simply checking if the trace satisfies the tno condition. As for (2), it presents us with an optimization opportunity, as we ideally want only to capture the orderings required for the monitor and not the complete orderings in the execution. The optimization could determine the smallest set of synchronization actions (SAs) required to capture the optimal faithful ratio. Fig. 7.5, shows a depiction of this optimization. However, obtaining optimal faithfulness with instrumentation is an exciting challenge that we leave for future work. Nevertheless, we show an example of optimizations that can help capture fewer synchronization actions from the program.

EXAMPLE 37 (OPTIMIZING INSTRUMENTATION) Let us re-examine the concurrent execution presented in Fig. 7.3a. The lock associated with the counter keeping track of readers, while useful to the program, can be omitted as the additional order provided by locks and unlocks on the counter (`c`) can be obtained from the test read lock (`t`) to order reads (`r`). However, this would not be the case if we want to establish an order for decrements (`d`). ∗

7.7 Implementation

We implemented the presented work as an extension to the observation layer for BISM we introduced in Section 5.6 We named the extension FACTS (**Fa**ithful and Sound **C**oncurrent **T**race**s**). With BISM transformers, the user specifies the actions needed for monitoring and selects the concurrency primitives they want to capture. We provide a pre-defined set of transformers for the most common concurrency primitives, such as `wait`, `notify`, `join`, `lock`, `unlock`, etc. Instrumentation extracts by default for each event the thread ID and event name (specified by the user), the resource that is producing the event, class name, method name, and line number. The *scoping* feature in BISM allows specifying different filters such as package, class, field names, etc. Instrumentation is performed at load-time, and generated events are passed at runtime to the observation layer and then into the two modules we detail below before being sent to the monitor.

Trace Reordering module. The vector clock algorithm processes the collected actions and pushes the timestamped monitoring actions into the concurrent trace, which is kept in memory or passed to the monitor. Depending on the monitoring scenario, the algorithm can run on a separate thread from the executing program or in the same thread. For representing vector clocks, we use tree clocks [MPTV22] in our implementation instead of classical vector clocks. Tree clocks improved the overall performance of the reordering algorithm and gave us a sublinear time on join and copy operations, in contrast to classic vector clocks, which always require $\Theta(k)$ time linear in the size of the vector.

Monitorability Checker module. This module checks if the concurrent trace is monitorable as discussed in Section 7.6. The takes as input a dependency relation $\mathcal{D}$ and runs attached to a running target program as a Java agent. It runs in parallel with the reordering algorithm and is concerned only with checking the tno condition. It is compatible with parametric slicing [CR09]; it slices events based on the runtime information bound to them. When receiving a processed event from the vector clock algorithm, before sending the event to the monitor, the checker checks if the event is ordered with the events from its slice (as per the dependency relation). If it detects missing orderings, warnings are issued along with event names and their location in the code.

7.8 Related Approaches

In this section, we discuss *property-based* dynamic verification techniques for concurrent programs that rely on traces. More specifically, on techniques developed for monitoring behavioral properties expressed in total order

formalisms and refer to [BMP18] for a detailed survey. These techniques typically analyze a trace to either *detect* or *predict* violations.

Detection techniques reason about single runs of a program. We mention runtime monitoring tools, namely Java-MOP [CR05a], Tracematches [AAC+05a, BHL+10], MarQ [RCR15b], and LARVA [CPS09b] chosen from the RV competitions [FNRT15, RHF16, BFB+17]. These tools support various specification formalisms such as past-time linear temporal logic, context-free grammars, finite-state machines, extended regular expressions, and Quantified Event Automata (QEA) [BFH+12]. Detection techniques do not establish causal orderings between events and rely on trace collection approaches (discussed in Section 7.2.1) to order the collected events. We have shown in this paper how this can produce unsound traces leading to unsound and inconsistent monitoring. These tools can benefit from concurrent traces to guarantee the soundness of their verdicts. EnforceMOP [LR13] for instance, can be used to detect and enforce properties (deadlocks as well). It controls the runtime scheduler and blocks threads that might cause a property violation, sometimes leading to a deadlock. It requires forced atomicity as the scheduler needs to decide at each step if the execution on some thread continues or not. In [CF16, ACF+22], the authors present a monitoring framework for actor-based systems. The tool detectEr monitors Erlang applications using traces collected using the native logging functionality. Our approach targets generic concurrency primitives and can also be used with actor-based systems.

Predictive techniques reason about all feasible interleavings from a recorded trace of a single execution. Their challenge is to construct sound and maximal causal models [SCR12] that allow exploring flexibly all feasible interleavings. In [RS04], the authors present an instrumentation algorithm based on vector clocks, used in [SRA03, JS08, CSR08] for generating the partial order from a running program. The algorithm maintains one vector clock for each thread and two for each shared variable. It executes synchronously with the executing program and is protected using *synchronized* blocks to force an overall sequentially consistent [Lam79] execution. Vector clock algorithms typically require synchronization between the advice, program actions, and algorithm's processing to avoid data races [CL02]. Our algorithm can run synchronously or asynchronously with the program depending on the monitoring scenario. As far as we know, it is unique in the context of online monitoring in establishing order off the critical path without the need to block the execution to process. In [JS08, GZC+11] the work targets type-state errors. jPredictor [CSR08] for instance, uses sliced causality [CR07] to prune the partial order such that only relevant synchronization actions are kept. The tool is demonstrated on atomicity violations and data races; however, we are not aware of an application in the context of generic behavioral properties.

In [FMRS12], the authors present ExceptioNULL that target null-pointer exceptions. Violations and causality are represented as constraints over actions, and the feasibility of violations is explored via an SMT constraint solver. GPredict [HLR15], for instance, targets generic concurrency properties. It allows the user to express properties with regular expressions and provides explicit concurrency idioms such as atomic and parallel regions. It establishes order by collecting thread-local traces and also producing constraints over actions. In addition to being incomplete due to the possibility of not getting results from the constraint solver, the analysis from these tools might also miss some order relations between events resulting in false positives. Of course, none of the presented predictive techniques are complete, i.e., can produce all possible feasible interleavings that lead to violations, due to the impossibility of constructing a complete causal model of the program. Furthermore, these techniques reason on sequentially consistent execution models [HS12a], restricting the space of possible interleavings of programs. The idea is that if a property is violated in a sequential consistency, then it will surely be violated in a more relaxed execution model. Our work focuses on providing concurrent traces for online detection techniques, and we have yet to explore their applicability in predictive contexts. Unfortunately, many tools from the mentioned approaches [HLR15, CSR08, SRA03, JS08, CSR08] are not available. Apart from tools dedicated to data races and atomicity violation detection, we found no available tools targeting general behavioral properties to compare with; tools that can establish causal order and instrument various custom regular actions from the program for monitoring.

7.9 Conclusion

We presented a general approach for defining and collecting traces of concurrent programs for the online monitoring of behavioural properties. We investigated the limitations of linear traces and showed when they lead to inconsistent and unsound verdicts. We established on the fly the causal ordering of events using a novel vector clock algorithm that does not require blocking the execution. For monitoring frameworks relying on totally ordered traces, we redefined the monitorability of concurrent traces to avoid unsound verdicts. We implemented our approach within

BISM to collect traces from JVM programs.

CHAPTER 8

Opportunistic Monitoring

Contents

8.1	**Introduction**	**107**
8.2	**Opportunistic Runtime Verification**	**108**
8.3	**Implementation**	**110**
8.4	**Conclusion**	**111**

Chapter abstract

In this chapter, we present the opportunistic monitoring approach for the online monitoring of multithreaded programs leveraging existing runtime verification (RV) techniques. In the previous chapter, we presented an approach to collect representative traces from concurrent executions. However, the cost of capturing synchronization actions and establishing causality between events is expensive. In this new setting, local monitors are deployed to monitor specific threads and communicate verdicts with a global monitor when reaching selected synchronization points such as locks. At the cost of a small delay in the verdict, this approach reduces additional overhead and interference with the program to synchronize monitors. In this chapter, we motivate the approach, provide a summary of the approach, and present how it is implemented with BISM.

8.1 Introduction

Motivation. Our focus is still on monitoring general behavioral properties targeting violations that cannot be traced back to classical concurrency errors such as data races and atomicity violations. In the previous chapter, we presented an approach to collect representative traces of concurrent executions. These traces can be used by existing runtime verification tools that rely on total order formalisms to soundly check properties. However, although we presented a non-blocking algorithm that can also run asynchronously, the cost of capturing synchronization actions and establishing causality between events is still expensive.

We aim to monitor concurrent programs using existing approaches while reducing the overhead of instrumentation. Within these existing approaches, two monitoring modes can be currently utilized to handle concurrency: *per-thread* and *global* monitoring. In per-thread monitoring, each thread is monitored in isolation, and the monitors are oblivious to events from other threads. Whereas in global monitoring, a global monitor is spawned and receives all events from all threads. This monitor is guarded by a lock to ensure that no two threads can send events concurrently.

EXAMPLE 38 (GLOBAL MONITORING) Figure 8.1 illustrates a high-level view of a concurrent execution fragment of *1-Writer 2-Readers*, where a writer thread writes to a shared variable, and two other reader threads read from it. The reader threads share the same lock and can read concurrently once one of them acquires it, but no thread can write nor read while a write is occurring. We only depict the read/write events and omit lock acquires and releases for brevity. In this execution, the writer acquires the lock first and writes (event 1), then after one of the reader threads acquires the lock, they both concurrently read. The reader performs 3 reads (events 2, 4, and 5), and the other reader performs 2 reads (events 3 and 6), after that the writer acquires the lock and writes again (event 7). A user may be interested in the following behavioral property: *"Whenever a writer performs a write, all readers must at least perform one read before the next write"*. Note that the execution here has no data races nor a deadlock, and techniques focusing on generic concurrency properties are not suitable for the property. Monitoring of this (partial) concurrent execution with both previously mentioned modes presents restrictions. For *per-thread* monitoring, since each of the readers is a thread, and the writer itself is a thread, it cannot check any specification that refers to an interaction between them. For *global* monitoring, it imposes an additional lock operation to send each read event to the monitor, introducing additional synchronization and suppressing the concurrency of the program.

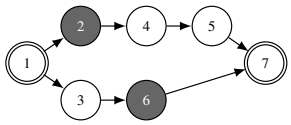


Figure 8.1: Execution fragment of *1-Writer 2-Readers*

In many cases, by reasoning about the concurrent program or making certain assumptions, one can optimize the monitoring process. A central observation we made is that when the program is free from generic concurrency errors such as data races and atomicity violations, a monitoring approach can be opportunistic and utilize the available synchronization in the program to reason about high-level behavioral properties. In the previous example, we know that reads and writes are guarded by a lock and do not execute concurrently (assuming we checked for data races). We also know that the relative ordering of the reads between themselves is not important to the property as we are only interested in counting that they all read the latest write. As such, instead of blocking the execution at each of the 7 events to safely invoke a global monitor and check for the property, we can have thread-local observations and only invoke the global monitor once either one of the readers acquires the lock or when the writer acquires it (only 3 events).

Methodology. We present an approach to opportunistic runtime verification which was first introduced in [EH18]. We aim to (i) provide an approach that enables users to arbitrarily reason about concurrency fragments in the program, (ii) be able to monitor properties *online* without the need to record the execution, (iii) utilize the existing tools and formalism prevalent in the RV community, and (iv) do so efficiently without imposing additional synchronization making it suitable for production environments. This proposes a two-level monitoring technique that allows for the reuse of existing runtime verification tools. This approach minimizes computational overhead

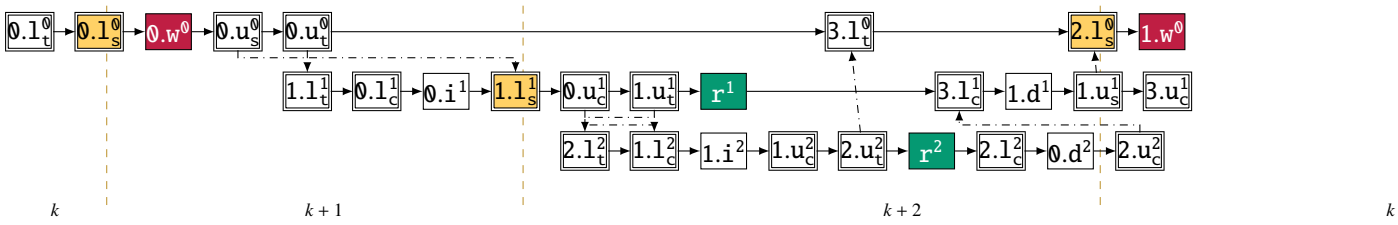


Figure 8.2: Concurrent execution fragment of 1-Writer 2-Readers.

and interference by eliminating the need for additional synchronization or the capture of program synchronization actions. However, this method comes at the cost of adding a small delay to determine the verdict as the monitor waits for the program to reach a synchronization point to check the property. The proposed approach is based on the following key ideas:

1. **Thread-Local Monitoring**: The first level involves thread-local monitoring, where events within each thread are totally ordered. This allows for the capture of properties specific to individual threads.
2. **Scope Monitoring**: The second level introduces the concept of *scopes*, which are regions in the program that are guaranteed to follow a total order. These scopes are defined based on existing synchronization actions in the program. A scope monitor aggregates the results of all thread-local monitors to perform global monitoring upon reaching these scopes.

Our method is based on certain assumptions about the synchronization of the program. These assumptions are reasonable for many concurrent programs, but may not hold for all programs. One of these assumptions is that the selected synchronization actions are guaranteed to follow a total order. Moreover, scopes are assumed to execute atomically at runtime.

The proposed approach enables more effective specification of behavioral properties by allowing reasoning about specific concurrency fragments in the program. This is in contrast to existing methods that are limited to either per-thread or global properties. It efficiently captures both local and global properties without imposing additional computational overhead.

Contributions. This work was first presented in [EH18]. In this chapter, we summarize the approach and present the implementation of the approach within BISM. In this thesis, we implement this approach within BISM which allows us to capture various concurrency constructs in JVM languages. We also provide a more comprehensive evaluation (see Section 9.6).

Chapter organization. The chapter is organized as follows. Section 8.2 present a summary with examples of the approach. Section 8.3 presents the implementation of the approach. Section 8.4 concludes the chapter.

8.2 Opportunistic Runtime Verification

In this section, we will summarize the opportunistic RV approach.

Thread types and events. The approach first differentiates threads based on the type of events they produce. Since an execution has a dynamic number of threads with numerous instructions, we introduce the notion of thread types, denoted by $\mathbb{T}$. A thread type converts relevant instructions into property events.

EXAMPLE 39 (EVENTS.) Consider the execution of the *readers-writers* program from Example 13. Depicted in Figure 8.2 is a fragment of the execution, where a writer thread writes to a shared variable, and two other reader threads read from it. Here we have two thread types: $\mathbb{T}_{rw} = \{\text{reader}, \text{writer}\}$. The events of interest are $\mathbb{E}_{\text{reader}} = \{\text{read}\}$ and $\mathbb{E}_{\text{writer}} = \{\text{write}\}$ corresponding to the green boxes in the figure.

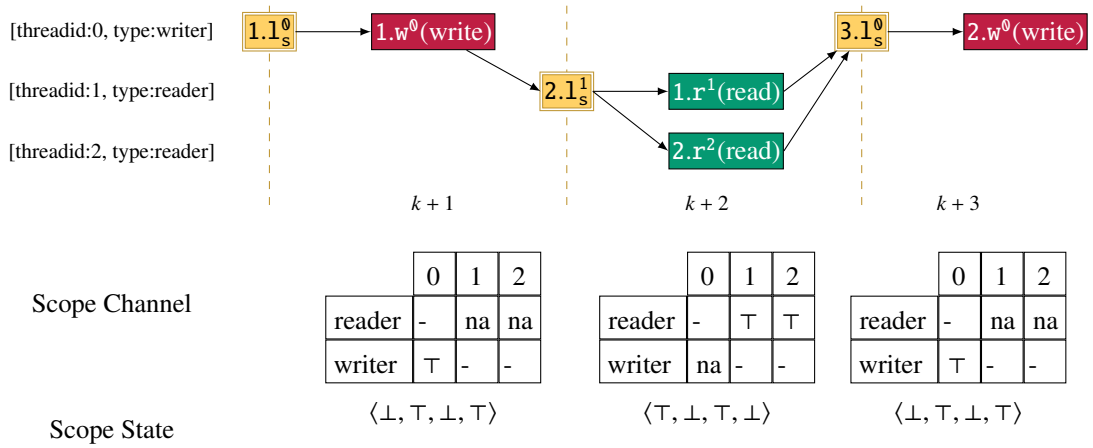


Figure 8.3: Example of a scope channel for 1-Writer 2-Readers.

Scopes. Within a concurrent execution, instructions in a given thread follow a total order while across threads, instructions are interleaved. Threads synchronize with each other through synchronization actions, such as locks and barriers. The opportunistic approach uses this synchronization to introduce the notion of a *scope* which allows us to reason over concurrent regions rather than instructions. A scope is a project of the concurrent execution into regions of interest. A scope is associated with a *synchronizing predicate* that delimits the scope region.

EXAMPLE 40 (SCOPE REGIONS) For the *readers-writers* program execution in Figure 8.2, we pick the resource service lock, denoted by $\mathtt{s}$, as scope delimiter. The scope region $\mathcal{R}_s(k+1)$ includes actions between $1.1^{\emptyset}_{s}$ and 2.1^{1}_{s}, i.e., $\mathcal{R}_s(k+1) = \{1.w^{0}, 1.u^{\emptyset}_{s}, 1.u^{\emptyset}_{t}, 2.1^{1}_{t}, 1.1^{1}_{c}, i^{1}\}$. The scope region $\mathcal{R}_s(k+2)$ contains two concurrent reads: $1.r^{1}, 2.r^{2}$. *

For reads and writes checking that the program is data race free is sufficient. However, for methods calls for instance, exisiting tools such as [HLR15, FF04] can be further utilized to guarantee the atomicity of these regions. Moreover, it is assumed that the program under consideration is properly synchronized and free from data races, as the methodology relies on existing synchronization mechanisms to define scope regions.

In our setup, the selection of a synchronizing predicate is manual and part of the specification. This user-defined nature allows for the customization of the scope based on the specific properties that are of interest. Given a property, the user-defined scope regions should delimit events whose order is significant for that property. For instance, for a property specifying that *"between each write, at least one read should occur"*, the scope regions should separate read and write events. The granularity of these regions, particularly when multiple threads are involved, is determined by the user's choice of the synchronizing predicate. Analyzing the program to automatically find and suggest suitable scopes for the user is an interesting but non-trivial challenge. This complexity arises from the need to understand the semantics of the program and the properties to be verified, which may require sophisticated static or dynamic analysis techniques.

Local properties. Within a scope region, properties are checked locally on each thread. A thread-local monitor checks a local property independently for each thread. These properties can be likened to *per-thread* monitoring. For any given thread, there is an established total order for all local events. As such the local properties are compatible with existing Runtime Verification (RV) techniques.

EXAMPLE 41 (LOCAL PROPERTIES) A local property, defined for the thread type reader, specifies that a reader should execute at least one read event. This requirement can be formulated using the classical LTL_3 [BLS11b] as $\varphi_{1r} \Leftrightarrow \mathbf{F}(\text{read})$. Similarly for the thread type writer, a local property specifies that a writer should execute at least one write event, i.e., $\varphi_{1w} \Leftrightarrow \mathbf{F}(\text{write})$. Figure 8.3 shows the projection of the execution and for each thread the result of evaluating the local properties over scope region. The evaluation of the local properties is recorded in the scope channel which we will discuss in the following section.

Scope state and scope properties. Now, the approach monitors per-thread local specifications based on thread type and aggregates the results for each concurrency region to construct a scope state. The scope monitor evaluates a sequence of such states which we call the scope trace.

EXAMPLE 42 (SCOPE PROPERTIES) Utilizing LTL, we formalize a set of three scope properties. These are derived from the scope states and utilize the alphabet {activereader, activewriter, allreaders, onewriter}:

- Mutual exclusion for readers and writers: $\varphi_0 \coloneqq$ activewriter **XOR** activereader.
- Exclusive writes: $\varphi_1 \coloneqq$ activewriter $\rightarrow$ onewriter.
- All readers process a written value: $\varphi_2 \coloneqq$ activereader $\rightarrow$ allreaders.

The global specification is thus defined as: $\mathbf{G}(\varphi_0 \wedge \varphi_1 \wedge \varphi_2)$. The scope trace $\langle \bot, \top, \bot, \top \rangle \cdot \langle \top, \bot, \top, \bot \rangle \cdot \langle \bot, \top, \bot, \top \rangle$, adheres to the given specification which is depicted in Figure 8.3. *

8.3 Implementation

In this section, the implementation of the approach is detailed, beginning with the communication of the monitor verdicts, followed by how we express properties with existing tools, and then covering instrumentation.

Scope channel. A *scope channel*, depicted in Figure 8.3, retains information of the local thread properties evaluation to create the scope state during execution. Each scope is linked to a unique scope channel with a distinct timestamp. This channel offers each thread-local monitor a dedicated memory slot to record its evaluation of local properties. Threads are restricted to write only in their allocated slots in the channel. The channel's timestamp is accessible to all threads within the scope, but only the scope monitor can increment it. For a given timestamp t, local monitors can no longer write information for any scope state that has a timestamp less than t. This ensures consistency of these states for any monitor associated with the scope.

Thread-local monitors. Each thread-local monitor is tasked with monitoring a specific local property for a particular thread. Threads are identified by unique identifiers and types. Depending on the properties to be checked, multiple monitors may be present on a single thread. These monitors are initiated at the thread's creation. They receive an event, conduct a check, and may record their result in the corresponding scope channel at the current timestamp.

Scope monitors. Scope monitors are in charge of verifying properties at the scope level. When a thread involved in the scope reaches a synchronizing action, it activates the scope monitor. This monitor depends on the scope channel, shared among all threads, for access to all observations. It may also use additional shared memory for its own state. The scope monitor is engaged atomically after a scope synchronizing action occurs. It first constructs the scope state from the thread-local monitors' results in the scope channel. Then, it executes the verification procedure on this state. Finally, it increases the timestamp of the scope channel before completing.

Specification. We will emphasize using the properties discussed in the previous section. Locally, eventual read and write are checked (represented as **F**(`read`) and **F**(`write`)), followed by counting the participating threads. This leads to forming atomic propositions such as `activereader`, `activewriter`, `allreaders`, and `onewriter`, indicating the presence of at least one reader, at least one writer, an equal number of reads to reader threads, and exactly one writer performing a write, respectively. The scope properties are defined within the same scope, which alternates between readers and writers.

Listing 8.1 shows how the opportunistic monitoring is specified by the user. The keyword *selectors* (Lines 1-3) is used to define the synchronizing predicate. The predicate AR is evaluated by identifying an action marked REENTRANTLOCK.ACQUIRE and align its context with the *lockname* key. The thread types are specified using the *types* keyword (Lines 5-14). For each type, the initiating spawn event is pinpointed (Lines 7, 11), followed by the identification of all other relevant events (Lines 8, 12). Here, the actions of reading and writing are distinguished based on their respective action labels. Following the definition of thread types, the process of identifying scopes is undertaken (Lines 16-18).

```
1  selectors {
2    event AR on "ReentrantLock.Acquire" when "%lockname == SharedArr.resourceLock"
3  }
4  types{
5    reader {
6      spawn on "Reader.Run"
7      event read on "SharedArr.read"
8    }
9    writer {
10     spawn on "Writer.Run"
11     event write on "SharedArr.write"
12   }
13 }
14 scope s0 (AR) {
15   property s0r on reader(read) is "LTL=F(read)"
16   property s0w on writer(write) is "LTL=F(write)"
17
18   atom activereader : count(s0r, T) > 0,
19   atom activewriter : count(s0w, T) > 0,
20   atom onewriter : count(s0w, T) == 1,
21   atom allreaders : count(s0r, T) == count(reader)
22
23   check "LTL=G(
24        (activereader XOR activewriter) && (activewriter => onewriter )
25              && (activereader => allreaders)
26     )"
27 }
```

Listing 8.1: Readers-writers specification.

A singular scope, designated as `s0`, is introduced, with its synchronizing predicate clearly defined as AR (Line 14). The next step involves determining local properties, which entails naming each property (Lines 15-16) and specifying the thread type and applicable events for each. After defining the local property, atomic propositions essential for the scope state are developed (Lines 18-21). This is achieved by employing the `count` function, which calculates the number of monitors across multiple threads that have returned a specific verdict for a local property. Utilizing `count` to aggregate the outcomes of local properties facilitates the formation of necessary atomic propositions for the scope state. The final step involves the introduction of the scope property (Lines 23-26), by using the atomic propositions defined in the scope.

Instrumentation. The opportunistic approach is implemented using the BISM framework. In the previous chapter, we discussed how we used BISM to instrument the program and capture concurrent traces. We provided a pre-defined set of transformers for the most common concurrency primitives. This library allows us to capture the synchronization actions of interest for the scope in the opportunistic setup. There we insert hooks to invoke the scope monitor when a scope synchronizing action is reached. It also allows us to capture thread-spawning events. Property events are also captured using regular transformers and hooks to the thread-local monitors are inserted. Moreover, thread-spawning events are also captured using regular transformers. Monitors are synthesized by passing the specified properties into the LamaConv [Ins], a tool employed for monitor synthesis.

8.4 Conclusion

This chapter represented the opportunistic approach for the online monitoring of multithreaded programs leveraging existing runtime verification (RV) techniques. We motivated this approach and presented a summary of the approach. Given the appropriate assumptions on the execution such as data race freedom, and atomicity of scope regions, which can be checked using existing tools, the opportunistic approach can be used to monitor concurrent programs. Moreover, by decentralizing the specification over local and global monitors, various interesting behavioral properties can now be checked which was not possible before. We then presented how this approach can be implemented so that the program can be instrumented to communicate with the local and global monitors relying on the BISM framework.

Part IV

Evaluation and Use Cases

CHAPTER 9

Evaluation

Contents

9.1	**Introduction**	**115**
9.2	**Evaluating BISM**	**116**
9.2.1	Methodology	116
9.2.2	Advanced Encryption Standard (AES)	117
9.2.3	Financial Transaction System	118
9.2.4	DaCapo Benchmarks	121
9.2.5	Threats to Validity	122
9.3	**Evaluating the BISM DSL**	**122**
9.3.1	Performance Evaluation	123
9.3.2	User Experience Evaluation	123
9.4	**Evaluating the Residual Analysis**	**124**
9.5	**Evaluating Concurrent Traces**	**125**
9.5.1	Effectiveness and Cost	125
9.5.2	Causal Dependence Relation in Specification Patterns	128
9.6	**Evaluating the Opportunistic Monitoring**	**129**
9.6.1	Readers-Writers	129
9.6.2	Other Benchmarks	130
9.7	**Conclusion**	**132**

9.1 Introduction

In this chapter, we empirically validate the theoretical and engineering contributions made in this thesis. The evaluation covers assessing the performance and scalability of the proposed methods and tools, as well as their expressiveness and ease of use.

Below is an overview of the chapter structure.

- In Section 9.2, we evaluate BISM to quantify the tool's performance and expressiveness, utilizing a set of benchmarks for validation. We compare it with other state-of-the-art tools, namely DiSL and AspectJ (see Section 5.7), to assess its performance and scalability. We assess the performance of the instrumented

programs by each tool, comparing load-time and compile-time instrumentation scenarios and measuring the overhead introduced by each tool.

- In Section 9.3 is dedicated to the assessment of the domain-specific language developed for specifying instrumentation. We evaluate the ease of use, expressiveness, and efficiency of the generated instrumentation code comparing it to AspectJ.
- In Section 9.4, we focus on the evaluation of our residual runtime verification approach presented in Chapter 6. We measure the reduction in instrumentation points and the corresponding impact on runtime performance. This is assessed through a set of benchmarks that test the approach's scalability and effectiveness.
- In Section 9.5 evaluates our contributions to the monitoring of concurrent programs. We assess the soundness and faithfulness of the collected concurrent traces and the efficiency of our real-time vector clock algorithm presented in Chapter 7. The evaluation is performed using both synthetic benchmarks and real-world applications.
- In Section 9.6 provides an evaluation of the opportunistic monitoring approach presented in Chapter 8. We focus in our evaluation on the performance of the approach and its ability to monitor multithreaded programs without introducing additional synchronization or significantly affecting program behavior.
- In Section 9.7 we conclude the chapter.

The results are intended to validate the proposed methods and tools, thereby contributing to their credibility and potential for adoption in the field of runtime verification.

9.2 Evaluating BISM

We evaluate our instrumentation framework BISM that we presented in Chapter 5. We compare it with other state-of-the-art tools, namely DiSL and AspectJ (see Section 5.7), to assess its performance and scalability. We assess the performance of the instrumented programs by each tool, comparing load-time and compile-time instrumentation scenarios and measuring the overhead introduced by each tool.

9.2.1 Methodology

Experiments and used programs. We compare BISM with DiSL and AspectJ in three different experiments. Table 9.1 illustrates how the three experiments are complementary to each other. We used the latest versions of DiSL [1] and AspectJ Weaver 1.9.4.

- The first experiment concerns the implementation of the Advanced Encryption Standard (AES). This experiment shows how BISM can perform inline instrumentation by inserting new bytecode instructions inside the target program to detect test inversion attacks on the application.
- The second experiment concerns a financial transaction system. This experiment shows how BISM can be used to instrument the system to monitor user-provided properties. The financial transaction system is a relatively small application with a low event rate.
- The third experiment concerns the DaCapo benchmark [BGH+06a]. This experiment shows how BISM can be used to instrument the benchmark and monitor for the good usage of data structures (with classical properties: **HasNext**, **UnSafeIterator**, and **SafeSyncMap**). DaCapo is a large benchmark classically used when evaluating runtime verification tools as it produces events at a high rate.

For the first experiment, instrumentation is performed at the level of the control-flow graph. For the two other experiments, the instrumentation is performed at the level of method calls to emit events. Note, AspectJ is not capable of instrumenting for inline monitoring of control-flow events, so we do not include it in the first experiment (with AES).

We run our experiments in both of BISM instrumentation modes, namely load-time and build-time. Running an experiment in load-time mode serves to compare the performance when the tools act as an interface between

[1] From https://gitlab.ow2.org/disl/disl.

the base program and the virtual machine. Running an experiment in build-time mode serves to compare the performance of the generated instrumented bytecode.

We note that DiSL wraps its instrumentation code with exception handlers. Exception handlers are not necessary for our experiments and have a performance impact. To guarantee fairness, we switched off exception handlers in DiSL.

Evaluation metrics. We consider three performance metrics: runtime, used memory, and bytecode-size. We are interested in evaluating the instrumentation overhead, that is, the performance degradation caused by instrumentation. For each metric, we use the base program as a baseline. For runtime, we measure the execution time of the instrumented program. For used memory, we measure the used heap and non-heap memory after a forced garbage collection. In load-time mode, we do not measure the used memory in the case of DiSL because DiSL performs instrumentation on a separate JVM process.

Evaluation environment. To run the experiments, we use Java JDK 8u251 with 2 GB maximum heap size on an Intel Core i9-9980HK (2.4 GHz. 8 GB RAM) running Ubuntu 20.04 LTS 64-bit. We consider 100 runs and then calculate the mean and the standard deviation.

In what follows, we illustrate how we carried out our experiments and the obtained results[2].

Table 9.1: A comparison between the experiments. LT is for load-time mode, and BT is for build-time mode. A checkmark (✓) indicates that the experiment involves the metric or the feature, whereas a cross mark (✗) indicates that the experiment does not involve the metric or the feature. Term NA abbreviates Not Applicable, and (-DiSL) indicates that the DiSL tool has been excluded.

		Performance Metrics			Instrumentation Level	Bytecode Insertion	Comparison with	
		Runtime	Used Memory	Bytecode Size			AspectJ	DiSL
AES	LT	✓	✓ (-DiSL)	NA	Bytecode Level	✓	NA	✓
	BT		✓	✓				
Transactions	LT	✓	✓ (-DiSL)	NA	Source code Level	✗	✓	✓
	BT		✓	✓				
DaCapo	LT	✓	✓ (-DiSL)	NA	Source code Level	✗	✓	✓
	BT		✓	✓				

9.2.2 Advanced Encryption Standard (AES)

Experimental setup. We compare BISM with DiSL in a scenario using inline monitors. We instrument an external AES implementation to detect test inversions in the control flow of the program execution. The instrumentation deploys inline monitors that duplicate all conditional jumps in their successor blocks to report test inversions. We implement the instrumentation as follows.

In BISM, we use built-in features to duplicate all conditional jumps utilizing the ability to insert raw bytecode instructions. In particular, we use the instrumentation selector `beforeInstruction` to capture conditional jumps. To extract the opcode for each conditional jump, we use the static context object `Instruction`, and to duplicate the operand values on the stack, we use the advice method `insert`[3]. We then use the control-flow instrumentation locators[4] to capture the successor blocks executing after every conditional jump. Finally, at the beginning of these blocks, we utilize `insert` to duplicate the conditional jump instruction. A transformer that detects test inversions and reinforces the control flow integrity can be found in Section 10.6.3.

In DiSL, we implement a custom `StaticContext` object to retrieve information from conditional jump instructions, such as the indices of jump targets and instruction opcodes. Note, we use multiple `BytecodeMarker`

[2]More details about the experiments and the material needed to reproduce them can be found at https://gitlab.inria.fr/bism/bism-experiments.

[3]Extracting stack values can be also alternatively achieved using dynamic context method `getStackValue` and adding new local variables.

[4]`OnTrueBranchEnter`, `onFalseBranchEnter`.

Table 9.2: Number of emitted events in AES experiment.

Plain-text size (KB)	2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8
Events (M)	0.9	1.8	3.6	7.3	14.9	29.5	58.5	117	233

snippets to capture all conditional jumps. To retrieve stack values, we use the dynamic context object. We then store the extracted information in synthetic local variables, and we add a flag to specify that a jump has occurred. Finally, on successor blocks, we map opcodes to Java syntax to re-evaluate conditional jumps using switch statements.

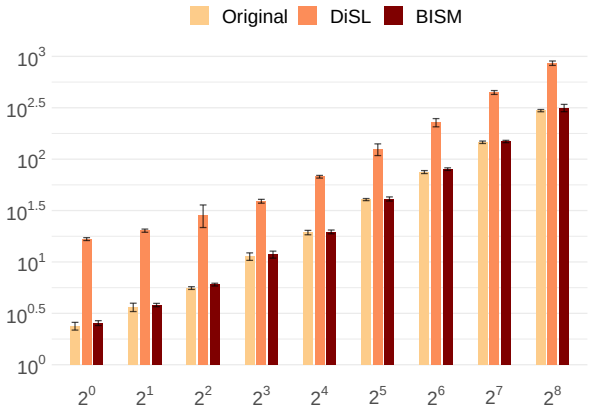


Figure 9.1: AES load-time instrumentation runtime (ms).

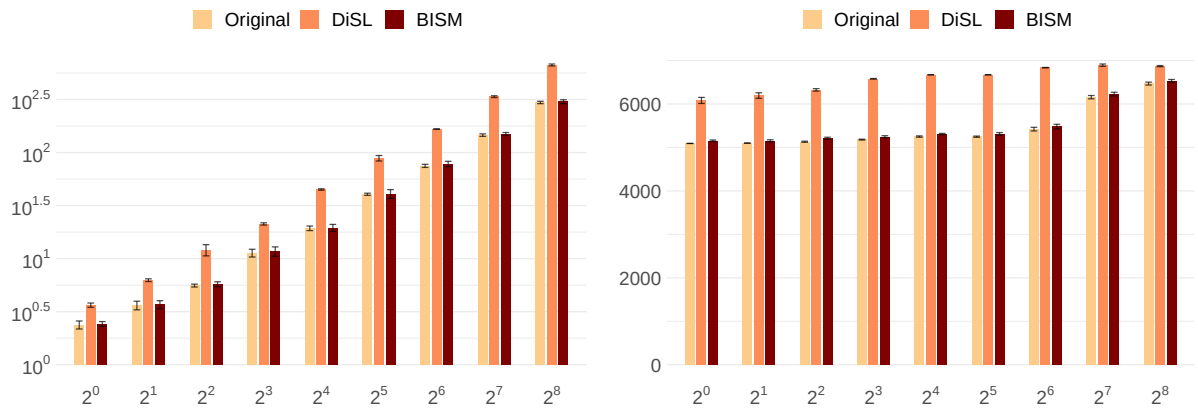
Load-time evaluation. We consider different sizes of the plain text to be encrypted by AES. Figure 9.1 reports runtime with respect to plain-text size, in load-time mode. BISM shows better performance over DiSL for all plain-text sizes. We do not measure the used memory because DiSL performs instrumentation on a separate JVM process which imposes a huge memory overhead. Also, AspectJ is excluded from this experiment as it cannot capture control-flow events.

Build-time evaluation. We replace the original classes of AES with independently instrumented classes from each tool. Figure 9.2 reports the runtime and used memory in build-time mode depending on plain-text size. BISM shows less overhead than DiSL in both runtime and used memory for all plain-text sizes. Moreover, BISM incurs a relatively small overhead for all plain-text sizes. Table 9.2 reports the number of generated events (corresponding to conditional jumps) after running the code (in millions). The bytecode size of the original AES class is 9 KB. After instrumentation, the bytecode size is 10 KB (+11.11%) for BISM, and 128 KB (+1,322%) for DiSL. So, BISM incurs less bytecode-size overhead than DiSL. The significant overhead in DiSL is due to the inability to inline the monitor in bytecode and having to instrument it in Java. We note that it is not straightforward in DiSL to extract control-flow information in Markers, whereas BISM provides this out-of-the-box.

9.2.3 Financial Transaction System

Experimental setup. We compare BISM with DiSL and AspectJ in a runtime verification scenario to monitor some properties of a financial transaction system. We use the implementation from CRV-14 [BFB+19a] to monitor the following properties:

- Property P1: Only users based in certain countries can be Silver or Gold users.
- Property P2: The transaction system must be initialized before any user logs in.
- Property P3: No account may end up with a negative balance after being accessed.
- Property P4: An account approved by the administrator may not have the same account number as any other already existing account in the system.



(a) Runtime (ms).

(b) Used memory (KB).

Figure 9.2: AES build-time instrumentation.

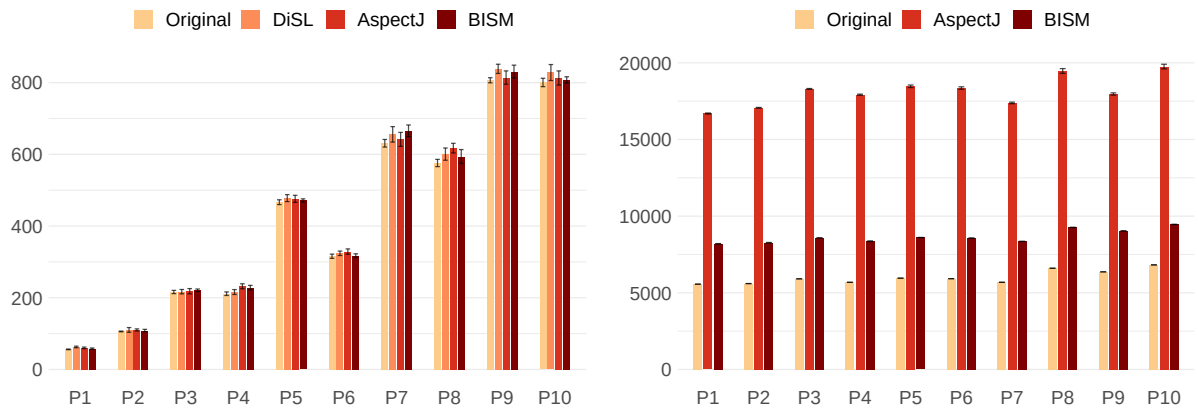
- Property P5: Once a user is disabled by the administrator, he or she may not withdraw from an account until being activated again by the administrator.
- Property P6: Once greylisted, a user must perform at least three deposits from external before being whitelisted.
- Property P7: No user may request more than 10 new accounts in a single session.
- Property P8: The administrator must reconcile accounts every 1000 external transfers or an aggregate total of one million dollars in external transfers.
- Property P9: A user may not have more than three active sessions at once.
- Property P10: Transfers may only be made during an active session (i.e., between a login and logout).

For each property using a set of events, we instrument the financial transaction system to generate those events. Such events mainly correspond on the system to method call with parameters or class field updates. For example, monitoring Property P6 requires the following events: `greylistUser(id)`, `depositFromExternal(id)` and `whitelistUser(id)`, where `id` is a unique user identifier. As test suite, we implement a custom set of scenarios that covers all of the above properties, and an external monitor library with stub methods that only count the number of received events. We implement instrumentation as follows:

- In BISM, we use the static context provided at method-call instrumentation selectors[5] to filter methods by their names and owners. To access the method calls' receivers and results, we utilize methods `getMethodArgs` and `getMethodResult` available in dynamic contexts. We then use argument processors and dynamic context objects to access dynamic values and pass them to the monitor. The extracted values are then passed to the monitor by invoking its appropriate method.
- In DiSL, we implement custom Markers to capture the needed method calls and use argument processors and dynamic context objects to access dynamic values. We note that it required to create a custom marker for each method call, which resulted in implementing 28 different marker classes.
- In AspectJ, we use the call pointcut, type pattern matching, and join point static information to capture method calls and write custom advices that invoke the monitor.

Load-time evaluation. Figure 9.3 reports the runtime and used memory for the considered properties in load-time mode (excluding DiSL in the case of used memory). BISM shows better performance over DiSL and AspectJ for properties P2, P5, P6, P8, and P10, for five properties out of ten. Whereas DiSL shows the best performance for P3 and P4, and AspectJ shows the best performance for properties P1, P7, and P9. The similar results of the tools is due to the fact that each property monitor augments the base program with a small number of advices at limited

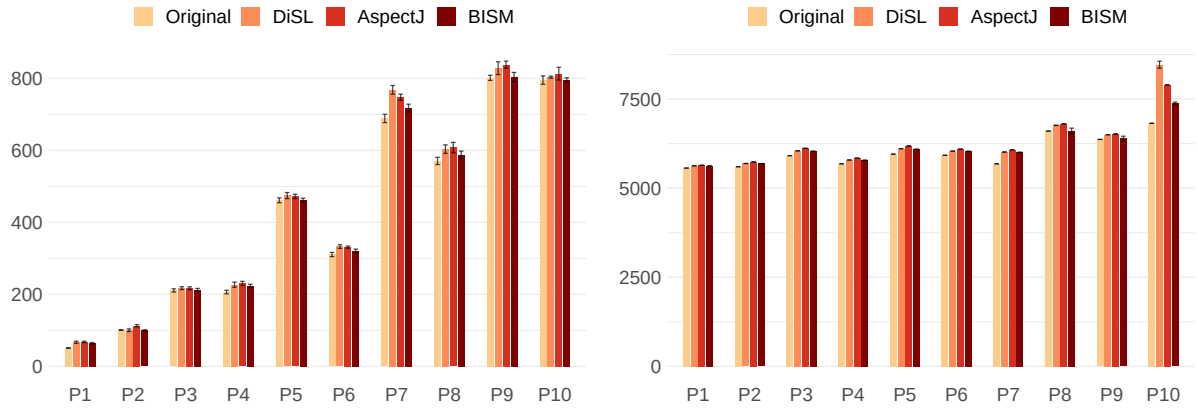
[5] `beforeMethodCall`, `afterMethodCall`.



(a) Runtime (ms).

(b) Used memory (KB).

Figure 9.3: Financial transaction system load-time instrumentation.



(a) Runtime (ms).

(b) Used memory (KB).

Figure 9.4: Financial transaction system build-time instrumentation.

locations, ranging between two and five advices per property. Hence, the results in load-time mode reflect the execution time of the woven advice more than the instrumentation overhead. Concerning used memory, BISM incurs much lower overhead than AspectJ for all properties.

Build-time evaluation. We replace the original classes of the scenarios with statically instrumented classes from each tool. Figure 9.4 reports the runtime and used memory for the considered properties in build-time mode. BISM shows less runtime and used-memory overheads than both DiSL and AspectJ for all properties. Table 9.3 reports the number of generated events after running the code (in thousands). The bytecode size of the classes of the overall original scenarios is 44 KB. After instrumentation, the bytecode size is 56 KB (+27.27%) for BISM, 84 KB (+90.9%) for DiSL, and 116 KB (+163.63%) for AspectJ. Hence, BISM incurs less bytecode-size overhead than both DiSL and AspectJ.

Table 9.3: Number of events generated by the financial transaction system for each monitored property.

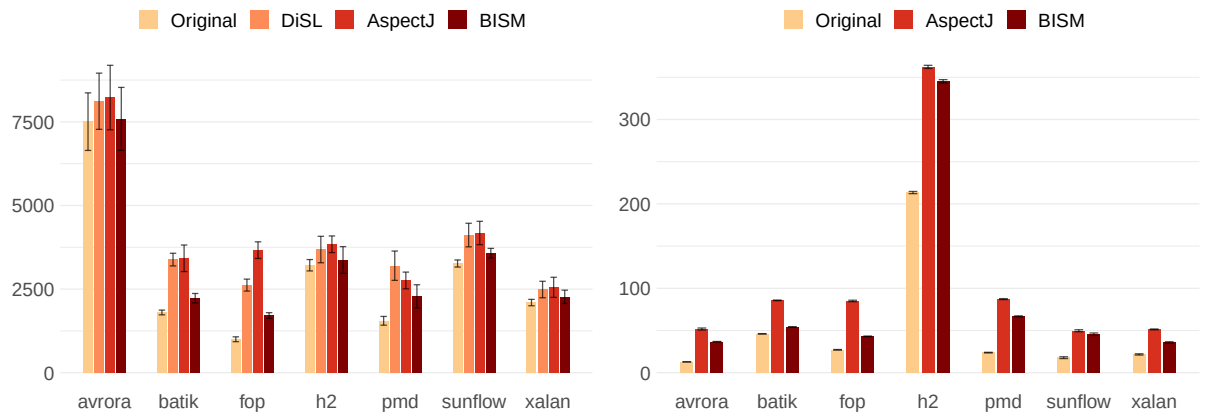
Property	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
Events (k)	10	10	11	33	71	69	125	104	192	154

9.2.4 DaCapo Benchmarks

Experimental setup. We compare BISM with DiSL and AspectJ in a general runtime verification scenario. We instrument the benchmarks in the DaCapo suite [BGH+06a] (dacapo-9.12-bach), to monitor for the classical **HasNext**, **UnSafeIterator**, and **SafeSyncMap** properties. The **HasNext** property specifies that a program should always call method `hasNext()` before calling method `next()` on an iterator. The **UnSafeIterator** property specifies that a collection should not be updated when an iterator associated with it is being used. The **SafeSyncMap** property specifies that a map should not be updated when an iterator associated with it is being used. We only target the packages specific to each benchmark and do not limit our scope to `java.util` types; instead, we match freely by type and method name. We implement an external monitor library with stub methods that only count the number of received events. We implement the instrumentation similarly to the second experiment:

- In BISM, we use the static context provided at method-call instrumentation selectors to filter methods.
- In DiSL, we implement custom Markers to capture the needed method calls.
- In AspectJ, we use the call pointcut, type pattern matching, and join point static information to capture method calls.

We choose the following benchmarks: avrora, batik, fop, h2, pmd, sunflow and xalan. For each benchmark, Dacapo provides a small, default and large workload. We choose the default workload which includes a number of warm-up runs performed internally.



(a) Runtime (ms).

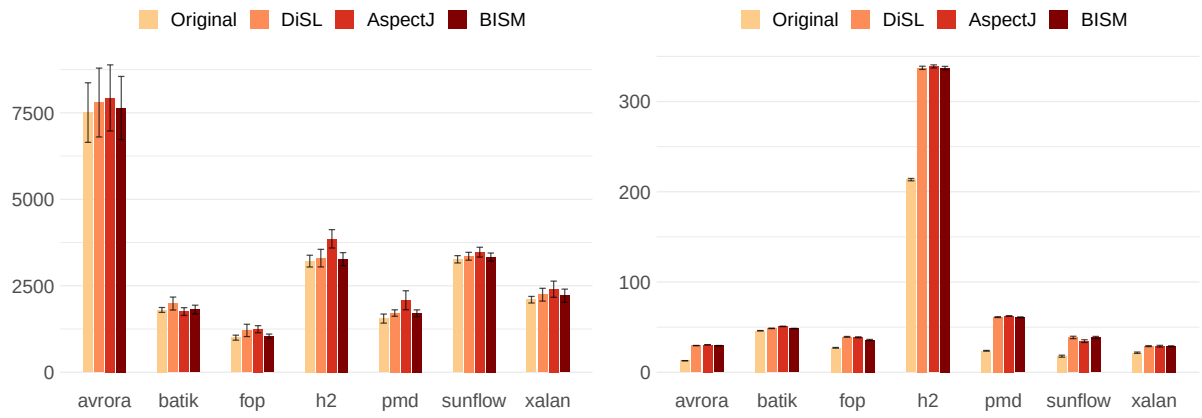
(b) Used memory (MB).

Figure 9.5: DaCapo load-time instrumentation.

Load-time evaluation. Figure 9.5 reports the runtime for the benchmarks. BISM shows better performance over DiSL and AspectJ in all benchmarks. DiSL shows better performance than AspectJ except for the pmd benchmark. For the pmd benchmark, this is mainly due to the fewer events emitted by AspectJ (see Table 9.5). We notice that AspectJ captures fewer events in benchmarks batik, fop, pmd, and sunflow. This is due to its inability to instrument synthetic bridge methods generated by the compiler after type erasure in generic types. BISM also shows less used-memory overhead over AspectJ in all benchmarks. Let us mention that we did not measure the used memory for DiSL since it performs instrumentation on a separate JVM process.

Build-time evaluation. We replace the original classes in the benchmarks with statically instrumented classes from each tool. Figure 9.6 reports the runtime and used memory of the benchmarks. BISM shows less runtime overhead in all benchmarks, except for batik where AspectJ emits fewer events. BISM also shows less used-memory overhead, except for sunflow, where AspectJ emits much fewer events.

Table 9.5 compares the instrumented bytecode. We report the number of classes in scope (Scope) and the instrumented (Instr.), and we measure the bytecode-size overhead (Ovh.) for each tool. We also report the number of generated events after running the code (in millions). BISM and DiSL generate the same number of events, while Aspect (AJ) produces fewer events because of the reasons mentioned above. The results show that BISM



(a) Runtime (ms).

(b) Used memory (MB).

Figure 9.6: DaCapo build-time instrumentation.

Benchmark	Scope	Instr.	Ref.	BISM		DiSL		AspectJ		Events (M)	
			KB	KB	Ovh.%	KB	Ovh.%	KB	Ovh.%	#	AspectJ
avrora	1,550	35	257	264	2.72	270	5.06	345	34.24	2.5	2.5
batik	2,689	136	1,544	1,572	1.81	1,588	2.85	1,692	9.59	0.5	0.4
fop	1,336	172	1,784	1,808	1.35	1,876	5.16	2,267	27.07	1.6	1.5
h2	472	61	694	704	1.44	720	3.75	956	37.75	28	28
pmd	721	90	756	774	2.38	794	5.03	980	29.63	6.6	6.3
sunflow	221	8	69	71	2.90	74	7.25	85	23.19	3.9	2.6
xalan	661	9	100	101	1.00	103	3.00	116	16.00	1	1

Table 9.4: For each benchmark in the DaCapo experiment, the table reports the number of classes in the scope of instrumentation (Scope), the instrumented classes (Instr.), the original (Ref.) and generated bytecode size and overhead per tool, and the number of emitted events, (#) for BISM and DiSL, and AspectJ separately.

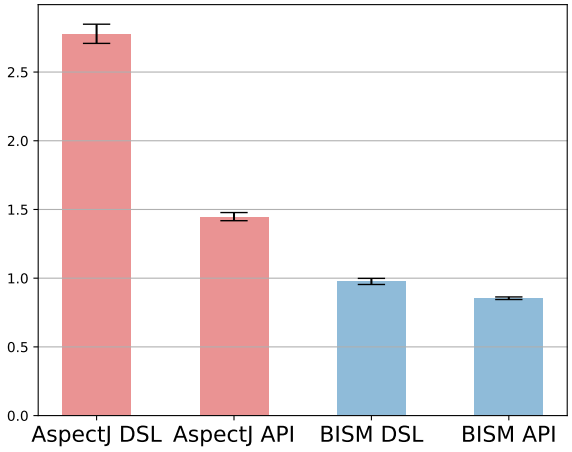
incurs less bytecode-size overhead for all benchmarks. We notice that even with exception-handlers turned off, DiSL still wraps a targeted region with try-finally blocks when the `@After` annotation is used. This guarantees that an event is emitted after a method call, even if an exception is thrown.

9.2.5 Threats to Validity

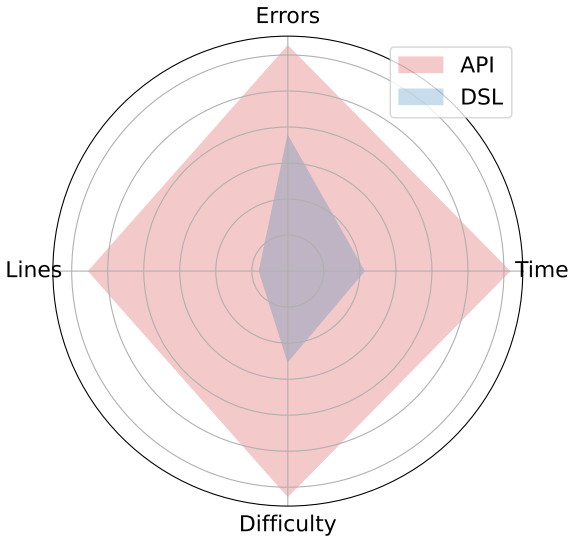
One of the threats to the validity of our experiments is the non-determinism of the DaCapo benchmarks: avrora, h2, pmd, sunfow, and xalan. We mitigate this by measuring the mean execution times over 100 runs. We also measure dispersion (variance and standard deviation) and report it in the graphs. Another threat to validity is the possible overhead caused by additional features found in DiSL and AspectJ. DiSL wraps its instrumented code with exception handling and a dynamic bypass check. We mitigate this by disabling any source of additional overhead in DiSL, to put it in the best configuration for performance comparison. Whereas, AspectJ generates additional classes that contain the implemented advice at each instrumentation point. However, there is nothing we can do as AspectJ is not customizable and not compatible with inline instrumentation.

9.3 Evaluating the BISM DSL

In this section, we provide an assessment of the DSL we presented in Chapter 5, specifically focusing on the overhead it introduces and the user experience. The full details for the experiments can be found at [SFc].



(a) Mean execution time in seconds.



(b) Comparison of API and DSL.

9.3.1 Performance Evaluation

For this evaluation, we compare the performance of the DSL with the BISM API and AspectJ approaches. We use the financial transaction system from [BFB+19a], instrumenting it to extract events from method calls, field operations, and method executions. We test four distinct approaches: (1) **AspectJ DSL**, in which an aspect (`.aj`) is written with the DSL; (2) **AspectJ API**, in which an aspect is written in Java syntax using annotations; (3) **BISM DSL**, our proposed DSL developed for BISM; and (4) **BISM API**, in which classical BISM transformers are written. In order to ensure a fair comparison, we used features that are common across all four approaches.

Each benchmark was run 20 times and the results are reported [6]. The results showed that BISM API outperforms the other methods, running 1.14 times faster than BISM DSL, 1.7 times faster than AspectJ API, and 3.25 times faster than AspectJ DSL. The performance results are illustrated in Figure 9.7a. This demonstrates that specifying instrumentation using the API generally leads to faster execution due to the extra delay incurred in DSL approaches from parsing the specification files. However, our proposed DSL outperforms AspectJ DSL and is even faster than AspectJ API, highlighting the effectiveness and efficiency of our DSL in the context of software monitoring and instrumentation.

9.3.2 User Experience Evaluation

Experiment design. We conducted an experiment in which 10 participants, with various experience levels in Java and runtime verification, were instructed to write transformers for monitoring four different properties sourced from [BFB+19b] using both methods. The experiment used a randomized block design. The participants were randomly divided into two equal groups: Group A and Group B. Group A used the API for Properties 1 and 3 and the DSL for Properties 2 and 4. In contrast, Group B used the DSL for Properties 1 and 3 and the API for Properties 2 and 4. This design allowed each participant to gain experience with both methods, enabling a fair comparison between the two techniques across all properties.

Collected metrics. For each property, the participants are asked to record the time they took to write the instrumentation, reported in minutes (**Time**). They also kept track of the number of mistakes they made during the process that necessitated recompilation and another run (**Errors**). In addition, the number of lines of code written for each transformer was noted, (**Lines**). Lastly, participants rated the difficulty of use on a scale of 1 to 5, where 1 signified very easy and 5 meant very hard, (**Difficulty**).

[6] The experiment utilized AspectJ AJC 1.9.7 and JDK 11.

Results and analysis. In Figure 9.7b, we report each metric normalized to a 0-1 range based on their respective minimum and maximum values. The results indicate a clear advantage of the DSL over the API for writing BISM transformers. Across all properties tested, the DSL not only needed significantly less time to implement but also resulted in fewer errors and less code. Furthermore, participants found the DSL easier to use. The good results of the DSL can be largely attributed to its concise syntax and straightforward usage. However, it is important to note that a considerable portion of errors in DSL mode were caused by improper referencing of the monitor and its package name, and the need for full specification of return types in patterns. Future iterations of the DSL will address these pattern specification issues.

9.4 Evaluating the Residual Analysis

We now report on our evaluation of the effectiveness of our approach. Full details can be found at [SFd].

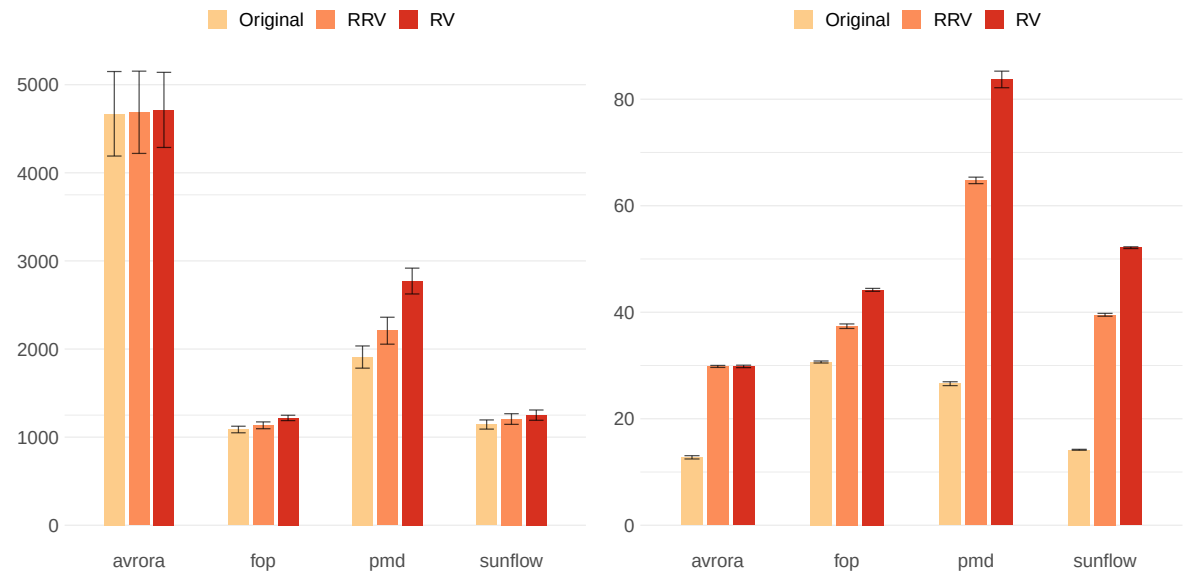
Experimental setup. We compare the instrumentation overhead with our residual analysis, denoted by **RRV**, and without the analysis, denoted by **RV**. We instrument with BISM the programs, in the DaCapo suite [BGH+06a], for the monitoring of the classical `SafeListIterator` (P1), `SafeMapIterator` (P2), and `SafeHasNext` (P3) properties. (P3) specifies that a program does not call the `next` method before calling the `hasNext` method of an iterator. We include as *escape* events (#) all assignments to class fields, all method calls that pass objects by references, and in addition, return statements that return objects [CGS+99]. We include in the *SafeList* all calls to methods of Java classes. Other than the method calls relevant to the property and captured by instrumentation, these calls do not produce events. We note that *fop* is the only single-threaded benchmark, however, we can use the multi-threaded benchmarks as we checked that for the properties all the events in the projected traces are being produced within the same thread. We consider 100 runs and then calculate the mean and the standard deviation[7].

Evaluation metrics. We consider the number of affected instructions, methods, and classes by our residual analysis (RRV) and without it (RV). We also consider the improvement factor. We are also interested in evaluating the runtime overhead, that is, the performance degradation caused by instrumentation for monitoring. For runtime, we measure the execution time of the instrumented program. For used memory, we measure the used heap and non-heap memory after a forced garbage collection.

Table 9.5: For each program (Bench), and property (P1), (P2), and (P3), we report # of relevant classes, methods, and instructions (Rel) producing events, number proved safe statically by our technique (Nop), # of events produced at runtime (RV) and after our analysis (RRV), improvement factor for # of instructions instrumented and events produced (Imp). K = 10^3, M = 10^6.

Bench	Property	# Classes		# Methods		# Instructions			# Events		
		Rel	Nop	Rel	Nop	Rel	Nop	Imp	RV	RRV	Imp
avrora	P1	41	14	99	56	165	86	2.09	1.36M	1.36M	1.00
fop	P1	123	33	275	103	700	210	1.43	729K	490K	1.49
sunflow	P1	11	2	35	15	50	15	1.43	2.55M	1.27M	2.00
pmd	P1	86	27	200	95	420	146	1.53	4.77M	778K	6.13
avrora	P2	41	19	111	78	160	117	3.72	353K	246K	1.43
fop	P2	100	28	206	85	2.9K	2.6K	9.19	545K	351K	1.55
sunflow	P2	11	6	32	24	40	26	2.86	2.55M	1.27M	2.00
pmd	P2	81	27	168	70	392	211	2.17	3.01M	2.6M	1.16
avrora	P3	32	11	76	33	160	79	1.98	1.5M	1.29M	1.16
fop	P3	70	7	145	31	376	67	1.22	1.07M	882K	1.21
sunflow	P3	8	2	12	3	29	3	1.12	3.93M	2.65M	1.48
pmd	P3	65	21	126	48	343	115	1.50	5.64M	5.23M	1.08

[7]We use Java JDK 8u251 with 16 GB maximum heap size on an Intel Core i9-9980HK (2.4 GHz. 16 GB RAM). We use the DaCapo version 9.12-bach.



(a) Execution time (ms).

(b) Used memory (MB).

Figure 9.8: Evaluation for the three properties.

Results. In Table 9.5, we report the results. The table demonstrates the effectiveness of the residual analysis as it reduces the number of instrumentation points by a factor of 2.5 on average (reaching 9.19), and accordingly, a reduction in the number of generated events at runtime by a factor of 1.8 on average (reaching 6.13). We notice that the reduction of instrumentation points does not always result in a reduction of runtime events for instance with *avrora* with (P1), where we find methods that produce most of the events that we could not prove safe statically. We also notice that most of our missed optimizations are due to escape # events (see Section 6.3.2). The more diverse operations between events, the more missed optimization. However, the *SafeList* can be improved with the help of escape analysis to include more instructions that we can guarantee are safe for our analysis and accordingly reduce the number of escape events. We leave that for future work as we envision adding plugins to incorporate static analysis. We also note that many of the events generated under classical instrumentation (RV) are irrelevant; they occur in methods that do not produce enough events to reach a final state in the monitor. As such, our analysis is effective in removing those from instrumentation. Figures 9.8 report the execution time and the memory usage for the benchmarks with all three properties combined. The figures show that **RRV** results in better performance in all benchmarks than classical instrumentation **RV**.

9.5 Evaluating Concurrent Traces

We demonstrate the effectiveness of our approach for capturing sound and faithful traces from Chapter 7. All experiments can be found in the artifact repository [SFa].

9.5.1 Effectiveness and Cost

We measure the effectiveness and cost of obtaining sound and faithful concurrent traces. Apart from tools dedicated to data races and atomicity violation detection, we found no available tools targeting general behavioral properties to compare with; tools that can establish causal order and instrument various custom regular actions from the program for monitoring. As such, we implemented the algorithm from [RS04], used in [SRA03, JS08, CSR08]. The implementation uses the same vector clock data structures, tree clocks [MPTV22], as in FACTS.

Experimental setup. We pick for our evaluation real-world Java applications from Renaissance [PRL+19] and DaCapo Benchmarks [BGH+06b], and synthetic programs from [HS12a]. We monitor properties that can be

expressed with total order formalisms to demonstrate our work and show how concurrent traces can help existing monitoring approaches adapt to concurrent programs. We compare three concurrent trace collection approaches: with **FACTS** in both *asynchronous* and *synchronous* modes, and with Algorithm **A** from [RS04] which cannot be run in asynchronous mode.

We also collect linear traces as collected by Java-MOP [CR05a] and show the number of ordering corrections made with concurrent traces. We instrument the programs to collect property-related events and for concurrent traces also capture synchronization actions such as thread operations, synchronized blocks and methods, locks, reads and writes to shared variables, and spawning actors [Agh86] in Akka [Akk22].

Benchmarks and specification.x The benchmarks are chosen to span different concurrency primitives such as thread operations, synchronized blocks, and methods, lock operations, reads and writes, and spawning actors [Agh86] in Akka [Akk22].

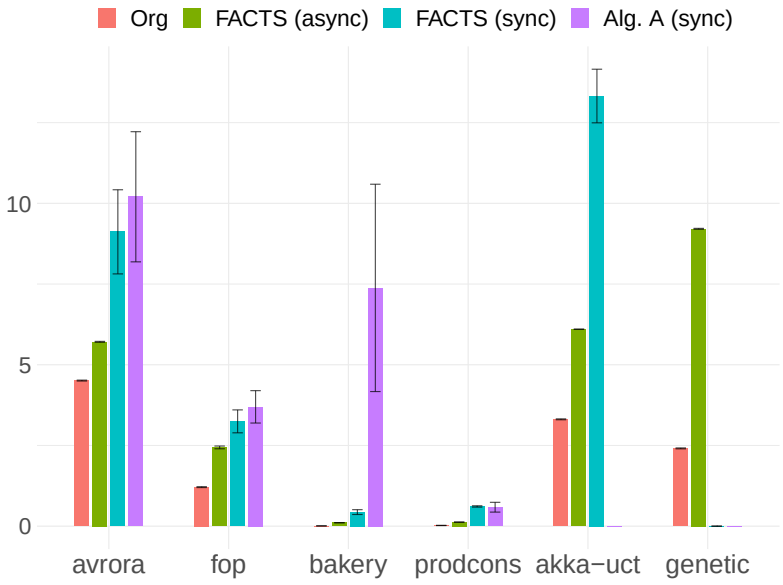
Program **akka-uct** performs load balancing of tasks using an Unbalanced Cobwebbed Tree computation with Akka [Akk22]. Worker tasks with different priorities, *urgent* and *normal*, are completed using actor nodes. We monitor a response property stating that *between the submission and the execution of a task with normal priority (events q and r resp.) if an urgent task is submitted (event p) it should execute (event s) in between q and r*. The property is identical to **P1** from Ex. 35. We have $\mathcal{I}_D = \{\langle s, q\rangle\}$ which is passed to FACTS. We specify instrumentation to extract events from within the execution of the nodes and target node creation and send messages for synchronization actions.

Program **future-genetic** executes a genetic algorithm optimization function using Jenetics [Jen18]. The program has a sequence of parallel tasks executing with contention between them. We check *whether dependent tasks execute in parallel*. The task execution is instrumented and delimited with a *before* task and *after* task events (*bt* and *at* respectively). The property is a mutual exclusion property similar to **P2** from Ex. 35 without q_3 and the *read* events. Here $\mathcal{I}_D$ is empty and passed to FACTS. We specify instrumentation to capture events that execute within the tasks and target synchronized blocks and methods and thread forks for synchronization.

We target type-state properties [SY86a] with the Dacapo benchmarks **avrora** and **fop**. We collect traces to monitor the SafeIterator property that specifies that *no thread should update a collection while another thread iterates over it*, and the HasNext property that *requires calling* `hasNext()` *on an iterator before calling the* `next()` *operation*. With FACTS, events are extracted with runtime information so that they are matched in a parametric monitoring setup [CR09]. Most of the Dacapo programs synchronize using synchronized blocks and methods, so we target those for collecting SAs. For both properties, $\mathcal{I}_D$ is empty and all events are expected to be ordered.

We also run an implementation of the Bakery lock algorithm [Lam74], **bakery (f)**. The algorithm performs synchronization using reads and writes. We introduce a bug such that synchronization between threads is faulty. We check if events in the critical section are not overlapping, i.e., atomic. We also include a classic producer-consumer program, **prods-cons (f)**, which performs synchronization with locks. We also introduce a bug in locking consumers which would spontaneously produce events without acquiring the lock of the shared resource. For both, we are checking a property similar to **P2** from Ex. 35, as such, a similar $\mathcal{I}_D$ is passed to FACTS.

Results and analysis. Fig. 9.9a report the mean execution time for 20 runs of the benchmarked programs. We first note that FACTS was capable of producing sound and faithful traces from all benchmarks as no marginal cases (i.e. conflicting writes) were reported from our vector clock algorithm, even for the *bakery* for instance which relies solely on reads and writes for synchronization. Second, running the algorithm in asynchronous mode, FACTS (async), interferes minimally with the program as it incurs a considerably low overhead in most of the benchmarks. Third, FACTS (sync) performs better than Algorithm A in most of the benchmarks. We fairly believe that our algorithm interferes less with parallelism in the programs as it imposes finer-grained synchronization than Algorithm A. This is highlighted with the bakery algorithm which synchronizes only using shared variables. Algorithm A requires the update of vector clocks associated with a read or write to be atomic through synchronization, while our algorithm does not. This causes the threads to spin more as more contention is added with Algorithm A. Forth, for **future-genetic** and **akka-uct**, these programs use parallel tasks and message-passing (resp.) for managing concurrency. We can see how capturing concurrent traces synchronously from them interferes severely with their behavior. Algorithm A and FACTS (sync) timed out with **future-genetic**, while for **akka-uct**, Algorithm A is not intended to handle message passing. Monitoring programs that use concurrency primitives with higher levels of abstraction need better adaption in the future; for now, we better observe and monitor them asynchronously.



(a) Execution time (s) with trace reordering.

	t-Mon
akka-uct	✗
future-genetic	✓
avrora	✓
fop	✓
bakery	✓
prods-cons	✓
bakery (f)	✗
prods-cons (f)	✗

(b) Trace Monitorability.

t-Mon	Verdict	
✓	T: 100%	F: 0%
✗	T: 32%	F: 68%

(c) Monitoring Soundness.

Figure 9.9: Experimentation Results.

Table 9.6: The table reports for each benchmark: the number of threads (Tr), execution time in seconds of the benchmark (Exec), # of events ($\mathbb{E}$) and their type (Type: S for synchronized, P for parallel), # of synchronization action captured (SA) for FACTS, vector clock algorithm time in sec (VCA), monitorability check **t-Mon**, # of faulty pair orderings (Faulty pairs) from linear traces. K = 10^3, M = 10^6.

	ORG				FACTS				Linear Traces	
	Tr	Exec	$\lvert\mathbb{E}\rvert$	Type	$\lvert$SA$\rvert$	VCA	Exec	t-Mon	Faulty	Exec
akka-uct	64	3.3	1.28M	P+S	3.2M	7.1	6.1	✗	686K	4.91
future-genetic	18	2.4	17M	P+S	1.26M	18.1	9.2	✓	-	9.1
avrora	8	4.5	2.5M	P+S	3.2M	2.9	5.7	✓	-	4.75
fop	1	1.2	1.6M	S	7K	0.6	2.47	✓	-	2.2
bakery	4	0.1	400K	S	9.3M	7.4	4.5	✓	-	0.17
prods-cons	28	0.002	112K	S	242K	0.4	0.13	✓	-	0.06
bakery (f)	4	0.1	400K	S	8M	6.5	4.3	✗	75K	0.16
prods-cons (f)	28	0.002	112K	S	212K	0.4	0.11	✗	21K	0.06

Table 9.9b, reports on the monitorability of the collected traces. The monitor is warned when **t-Mon** is false as it may produce unsound verdicts. We introduce two buggy implementations **prods-cons (f)** and **bakery (f)**, where synchronization between threads is faulty and events execute without acquiring the locks, leading to missing orderings in the executions (resp. the trace). We use both a correct implementation where **t-Mon** = $\top$ and the faulty one where **t-Mon** = $\bot$. We monitor with Java-MOP [CR05a] and collect the verdicts. A *true* verdict (T) means the property is not violated. We report the results of 100 executions in Table 9.9c. We find that monitoring with **t-Mon** = $\top$ yields a verdict *true* (32%) for some executions, while for others, it yields *false* (68%). FACTS is capable of producing warnings in all executions of the faulty programs. In the correct program, verdicts are indeed consistent because the execution of the events themselves is linearized.

Fig. 9.10 reports also in addition to the execution times of the programs the execution times for the vector clock algorithms (labeled VCA). For (async) the algorithm time is excluded from the execution time.

Table 9.6 reports more details about the execution. The detected faulty pairs correspond to unordered events in the execution that are arbitrarily ordered with linear traces. Their order differs in each execution depending on the scheduler of the execution environment. However, they are captured correctly with concurrent traces. The main result we have is that FACTS is capable of reporting the missing orderings and warns the monitor that the execution is not monitorable. For **akka-uct**, urgent and normal tasks are executed concurrently. The concurrent

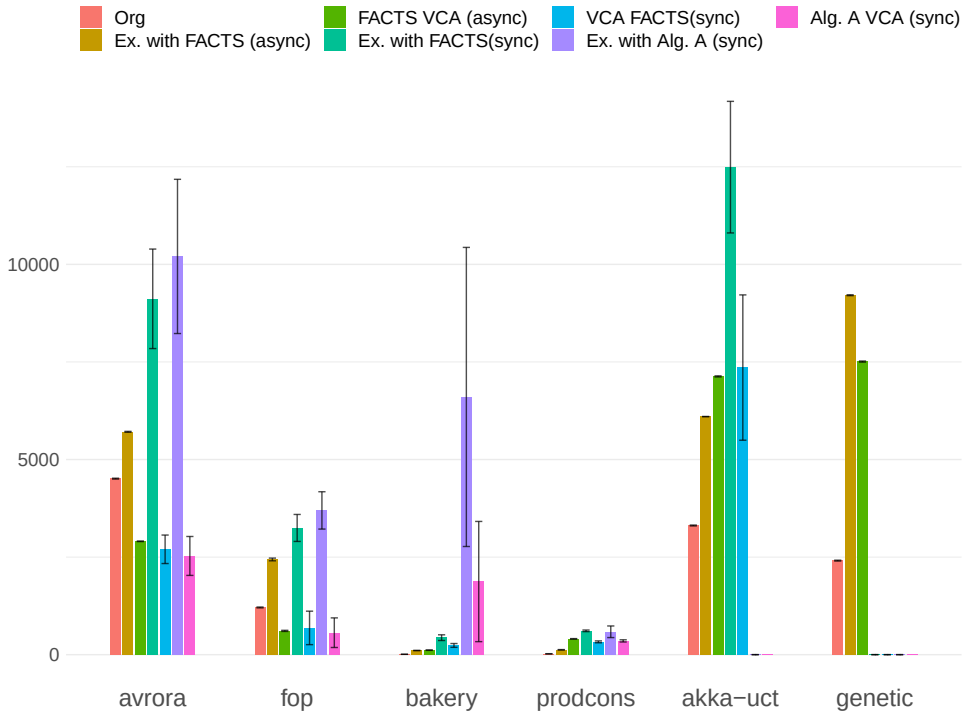


Figure 9.10: Execution time (s) with vector clock algorithm running times.

trace is not enough and monitoring with linear traces produces unsound verdicts. For **bakery** and **prods-cons**, the locking mechanism is faulty. Although the events are collected from supposedly synchronized regions, one should never assume atomicity between events as one might be monitoring to check the atomicity assumption in the first place. For **avrora**, the related events are all synchronized as they are produced by the same threads, and hence here one can use linear traces. For **fop**, running on a single thread, linear traces are ideal and can be safely used. For **future-genetic**, no reported faulty pairs as it seems related events are properly ordered. The execution times with FACTS show that the cost is high when collecting representative traces. Compared to linear traces, the overhead in the execution (Exec) is expected as FACTS is collecting extra synchronization actions. In many cases, they are more than property events. However, the overhead can be reduced by targeting frameworks that provide higher-level abstractions of concurrency and are built on top of low-level primitives, such as actor systems and fork-join frameworks.

9.5.2 Causal Dependence Relation in Specification Patterns

We evaluate how much faithfulness can be relaxed while still being able to soundly monitor an execution. We extract the independence relation $\mathcal{I}_D$ of 55 event-based property specifications from [DAC99, Pat] written as Quantified Regular Expressions (QREs). These specifications span commonly occurring patterns in the verification of programs. They are classified into *occurrence* patterns (first two in green in Fig. 9.11) that target the occurrence of a given event, and *order* patterns (in purple) that target the relative order in which multiple events occur during execution. An example of an *order* pattern is the precedence property: some event S *precedes* some event P, which can be used to specify a requirement that a resource can only be granted in response to a request.

For each specification, we generate an automaton and run Algorithm 3 to find pairs in $\Sigma\times\Sigma$ with causal independence. We do not count pairs with the same symbol as these are in $\mathcal{I}_D$ by definition. For an automaton with the alphabet Σ, the total number of relevent pairs we consider is then $(|\Sigma| \times |\Sigma|) - |\Sigma|$. The patterns contain specifications that vary in $|\Sigma|$ ranging from 1 to 6 letters, we exclude single-letter specifications. For example, for the property s *precedes* p, we find that the pair (s, p) is not in $\mathcal{I}_D$. However, for the property s *precedes* p *before* r, we find that the pair (s, r) in $\mathcal{I}_D$. When monitoring such a property, the concurrent traces are still monitorable even when S and R are not ordered, hence, only requiring order between (s, p) and (p, r). Fig. 9.11 reports the average percentage of pairs in $\mathcal{I}_D$ grouped by pattern and alphabet size. We notice higher percentages for *order* patterns, which are

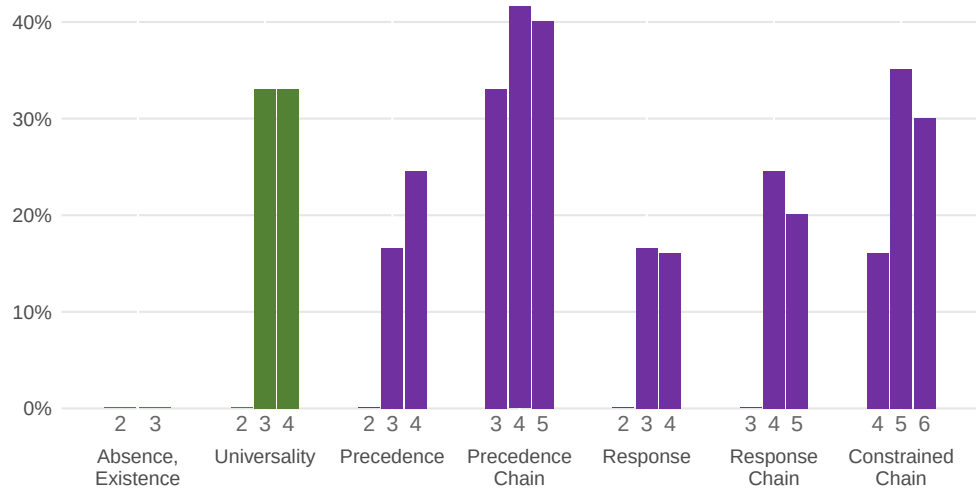


Figure 9.11: Percentage of pairs in $\mathcal{I}_D$ grouped by pattern and alphabet size.

quite common when specifying properties for concurrent programs. For instance, precedence and response patterns often go together and allow users to specify properties over the order of events performed by different threads. The results show that most specifications contain pairs with causal independence, and that faithfulness can be often relaxed when monitoring concurrent programs.

9.6 Evaluating the Opportunistic Monitoring

We here evaluate the opportunistic monitoring approach we presented in Chapter 8. We first opportunistically monitor *readers-writers*, using the specification found in Example 42. We then demonstrate our approach with classical concurrent programs. Full details about these experiments are available at [SF23a].

9.6.1 Readers-Writers

Experiment setup. We instrument the *readers-writers* program to include our monitoring approach and compare it with a global monitoring approach using a specially designed aspect in AspectJ. We consider three distinct scenarios: non-monitored, global, and opportunistic monitoring.

In the non-monitored scenario, there is no monitoring activity. For the second and third scenarios, global and opportunistic monitoring are employed, respectively. It is important to note that global monitoring necessitates additional locks at the monitor level for all concurrently occurring events. We ensure the program's correct synchronization and absence of data races using RVPredict [HMR14b].

Measures. We identify two key parameters: the *number of readers* (nreaders), and the *width of the concurrency region* (cwidth). nreaders is used to specify the maximum number of parallel threads checking local properties within a particular concurrency region. cwidth, meanwhile, quantifies the number of concurrent read operations each reader executes when acquiring the lock, measured in terms of the number of `read` events.

An increase in the size of the concurrency regions results in increased lock contention, especially when multiple concurrent events lead to the global monitor engaging a lock. For our experiments, we set the number of writers to match the values of nreaders, which vary across a range including {1, 3, 7, 15, 23, 31, 63, 127}. Similarly, cwidth takes values from the set {1, 5, 10, 15, 30, 60, 100,150}. The experiment involves 100,000 write operations and 400,000 read operations, with the reads evenly distributed among the readers. We measure the execution time (in milliseconds) across 50 iterations of the program for each combination of parameters and scenarios.

Preliminary results. We present the results using averages and illustrate them with scatter plots and linear regression curves in Figures 9.12 and 9.13. Figure 9.12 demonstrates the overhead change as the number of readers (nreaders) varies. In the non-monitored base program, the execution time increases due to the rising lock contention

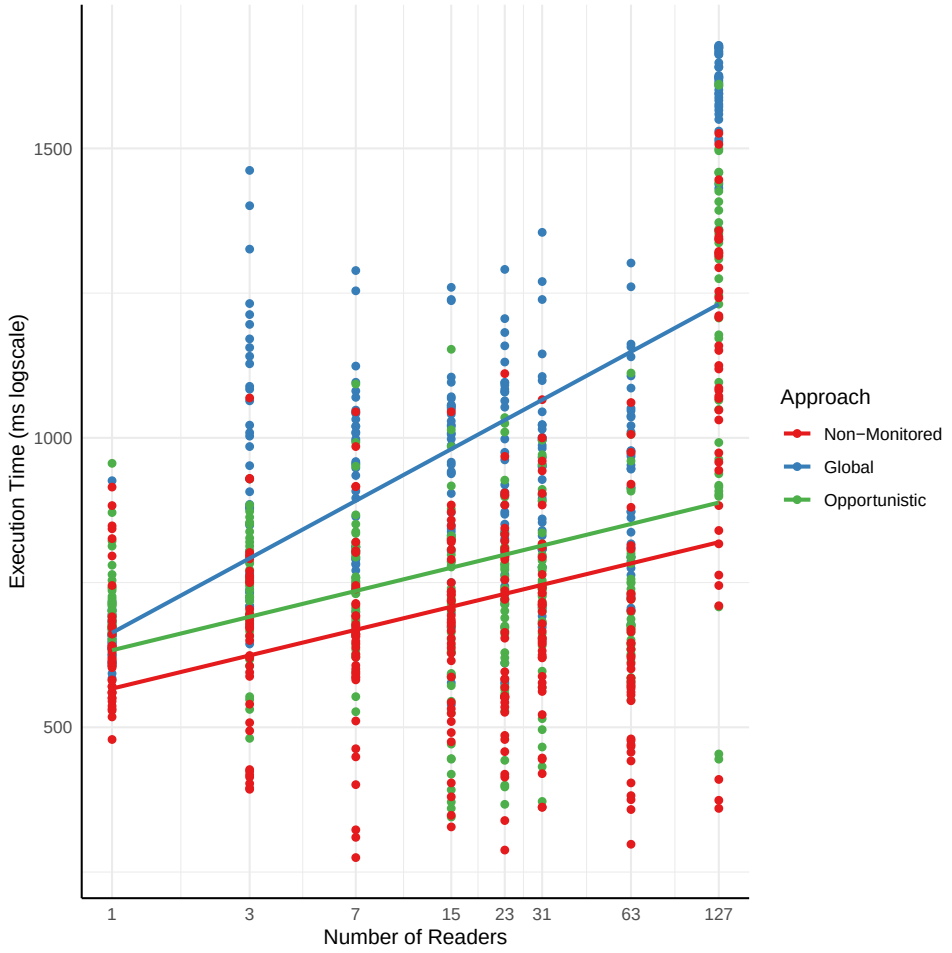


Figure 9.12: Execution time for *readers-writers* when varying the number of readers.

and the Java Virtual Machine (JVM) managing more threads. In global monitoring, the overhead predictably increases with the growing number of threads. As the number of readers increases, the program faces blockages on reads, which ideally should occur concurrently. In opportunistic monitoring, we observe a stable runtime compared to the original program, due to the lack of additional locks and only the delay from evaluating local and scope properties affecting it. Figure 9.13 examines the overhead with changes in the concurrency region's width (cwidth). For the base program, we note a decrease in execution time as more reads happen concurrently without contention on the shared resource lock. In global monitoring, a slight decrease occurs, while opportunistic monitoring shows a more substantial reduction. The increase in concurrent events within a concurrency region brings to light the overhead caused by the global monitor's locking. The global monitor needs to lock to linearize the trace, impacting concurrency. Comparing the patterns of global and opportunistic monitoring reveals that opportunistic monitoring closely follows the speed improvement seen in the non-monitored program, while global monitoring is significantly slower. We anticipated opportunistic monitoring to show a positive performance impact when concurrency regions have a high density of events.

9.6.2 Other Benchmarks

We target classical benchmarks that use different concurrency primitives to synchronize threads. We perform global and opportunistic monitoring and report our results using the averages of 100 runs in Figure 9.14. We use an implementation of the Bakery lock algorithm [Lam74], for two threads *2-bakery* and *n* threads *n-bakery*. The algorithm performs synchronization using reads and writes on shared variables and guarantees mutual exclusion on the critical section. As such, we monitor the program for the *bounded waiting* property which specifies that a process should not wait for more than a limited number of turns before entering the critical section. For opportunistic

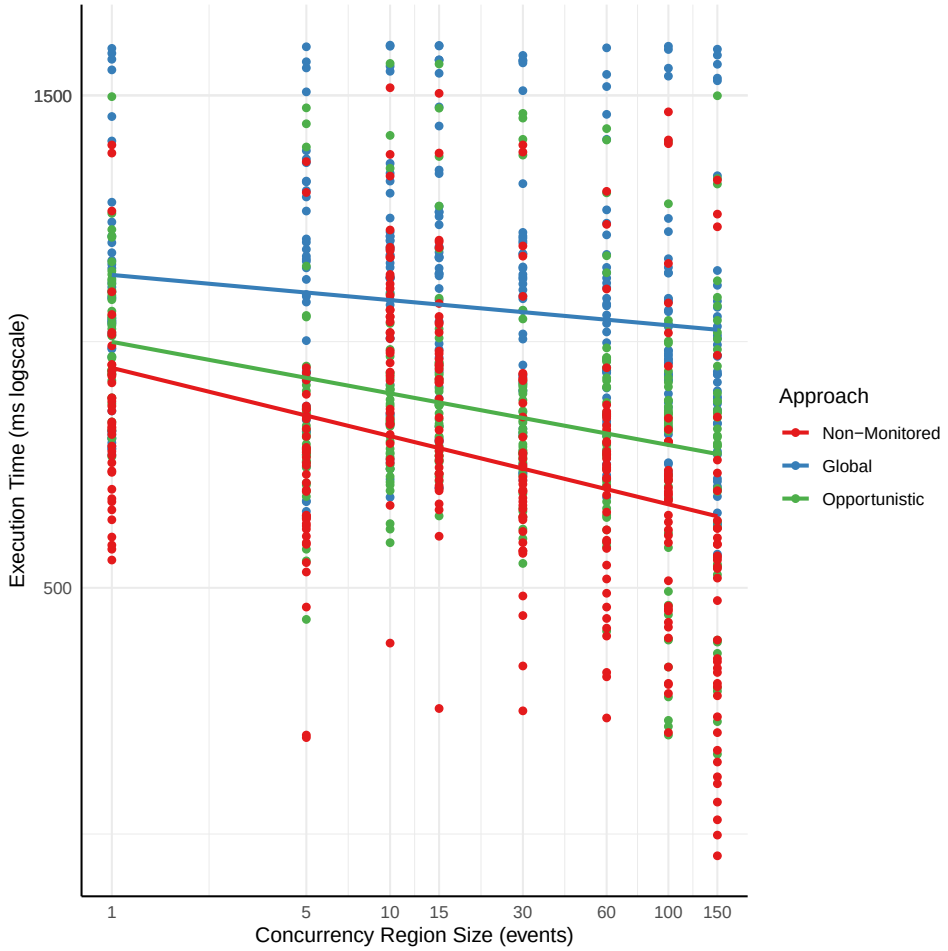


Figure 9.13: Execution time varying the number of events in the concurrency region.

monitoring, thread-local monitors are deployed on each thread to monitor if the thread acquires the critical section. Scope monitors check if a thread is waiting for more than n turns before entering the critical section. We notice slightly less overhead with opportunistic than global for *2-bakery* and more overhead with opportunistic on *n-bakery*. This is because of the small concurrency region (cwidth) which is equal to 1. As such, the overhead of evaluating local and scope monitors by several threads, having a cwidth of 1, exceeds the gain in performance achieved by our approach and hence not fitting for opportunistic monitoring.

We also monitor a textbook example of Ping-Pong algorithm [GR92] that is used for instance in databases and routing protocols. The algorithm synchronizes, using reads and writes on shared variables and busy waiting, between two threads producing events pi for the pinging thread and po for the pong thread. We monitor for the *alternation* property specified as $\varphi \stackrel{\text{def}}{=} (\text{ping} \implies \mathbf{X}\text{pong}) \wedge (\text{pong} \implies \mathbf{X}\text{ping})$. We also include a classic producer-consumer program from [HS12b] which uses a concurrent FIFO queue using locks and conditions. We monitor the *precedence* property, which specifies the requirement that a consume (event c) is preceded by a produce (event p), expressed in LTL as $\neg c \, \mathbf{W} \, p$. For both above benchmarks, we observe less overhead when monitoring with opportunistic, since no additional locks are being enforced on the execution.

We also monitor a parallel mergesort algorithm which is a divide-and-conquer algorithm to sort an array. The algorithm uses the fork-join framework [Lea00] which recursively splits the array into sorting tasks that are handled by different threads. We are interested in monitoring if a forked task is returning a correctly sorted array before performing a merge. The monitoring step is expensive and linear in the size of the array as it involves scanning it. For opportunistic, we use the joining of two subtasks as our synchronizing action and deploy scope monitors at all levels of the recursive hierarchy. We observe less overhead when monitoring with opportunistic than global monitoring, as concurrent threads do not have to wait at each monitoring step. This benchmark motivates us to

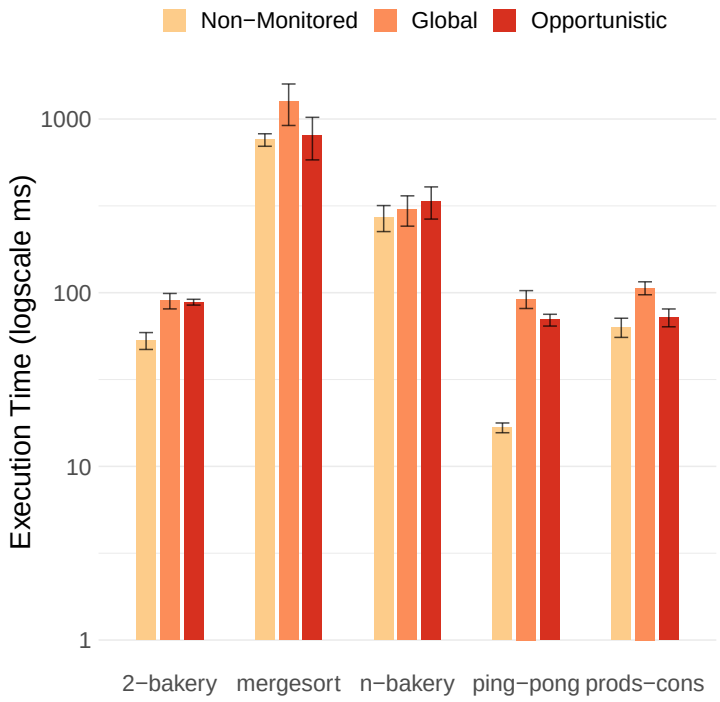


Figure 9.14: Execution time of benchmarks.

further investigate other hierarchical models of computation where opportunistic RV can be used such as [DG08].

9.7 Conclusion

The results presented in this chapter validate our proposed methods and tools throughout this thesis. The evaluation shows the fitness of BISM to be used as a generic tool for runtime verification of JVM-based languages. BISM is capable of handling more use cases than other state-of-the-art tools, namely DiSL and AspectJ, and it is more efficient in terms of performance and memory consumption. The additional DSL that is developed for BISM also shows its effectiveness in terms of performance and ease of use.

The residual runtime verification evaluation shows its effectiveness in reducing the number of instrumentation points and the corresponding impact on runtime performance. Such an analysis can be implemented easily within BISM transformers using the same API that is used for instrumentation.

Moreover, for concurrent programs, the evaluation shows the ability of our proposed vector clock algorithm implemented within FACTS to collect concurrent traces on the fly while interfering minimally with the program's execution compared to other trace collection approaches. We also showed how to assess a collected trace and decide whether it is fit for monitoring or not while using automata-based properties. The opportunistic monitoring assessment also shows how to monitor multithreaded programs without introducing additional synchronization or significantly affecting program behavior.

CHAPTER 10

UseCases

Contents

10.1 Introduction **133**
10.2 Law of Demeter Checker **134**
10.3 Code Analysis of Programs **134**
10.3.1 Mc Cabe Complexity. 136
10.3.2 ABC Complexity. 136
10.3.3 Unused Variables. 137
10.4 Obfuscation **137**
10.4.1 Renaming Obfuscator 138
10.4.2 Junk Code Obfuscator 138
10.5 Mutation of Programs **139**
10.5.1 Return Mutator: *Value Mutation* 139
10.5.2 Instruction Mutator: *Operator Mutation* 139
10.5.3 Void Call Mutator: *Statement Mutation* 140
10.6 Runtime Verification and Enforcement **141**
10.6.1 Good Java Practices: HasNext Property 141
10.6.2 Concurrent Executions: Forcing Advice Atomicity 141
10.6.3 Test Inversion Attack Detection and Enforcement 141
10.7 Logging **143**
10.8 Dynamic Profiling **143**
10.8.1 Call Graph 143
10.8.2 Object Allocation 143
10.9 Dynamic Analysis with Complex Event Processing **145**
10.10 Conclusion **145**

10.1 Introduction

In this chapter, we present several use cases to demonstrate the applicability of BISM in different contexts. We aim here to demonstrate the ability of BISM to be used as a framework for bytecode analysis and instrumentation. Note that all the use cases presented in this chapter can be implemented using bytecode manipulation libraries

such as ASM [BLC02] or Soot [VRCG+99]. The chapter is organized as follows. In Section 10.2, we use BISM to detect violations of the Law of Demeter in a program. This property can be checked statically without executing the program. In Section 10.3, we use BISM to compute several important code analysis metrics on a program. In Section 10.4, we use BISM to obfuscate the program using two different techniques. In Section 10.5, we use BISM in the context of mutation testing. We implement 3 different mutation operators and use BISM to apply them to a program. In Section 10.6, we use BISM to enforce certain properties on the execution of a program. In Section 10.7, we use BISM to log the execution of a program. In Section 10.8, we use BISM to profile the execution of a program. In Section 10.9, we motivate the usage of BISM to implement a dynamic analysis tool based on Complex Event Processing. In Section 10.10 we conclude the chapter.

10.2 Law of Demeter Checker

Our first use case checks a static property on a program using BISM. The Law of Demeter [Lie89] (LOD) is a software engineering principle that states that a method in a class should have limited knowledge about other classes. Specifically, it should only call methods of its class, methods of objects passed to it as arguments, objects created locally within the method, and methods of static fields. This principle is aimed at reducing the coupling between classes.

By adhering to the LOD, a class becomes less dependent on other classes, enhancing maintainability and the overall structure of the code. In every method *m* of a class *A*, the calls are restricted to a specific set of classes:

- The class itself, i.e., *A*.
- Classes of the class fields of *A*.
- Classes of the method *m* parameters.
- Classes whose constructors are invoked in the method *m* body.

This restriction is crucial for ensuring that a class does not become overly reliant on the internal workings of other classes, thereby promoting a more modular and robust design. Following the Law of Demeter leads to software that is easier to maintain and extend. It fosters a design where classes are more isolated, reducing the impact of changes in one part of the system on the rest.

Listing 10.1 shows a BISM transformer for checking the Law of Demeter. The transformer uses the method call join points and filters for invocations of methods on objects. The `onMethodEnter` selector extracts the class name of the method owner, the class names of the fields of the class and the class names of the method parameters to add them to a set of allowed types. The `beforeMethodCall` extracts the class name of the method owner and checks if it is in the set of allowed types. If it is not, it adds it to the map of potential violations. The `onMethodExit` iterates over the map of potential violations and checks if the method owner is in the set of allowed types. If it is not, it reports a violation.

Note that implementing such a checker in AspectJ is infeasible as it requires static context information that is not available at weave time to the user. In [WL05], the authors present this use case as well as checking for side effects in getter methods as motivation to implement an extension for AspectJ. They propose a solution that uses statically executable advice which adds weave-time capabilities to AspectJ. With DiSL such a checker can be implemented, however, it would require the user to run the program. We can see that implementing such static checks is straightforward using BISM. The BISM transformer is implemented in a few lines of code and can analyze the program statically without executing it.

10.3 Code Analysis of Programs

We consider the analysis of program codes along with quality metrics on class files. Software quality is a classic concern in software engineering. Measuring software quality is instrumental in ensuring several properties such as low technical debt, upgradable software, and secure coding. In [HWY09, KAX+99], white-box (i.e., based on source code) analysis metrics are defined to measure quality, understandability, and maintainability. The higher level of abstraction and the updatability of the source code (access to the documentation, comments, fully structured...) are incentives for defining code analysis techniques on source code. As such, there is a lack of tools to compute quality metrics on the bytecode.

```
1
2  public class LawOfDemeterChecker extends Transformer {
3
4
5     HashSet<String> allowedTypes = new HashSet<>();
6     HashMap<String, String> potentialViolations = new HashMap<>();
7
8
9
10    public void onMethodEnter(Method m, MethodDynamicContext dc) {
11
12       allowedTypes = new HashSet<>();
13       potentialViolations = new HashMap<>();
14
15       // Rule 1 : the class itself,
16       // adds the class itself as permissable type
17       allowedTypes.add(m.classContext.classNode.name);
18
19       // Rule 2 : classes of the class fields,
20       // adds all classnames of fields into permissable types
21       m.classContext.classNode.fields
22             .forEach(f ->
                   allowedTypes.add(Type.getType(f.desc).getClassName().replace(".",
                   "/")));
23
24
25       // Rule 3 : classes of the method m parameters
26       // adds all parameter types into permissable types
27       Arrays.stream(Type.getArgumentTypes(m.methodNode.desc))
28             .forEach(t -> allowedTypes.add(t.getClassName().replace(".", "/")));
29
30    }
31
32
33    public void beforeMethodCall(MethodCall mc, MethodCallDynamicContext dc) {
34
35       // Rule 4 : classes whose constructors are invoked in the method m body
36       // if the call is a constructor call add to method permissable types and returns
37       if (mc.methodName.contains("<init>")) {
38          allowedTypes.add(mc.methodOwner);
39          return;
40       }
41
42       // if the method owner is already in method permissable, no need to add it as a
             potential violation
43       if (allowedTypes.contains(mc.methodOwner)) {
44          return;
45       } else {
46          // the method call is not in method permissable, then add it to the potential
                vioilations map
47          potentialViolations.put(mc.methodName, mc.methodOwner);
48       }
49
50
51    }
52
53
54    public void onMethodExit(Method m, MethodDynamicContext dc) {
55
56       System.out.println(allowedTypes);
57
58       // for all potential violations, if they are not in permissable then report a
             violation
59       for (Map.Entry<String, String> pv : potentialViolations.entrySet()) {
60          if (!allowedTypes.contains(pv.getValue())) {
61             System.out.println("LOD violated in " + m.className + "." + m.name + " when
                   calling " + pv.getValue() + "." + pv.getKey());
62          }
63       }
64    }
65 }
```

Listing 10.1: Law of Demeter instrumentation.

BISM permits accessing and computing many valuable properties that can be used to compute standard metrics relying on the CFG of methods, the number of variables and method calls, and the program instructions. This makes such analysis possible on legacy software.

While BISM does not provide access to the source code nor to some classical metrics like Lines Of Code or NPATH complexity, it still provides essential static information. Next, we show how to compute the following software quality metrics: Mc Cabe Cyclomatic complexity, ABC Metric, and the count of unused variables.

10.3.1 Mc Cabe Complexity.

The Mc Cabe Cyclomatic complexity [McC76] is defined as the maximum number of independent paths in a CFG. For a CFG G, it is easily computable by: $V(G) = |Edges_G| - |Nodes_G| + 2$. In Listing 10.2, the transformer uses the computed CFG to count the number of conditional edges inside it.

```
int edgeNumber;

@Override
public void onMethodEnter(...){
  edgeNumber = 0;
}

@Override
public void onBasicBlockExit(BasicBlock bb,...){
  switch (bb.blockType){
    case CONDJUMP:
    case SWITCH:
      edgeNumber++;
  }
}

@Override
public void onMethodExit(Method m,...) {
  int c = m.getNumberOfBasicBlocks() - edgeNumber + 2;
  Log("Cyclomatic complexity of "+m.name+" is : "+ c);
}
```

Listing 10.2: Mc Cabe cyclomatic complexity.

10.3.2 ABC Complexity.

To compute the ABC complexity, we only need to classify instructions and basic blocks. ABC complexity quantifies software complexity by counting Assignments (A), Branching (B), and Conditionals (C) operations. Computing ABC complexity [Fit00] relies on the capability to distinguish between branching, assignments, and conditional jumps. Listing 10.3 shows a transformer that computes the complexity using `blockType` and `opcode` fields of static contexts.

```
public void onMethodEnter(Method m,...) {
  A=B=0;
  C = m.methodNode.tryCatchBlocks.size();
  //To count the try and catch as conditionals
}
public void beforeInstruction(Instruction ins,...) {
  if (isAssignInstr(ins))
    A++;
  //Handle branches
  if (ins.opcode == GOTO || ins.opcode == NEW || ins.isBranchingInstruction() )
    B++;
}
public void beforeMethodCall(...) {
  B++;
}
public void onBasicBlockExit(BasicBlock bb,...) {
  if (bb.blockType == CONDJUMP)
    C++;
}
public void onMethodExit(Method m,...) {
  Log("ABC of "+m.name+" is "+ Math.sqrt(A*A+B*B+C*C));
}
```

Listing 10.3: ABC complexity.

10.3.3 Unused Variables.

We consider that a variable is not used in a method if it is never loaded within the method. For this, the transformer in Listing 10.4 checks whether an instruction is a *direct load* which takes as a parameter a variable index and pushes its value onto the stack. In such a case, the transformer retrieves the index of the variable and sets it as *not unused* in the mapping implemented by boolean array `unusedVars`. The check is run on all instructions, and the variables that have never been loaded on the stack are declared unused.

```
public void onMethodEnter(Method m,...) {
 unusedVars = new boolean[ m.methodNode.localVariables.size()];
 Arrays.fill(unusedVars, true);
}

public void beforeInstruction(Instruction ins,...) {
  if (ins.opcode >= LDC && ins.opcode <= SALOAD){
    //Loading a local variable, therefore variable is used
    if (ins.node instanceof VarInsnNode)
      unusedVars[(ins.node).var] = false;
    else if (ins.node instanceof IincInsnNode)
      unusedVars[(ins.node).var] = false;
  }
}
```

Listing 10.4: Unused variables.

10.4 Obfuscation

Obfuscation [CTL97] is an automated technique used to make the code of a program unintelligible to humans while preserving its functionality. Obfuscation is used to protect intellectual property and to prevent reverse engineering. We implement two obfuscation techniques using BISM: renaming and junk code insertion. These use cases demonstrate BISM's ability to freely manipulate the bytecode of a program without any restrictions. The code uses the abstractions of the ASM library to manipulate the bytecode.

```
public class Renaming extends Transformer {
   //An optional prefix to the renamed objects, use of UTF-8 is valid in bytecode, not
       in source code.
   final String prefix = " ";

   public void onClassEnter(ClassContext c, ClassDynamicContext dc) {
      for (FieldNode f : c.classNode.fields){
         f.name = rename(c.name, f.name);
      }
      for (MethodNode m : c.classNode.methods){
         m.name = rename(c.name, m.name);
      }
   }

   public void onMethodEnter(Method m, MethodDynamicContext dc) {
      for (LocalVariableNode v : m.methodNode.localVariables){
         v.name = rename(m.className, v.name);
         v.desc = Type.BYTE_TYPE.getDescriptor();
      }
   }

   public void beforeMethodCall(MethodCall mc, MethodCallDynamicContext dc) {
      remove(mc.ins);
      insert(new MethodInsnNode(mc.ins.opcode, mc.methodOwner, rename(mc.methodOwner,
          mc.methodName), mc.methodnode.desc));
   }

   public void beforeInstruction(Instruction ins, InstructionDynamicContext dc) {
      if (ins.node instanceof FieldInsnNode){
         remove(ins);
         insert(new FieldInsnNode(ins.opcode, ((FieldInsnNode) ins.node).owner,
               rename(((FieldInsnNode) ins.node).owner,((FieldInsnNode) ins.node).name),
               ((FieldInsnNode) ins.node).desc));
      }
   }
}
```

Listing 10.5: Renaming obfuscation.

10.4.1 Renaming Obfuscator

We present a simple renaming obfuscation that renames all the methods and fields of a program. Listing 10.5 shows the implementation of the renaming obfuscation. The transformer processes names with a fixed function `rename`. We omit its implementation for brevity, however, this function renames the names into ones that contain Unicode characters. This transformation can cause naive decompilers unable to retrieve a workable source code and struggle to parse the Unicode characters. The renaming obfuscation is implemented using the `OnClassEnter` and `OnMethodEnter` selectors. At the class enter join point (Lines 4-12), the transformer renames all the fields and methods of the class. At the method enter join point (Lines 14-20), the transformer renames all the local variables of the method and changes their type to `byte`. Before each method call, the transformer renames the method name (Lines 22-25). Before each field access, the transformer renames the field name (Lines 28-32).

10.4.2 Junk Code Obfuscator

Another obfuscation technique is to insert junk code into the program. Listing 10.6 shows the implementation of the junk code obfuscation. We use the selectors `OnMethodEnter` and`OnMethodExit` to add the junk code (respectively the opaque predicate) after the real code (respectively before). These selectors ensure that the position on each transformed method is only called once by method, which ensures that no duplication happens. Method `insertOpaquePredicate(..)` inserts the opaque predicate at the beginning of the method based on a random number and the bitstrings properties. It is easy to detect as it is static, but it is a proof of concept of what an opaque predicate as defined in [CTL98] could be.

```
1  public class JunkCode extends Transformer {
2     static LabelNode l_junk, l_real, l_end;
3
4     public void onMethodEnter(Method m, MethodDynamicContext dc) {
5        l_real = new LabelNode();
6        l_junk = new LabelNode();
7        l_end = new LabelNode();
8        insertOpaquePredicate(m, l_real, l_junk);
9     }
10
11    public void onMethodExit(Method m, MethodDynamicContext dc) {
12       insert(new JumpInsnNode(Opcodes.GOTO, l_end));
13       //junky code
14       insert(new InsnNode(Opcodes.POP));
15       //To let the user think this code has an importance and needs to be executed
             before real code
16       insert(new JumpInsnNode(Opcodes.GOTO, l_real));
17       insert(l_end);
18    }
19
20    private void insertOpaquePredicate(Method m, LabelNode real_code, LabelNode
          junk_code){
21       /* Calculation */
22       insert(new TypeInsnNode(Opcodes.NEW, "java/util/Random"));
23       insert(new InsnNode(Opcodes.DUP));
24       insert(ASMFactory.invokeSpecial("java/util/Random", "<init>", "()V"));
25       insert(new InsnNode(Opcodes.DUP));
26       insert(ASMFactory.invokeVirtual("java/util/Random", "nextInt", "()I"));
27       insert(ASMFactory.invokeStatic("java/lang/Math", "abs", "(I)I"));
28       insert(new InsnNode(Opcodes.SWAP));
29       insert(ASMFactory.invokeVirtual("java/util/Random", "nextInt", "()I"));
30       insert(ASMFactory.invokeStatic("java/lang/Math", "abs", "(I)I"));
31       insert(new InsnNode(Opcodes.DUP_X1));
32       insert(new InsnNode(Opcodes.IOR));
33
34       /* jumping */
35       insert(new JumpInsnNode(Opcodes.IF_ICMPGT, junk_code));
36
37       /* Real code beginning */
38       insert(real_code);
39
40       m.methodNode.maxStack += 3;
41       m.methodNode.maxLocals += 1;
42    }
43 }
```

Listing 10.6: Junk code obfuscation.

10.5 Mutation of Programs

We consider software testing and, more particularly, mutation testing (see [JH11] for a survey). Mutation testing aims to ensure software quality by checking that slightly modified versions of a program (i.e., mutants) will not pass the same tests as the original. Mutants emulate the programs that would be obtained as the result of programmers' mistakes. There are various types of mutations of various complexity levels [DLS78, OU01]. We consider the following types of often-occurring mutations:

- Value mutations, which change variable values in the program or return values.
- Operator mutations, which change the logical or arithmetical operators used across the program.
- Statement mutations, which change complex constructions, like method calls or even the CFG of the program.

In the following, we define an example mutator for each type of mutation, i.e., a transformer producing such mutations.

10.5.1 Return Mutator: *Value Mutation*

The mutator in Listing 10.7 emulates the fact that a default return value has been forgotten in the program. Hence, the target method always returns the same fixed value instead of the normally computed one. For this, the mutator uses the `onMethodExit` join point and detects whether the parameter method `m` returns a value using the method type. In such a case, the mutator removes the value from the stack. Then, a fixed value (here 0 for integers) is pushed onto the stack to be returned.

```
onMethodExit(Method m,...){
  //Detecting return type
  if (getReturnType(m.methodNode.desc) == VOID_TYPE)
    return;
  //Remove return value from stack
  if (getReturnType(m.methodNode.desc) .getSize() == 1)
    insert(new InsnNode(POP));
  else
    insert(new InsnNode(POP2));

    //Push fixed return value (0)
  switch(getReturnType(m.methodNode.desc) .getSort()){
    case INT:
      insert(new InsnNode(ICONST_0));
      break;
      ...
  }
}
```

Listing 10.7: Return mutator.

10.5.2 Instruction Mutator: *Operator Mutation*

The mutator in Listing 10.8 performs some replacements on a specified set of instructions. The mutator is generic and relies on some abstract methods. A replacement instruction can either be randomly chosen or obtained using a user-defined *mapping* between instructions. To do this for an instruction, the mutators check whether the instruction is in its scope and if so, it replaces it.

```
beforeInstruction(Instruction ins, ...){
 if (isCovered(ins)){
  remove(ins);
  if (negate) insert(negate(ins));
  else insert(random(ins));
 }
}
//Check whether a particular instruction is covered (type, position, ...)
abstract boolean isCovered(Instruction);
//Choose a random operation (compatible in terms of type, arg count ...)
abstract AbstractInsnNode random(Instruction);
//Negate the opcode of a given instruction when applicable
abstract AbstractInsnNode negate(Instruction);
```

Listing 10.8: Generic instruction mutator.

We present two instances of the operator mutator, which are obtained by implementing the abstract methods.

- The mutator in Listing 10.9 targets *conditional operators*, which are detected as *conditional jump* instructions. Another comparison operator replaces conditional operators without changing their destination.
- The mutator in Listing 10.10 targets binary arithmetic operators on integers. Arithmetic operators are replaced either by a random operator or the complementary one (− and +, & and | for bitstring operators...).

```
//If it is a conditional
isCovered(Instruction ins) {
  return ins.isConditionalJump();
}

//Choose a random if which is compatible in terms of type and arg count
random(Instruction insIf) {
  if (insIf.opcode >= Opcodes.IFEQ && insIf.opcode <= Opcodes.IFLE)
    return new JumpInsnNode(randomIFBetween(IFEQ,IFLE), insIf.node.label);
  ...
}

//Negate the opcode of a given if
negate(Instruction ins){
  if (ins.opcode == Opcodes.IFNULL || ins.opcode % 2 == 1)
    return new JumpInsnNode(ins.opcode +1, ins.node.label);
  else
    return new JumpInsnNode(ins.opcode -1, ins.node.label);
}
```

Listing 10.9: Decision mutator.

```
final List<Integer> I2Opcodes = List(
 Opcodes.IADD, Opcodes.ISUB,
 Opcodes.IMUL, Opcodes.IDIV,...);

isCovered(Instruction ins){
  //All double int operand arithmetic instructions
  return I2Opcodes.contains(ins.opcode);
}

random(Instruction ins){
  return new InsnNode(I2Opcodes.get( Math.random()*I2Opcodes.size()));
}

negate(Instruction ins){
  return new InsnNode(ins.opcode + ( I2Opcodes.indexOf(ins.opcode) % 2 == 0 ? 1 : -1));
}
```

Listing 10.10: Arithmetic mutator.

10.5.3 Void Call Mutator: *Statement Mutation*

The mutator in Listing 10.11 removes calls to methods with the void return type. For this, whenever there is a call to such a method, the transformer unloads its parameters from the stack and removes the `INVOKEX` opcode. To check for return types and unload the parameters differently regarding their sizes, the transformer iterates through the method descriptor[1] available through the static context attribute `mc.methodnode.desc`.

```
beforeMethodCall(MethodCall mc,...){
  if (getReturnType(mc.methodnode) != VOID_TYPE)
    return;
  //Pop each argument, respecting its size
  for (var arg: getArgumentTypes(mc.methodnode))
    insert(new InsnNode(arg.getSize() == 1 ? POP : POP2);
  remove(mc.ins);
}
```

Listing 10.11: Void call mutator.

[1]The descriptor is a string representing a type, for a method it permits to access the return and argument types.

10.6 Runtime Verification and Enforcement

In this section, we demonstrate how to use BISM to instrument for runtime verification and enforcement of properties on the execution of a program.

10.6.1 Good Java Practices: HasNext Property

We first consider a simple property that is considered a good Java practice. The **HasNext** property on iterators specifies that the `hasNext()` method should be called and return true before calling the `next()` method on an iterator. We evaluated this property with BISM in Section 9.2.4. Listing 10.12 shows a BISM transformer for instrumenting to monitor the property. The transformer is written with the BISM DSL. It specifies two pointcuts to capture the method calls to `hasNext()` and `next()` on iterator objects. A type-matching pattern is used to match the method calls on iterator objects. Events are then defined to extract two lists from the program, one containing key names we chose and the other containing values. The values extracted are the event name and the iterator object instance. The events are then sent to a monitor implemented in a separate class.

```
pointcut pc1 before MethodCall(* *.*Iterator.next())
pointcut pc2 before MethodCall(* *.*Iterator.hasNext())

event e1(["name", "iterator"], ["n",getMethodReceiver]) on pc1
event e2(["name", "iterator"], ["h",getMethodReceiver]) on pc2

monitor m1{
   class: Monitor
   events: [
      e1 to observe(List, List),
      e2 to observe(List, List)
         ]
}
```

Listing 10.12: **HasNext** instrumentation.

10.6.2 Concurrent Executions: Forcing Advice Atomicity

In Section 7.2.1, we discussed how to force the linearization of events in concurrent programs. Our goal here is to force the atomicity of execution between a program action and its advice. To do so we need to target each join point that corresponds to the program action at two different places: before and after the execution of the advice. Before the execution of the join point, we need to acquire a lock and after the execution of the join point we need to release the lock.

Listing 10.13 shows a transformer that forces the atomicity of the execution of a method call and its advice. We use the `beforeMethodCall` join point to insert the code that acquires the lock before the execution of the method call. We use the `afterMethodCall` join point to insert the code that releases the lock after the execution of the method call. This is equivalent to inserting `synchronized(Monitor.getLock()){ .. }` blocks around the method call.

It is important to note that this instrumentation scenario necessitates the capability for arbitrary code insertion. It requires the ability to insert code at nonadjacent locations in the program. BISM offers a straightforward API to insert arbitrary instructions at any point within the program, using the ASM syntax. Such instrumentation is not possible with AspectJ or DiSL since they do not allow arbitrary bytecode insertion.

10.6.3 Test Inversion Attack Detection and Enforcement

Fault injection attacks aim to modify the behavior of the program by injecting faults in the program. They are often used to perturb smartcards and chip behavior to gain unauthorized access or leak information. One type is the *Test Inversion attack* which aims to break the control-flow integrity of a program. It targets conditional jumps in the control flow of a program. More precisely, in a program, if a comparison ends up with a true verdict and therefore directs the execution flow to the true branch, the test inversion attack will instead direct this flow to the false branch. In Section 9.2.2, we evaluated the detection of test inversion attacks using BISM. We here show how to also enforce the integrity of the control flow in a program using BISM.

```
public class MethodCallsFA extends Transformer {
   int lvUpdate;
   @Override
   public void beforeMethodCall(MethodCall mc, MethodCallDynamicContext dc) {
      lvUpdate = mc.ins.basicBlock.method.methodNode.maxLocals;
      mc.ins.basicBlock.method.methodNode
            .visitMaxs(mc.ins.basicBlock.method.methodNode.maxStack + 1, lvUpdate + 1);

      entermonitor(mc, lvUpdate);
      StaticInvocation inv = new StaticInvocation(..);
      invoke(inv);
   }

   private void entermonitor(MethodCall mc, int lvn) {
      insert(new MethodInsnNode(Opcodes.INVOKESTATIC, "Monitor", "getLock",
            "()Ljava/lang/Object;", false));
      insert(new InsnNode(Opcodes.DUP));
      insert(new VarInsnNode(Opcodes.ASTORE, lvn));
      insert(new InsnNode(Opcodes.MONITORENTER));
   }

   @Override
   public void afterMethodCall(MethodCall mc, MethodCallDynamicContext dc) {
      insert(new VarInsnNode(Opcodes.ALOAD, lvUpdate));
      insert(new InsnNode(Opcodes.MONITOREXIT));
   }
}
```

Listing 10.13: Forcing advice atomicity.

Listing 10.14 shows the BISM transformer to detect and reinforce control flow integrity in case of a test inversion attack. Before each conditional jump in a method, the transformer duplicates the stack values that are used in the conditional jump. In each branch of the conditional, we inject code to reevaluate the condition and jump to this second evaluation target in case we get the wrong verdict.

```
public class EnforceTest extends Transformer {
   LabelNode l_t = new LabelNode();
   LabelNode l_f = new LabelNode();
   String message = "Test Inversion attack is detected. Jumping to correct branch.";
   public void beforeInstruction(Instruction ins, InstructionDynamicContext dc) {
      if (ins.isConditionalJump()) {
         if (ins.stackOperandsCountIfConditionalJump() == 1)
            this.insert(new InsnNode(Opcodes.DUP));
         else
            this.insert(new InsnNode(Opcodes.DUP2));
      }
   }
   @Override
   public void onBasicBlockEnter(BasicBlock bb, InstructionDynamicContext dc) {
      if (bb.blockType == BlockType.CONDJUMP){
         l_t = new LabelNode();
         l_f = new LabelNode();
      }
   }
   @Override
   public void onTrueBranchEnter(BasicBlock jumpingBlock, InstructionDynamicContext dc)
       {
      this.insert(new JumpInsnNode(jumpingBlock.getLastRealInstruction().opcode, l_t));
      print(message);
      this.insert(new JumpInsnNode(Opcodes.GOTO, l_f));
      this.insert(l_t);
   }
   @Override
   public void onFalseBranchEnter(BasicBlock jumpingBlock, InstructionDynamicContext
       dc) {
      this.insert(new JumpInsnNode(jumpingBlock.getLastRealInstruction().opcode, l_t));
      print(message);
      this.insert(new JumpInsnNode(Opcodes.GOTO, l_f));
      this.insert(l_f);
   }
}
```

Listing 10.14: Test Inversion Enforcement instrumentation.

10.7 Logging

Logging is a classic example of a cross-cutting concern that is better implemented following the aspect-oriented paradigm. Indeed, by using Java annotations, one can mark methods that require logging and avoid polluting the source code with multiple logging instructions. This way, one can instrument the program and insert the logging instructions only on annotated methods.

Listing 10.15 shows a transformer that instruments the program to log the execution of selected methods in a program on method entries and exits. One way to mark methods that need to be logged in an application by a developer is by creating a custom annotation and annotating the needed methods. The transformer looks for a hypothetical `@Log` annotation inserted at methods in the base program. The annotation indicates that the method needs to be logged. BISM provides access to annotations on methods through the static context. The `isAnnotated("Log")` returns a boolean flag that indicates if the method is annotated with `@Log`. Then, a simple log message is printed on the console. This example can be extended to extract and log the arguments passed to the method (see Section 10.8.1).

```
1  @Override
2  public void onMethodEnter(Method m, MethodDynamicContext dc){
3     if (m.isAnnotated("Log")) //Checks annotation on method
4        println("Entering method: " + m.name);
5  }
6
7  @Override
8  public void onMethodExit(Method m, MethodDynamicContext dc){
9     if (m.isAnnotated("Log"))
10       println("Exiting method: " + m.name);
11 }
```

Listing 10.15: Logging method execution.

10.8 Dynamic Profiling

We demonstrate how to implement dynamic profiling with BISM. We collect dynamic context from a running program, including the number of method invocations, runtime types of method arguments (Section 10.8.1), number of allocated objects (Section 10.8.2), and return types (Section 10.8.2). We do not focus on implementing the profiler tool but only on how to extract context using BISM.

10.8.1 Call Graph

We consider the dynamic call graph of a program which represents the calling relationship between methods in program execution. For each method call in an execution, we are interested in extracting runtime information from the *caller* and *callee* methods. Listing 10.16 shows the code of a transformer that instruments to extract the caller and callee classes and method names along with their runtime arguments, at each method call. The arguments of the caller and callee are extracted using the dynamic context method `dc.getMethodArgs()`. We instrument two synthetic local arrays in the base program to store the extracted values locally in the method. For the caller, the arguments are retrieved once at method enter to avoid repeating the argument extraction for each invocation by the caller. At `onMethodEnter`, calling `dc.getMethodArgs()` will retrieve the needed values from the local variables of the method. As for the callee, the `dc.getMethodArgs()` will retrieve the arguments directly from the stack. Then, before each method call, an invocation to the profiler method `callGraph` is instrumented, passing the static and dynamic information.

10.8.2 Object Allocation

Object allocation is an important metric in dynamic profiling that allows the user to know the number of created objects in the program and estimate the used memory. Listing 10.17 shows a transformer that instruments to capture allocated objects and arrays in a program. We use the `beforeInstruction` join point and filter for all `NEW` opcodes. To extract the type of the created object, we use the access granted by BISM to the ASM instruction node object and get more details from the bytecode instruction. The extracted static information is then passed to the profiler by invoking its appropriate method.

```
LocalArray callerArgs;
LocalArray calleeArgs;

public void onMethodEnter(Method m, MethodDynamicContext dc){
   //Initialize the local arrays
   callerArgs = dc.createLocalArray(m);
   calleeArgs = dc.createLocalArray(m);

   int args = m.getNumberOfArguments();
   DynamicValue dv;
   for (int i = 1; i < args + 1; i++) {
      dv = dc.getMethodArgs(m, i);
      dc.addToLocalArray(callerArgs, dv);
   }
}
public void beforeMethodCall(MethodCall mc, MethodCallDynamicContext dc){

   dc.clearLocalArray(mc, calleeArgs);

   int args = mc.getNumberOfArgs();
   DynamicValue dv;
   for (int i = 1; i < args + 1; i++) {
      dv = dc.getMethodArgs(mc, i);
      dc.addToLocalArray(calleeArgs, dv);
   }

   //Invoke profiler
   StaticInvocation sti = new StaticInvocation("Profiler", "callGraph");
   sti.addParameter(dc.getThreadName(mc));
   sti.addParameter(mc.method.fullName);
   sti.addParameter(callerArgs);
   sti.addParameter(mc.fullName);
   sti.addParameter(calleeArgs);
   invoke(sti);
}
```

Listing 10.16: Profiling the call graph.

```
@Override
public void beforeInstruction(Instruction ins,...) {
 //Object creation opcodes
   if (ins.opcode == Opcodes.NEW
   || ins.opcode == Opcodes.NEWARRAY
   || ins.opcode == Opcodes.ANEWARRAY
   || ins.opcode == Opcodes.MULTIANEWARRAY) {

      TypeInsnNode instruction = (TypeInsnNode) ins.node;
      //Invoke profiler
      StaticInvocation sti = new StaticInvocation("Profiler", "allocation");
      sti.addParameter(ins.method.fullName);
      sti.addParameter(ins.opcode);
      sti.addParameter(instruction.desc);
      invoke(sti);
   }
}
```

Listing 10.17: Profiling object allocation.

Return Types

Listing 10.18 shows how to extract return types from methods. We use the `afterMethodCall` join point and filter using the static context provided `mc.returns` which returns a boolean flag indicating if the method has a return type in its signature. Then, we extract the return result into the dynamic value object `dv`. After that, an invocation to the profiler is instrumented, which passes the needed information. We choose to box the return value for a more generic implementation.

```
1  @Override
2  public void afterMethodCall(MethodCall mc, MethodCallDynamicContext dc){
3     //If a method returns
4     if (mc.returns) {
5        //Get the result
6        DynamicValue dv = dc.getMethodResult(mc);
7
8        //Invoke profiler
9        StaticInvocation sti = new StaticInvocation("Profiler", "returnTypes");
10       sti.addParameter(caller);
11       sti.addParameter(mc.fullName);
12       sti.addBoxedParameter(dv);
13       invoke(sti);
14    }
15 }
```

Listing 10.18: Profiling return types.

10.9 Dynamic Analysis with Complex Event Processing

The capabilities of BISM can be extended by integrating it with other tools. One possibility is integrating with Complex Event Processing (CEP) engines to perform dynamic analysis of programs. With the expressiveness capabilities of BISM, namely the ability to extract events with various granularity levels and the ability to guide the instrumentation process, we can integrate it with a CEP engine to perform various dynamic analyses on programs. Such an engine would facilitate writing dynamic analyses instead of writing them from scratch and would provide a unified interface for writing and executing them. In the next chapter (Chapter 11), we demonstrate the integration of BISM with the BeepBeep CEP engine to perform dynamic analysis on programs.

10.10 Conclusion

In this chapter, we demonstrated the applicability of BISM in different contexts. The use cases demonstrate that BISM can be used as a generic tool for analyzing programs both statically and dynamically. BISM can be used effectively to perform a variety of static analyses since it facilitates the traversal of the program and access to its static context. It does so by providing the appropriate abstractions. For dynamic analysis, BISM can be effectively used to instrument the program to extract dynamic context and events. The extracted context can be used to perform various dynamic analyses on the program.

CHAPTER 11

Dynamic Program Analysis with BISM and Complex Event Processing

Contents

11.1 Introduction **149**
11.2 Dynamic Program Analysis **150**
11.2.1 Existing Approaches 150
11.2.2 Limitations 151
11.3 BeepBeep Overview **152**
11.4 The BISM-BeepBeep Integration **153**
11.4.1 Implementation 153
11.4.2 Runtime Verification: Monitoring and Synthesis 153
11.4.3 Profiling: The Dynamic Call Graph 155
11.4.4 Log Analysis: Complex Instrumented Events 155
11.4.5 Coverage: Versatile Metrics 157
11.5 Experimental Evaluation **159**
11.5.1 Monitoring 159
11.5.2 Coverage 160
11.5.3 Profiling 160
11.6 Conclusion **161**

Chapter abstract

This chapter bridges BISM and BeepBeep, a complex event processing engine, to provide a flexible and modular approach to dynamic program analysis for JVM-based languages. This combination enhances expressiveness, promotes reusability, allows simultaneous and independent analyses, and integrates seamlessly into JVM-based projects. Various analyses such as monitoring, profiling, coverage measurement, and complex event generation are demonstrated, showcasing the approach's flexibility.

11.1 Introduction

Motivation. Several complementary approaches are available to developers to ensure the quality and reliability of the systems for which they are responsible. *Dynamic program analysis* [GS14] is a term that encompasses a set of techniques, formal and informal, that involve examining a program while it is running or through a postmortem analysis of its execution traces (logs), in order to identify errors, bugs, or unusual behaviors. Developers can leverage dynamic program analysis in multiple ways. For example, *profiling* instruments a program in order to retrieve information about performance or resource consumption [AGH+15] or examine the cause of possible deadlocks in the presence of multiple threads [ABF+10, RB20]. A program can also be instrumented manually using *logging statements* [CJ21]; the resulting logs can be analyzed to reveal anomalies in the operation of a system [JR22], process telemetry or troubleshoot problems [BDDF16]. *Runtime verification* evaluates formal properties related to the correctness of a running program as its execution unfolds [BFFR18b]. Finally, dynamic program analysis can also complement existing *software testing* activities, for example by evaluating coverage metrics on a test suite is being run [BL19].

Despite a myriad of existing approaches and tools for profiling, runtime verification, log analysis, and testing, each typically addresses a specific analysis type, with significant limitations. For instance, runtime verification allows users to check correctness properties, but these are limited to yes/no conditions, demonstrating limited expressiveness. Profilers are primarily focused on identifying memory and performance issues, offering only a handful of built-in analytics with little support for user-defined queries. Log analysis tools are limited as they do not operate online and provide limited query options. Testing tools are capable of measuring coverage and test success, but their capabilities do not extend beyond these functions.

Beyond these natural limitations, these tools often lack the required flexibility for incorporating unique or more tailored analyses. In addition, each of these techniques comes with its particular set of concepts and tools evolving independently of each other. Thus, a developer who wants to profile a program for memory consumption *and* monitor an execution property at runtime will most likely need to employ two different tools, each instrumenting the program in its own way, and requiring the use of different input languages or settings to specify the computation they must respectively execute.

Methodology. Writing custom dynamic analysis tools requires handling two challenging tasks: instrumentation and analysis. Selecting the right instrumentation framework becomes key in this context. For example, as discussed in Chapter 3, bytecode manipulation libraries such as ASM [BLC02] and Soot [VRCG+99] allow for flexible program traversal, extensive low-level coverage and bytecode transformations. Nonetheless, implementing basic instrumentation with these can be quite verbose and demands a certain level of expertise from the user. In contrast, aspect-oriented programming frameworks like AspectJ [Asp] provide a high-level language for specifying instrumentation as well as analysis logic. However, AspectJ lacks bytecode coverage restricting its capability to capture low-level events. Moreover, without the ability to guide the analysis by writing concise static analyzers pre-instrumentation, users can only specify the code for injection into the program, making tasks like extracting a method's control-flow graph unattainable. Given these considerations, we use BISM which effectively bridges the gaps between both approaches and offers a comprehensive solution to the instrumentation needs of dynamic analysis.

However, having a powerful instrumentation platform is not sufficient to obtain a versatile dynamic program analysis tool crossing over the four application domains discussed in Section 11.2.1. Frameworks such as BISM or AspectJ indeed allow users to inject arbitrary pieces of Java code into the execution of an instrumented program; thus, in theory, they are sufficient to allow the implementation of any calculation or analysis of the data extracted from the execution of the said program. However, this offloads on the user the responsibility of programming from scratch which can ultimately become a profiler, a coverage metric or a full-fledged temporal logic monitor. A better (and more realistic) approach is to channel the instrumentation into a mechanism for expressing calculations at a high level of abstraction while maintaining great flexibility and avoiding the pitfall of providing a predefined set of hard-coded recipes.

This is where BeepBeep [Hal18] comes into play. Being a generic event stream processing platform with numerous extensions, it is a good candidate to receive the data elements generated by an instrumented program, and let the user shape arbitrary processing pipelines according to the specific use case at hand, expressed at a suitable level of abstraction. BeepBeep processing units take care of buffering, synchronization between multiple streams, and numerous other lowly tasks that a user would otherwise need to implement directly each time. The use of a

higher-level processing library provides additional benefits: BeepBeep calculations themselves can be abstracted further through the design of domain-specific languages; moreover, BeepBeep has *explainability* features, which allows it to extract elements of a long event stream that explain the output of a calculation [Hal20].

Contributions. We introduce a flexible and modular approach to dynamic program analysis that effectively addresses the limitations of existing tools, in particular their limited expressivity and tight coupling between instrumentation and analysis. Our approach, designed for JVM-based languages including Java, Scala, Kotlin, and Groovy, decouples the instrumentation and analysis process. We utilize BISM to instrument programs and extract events. Simultaneously, we employ a complex event processing engine, BeepBeep [Hal18], to perform a diverse range of analyses on the collected information. BeepBeep ships with off-the-shelf *palettes* for various purposes, such as signal processing, XML manipulation, plotting, and finite-state machines. This combined approach offers numerous improvements over existing tools. It enhances expressiveness in instrumentation by capturing both high-level and low-level events, promotes reusability with straightforward BeepBeep pipeline recycling, and allows both synchronous and asynchronous analyses on the same program execution, enhancing efficiency and isolation. Additionally, it offers seamless integration into any JVM-based project and development pipeline.

Chapter organization. The rest of the chapter is organized as follows. Section 11.2 gives an overview of existing solutions in dynamic program analysis and discusses their limitations. Section 11.3 provides background information on BeepBeep. Section 11.4 presents our approach and demonstrates its flexibility by showcasing different analyses, including monitoring, profiling, measuring branch coverage in unit tests, and generating complex composite events. Section 11.5 experimentally compares the performance of the tool with existing solutions; finally Section 11.6 concludes the paper and discusses future work.

11.2 Dynamic Program Analysis

As opposed to static program analysis, which reasons over the behavior of a system without resorting to executing it, dynamic program analysis encompasses all techniques that collect information about a program while it is running. This broad definition leads to a large variety of tools and approaches, which we describe and discuss in this section.

11.2.1 Existing Approaches

We here review relevant existing approaches to dynamic program analysis. For each of them, we describe the principle of the approach and mention a few tools available to developers; our study focuses on Java and JVM-based languages and solutions. We omit any discussion about runtime verification since it was covered in Chapter 1.

Profiling. A first subset of dynamic analysis solutions concentrate on *profiling*, which consist of gathering statistics on the low-level execution of a program. A typical profiler collects information about the number of live object instances present in the heap, the amount of memory consumed, the state of the call stack, and CPU usage. Profiling is typically aimed at troubleshooting performance issues of a program, such as identifying bottleneck methods, data races or memory leaks.

A simple yet free profiler for Java is VisualVM [vis], which offers these functionalities through a graphical interface where the user can select the data elements to collect, display them as plots or explore them interactively. Other tools, such as Trace Compass [tra] or JProbe [jpra], provide similar functionalities.

Log analysis. Another possible approach is to study the logs produced by a software system; log sources can include execution logs generated from instrumented programs using logging libraries, server logs, system logs, and even network packet captures. Off-the-shelf log analysis tools, such as EventLog Analyzer [eve], GoAccess [goa] and Splunk [spl] offer functionalities to filter, search, aggregate and visualize data extracted from event logs, typically by selecting through a range of predefined calculations.

Some of the aforementioned solutions can be used for the detection of anomalies in the execution of a software system; some tools focus solely on this aspect, such as Logpai [HZHL16], Palisade [KDG+21] and MADneSs [ZCB21]. Since a log can be seen as a form of prerecorded event stream we shall also mention in this category a variety of solutions designed to perform calculations on streams, such as Esper [esp] and Siddhi [SGLN+11] —although we find no record of their use for dynamic program analysis.

Testing and Coverage. Calculating the coverage achieved by a test suite can be seen as a form of dynamic analysis, since it collects information about a program as it executes. In the realm of JVM-based languages, a popular coverage measurement tool is JaCoCo [JaC], which operates as an agent passed to the Java Virtual Machine, gathers coverage information about each line reached by a set of tests, and collates these results in the form of an interactive dashboard. JaCoCo calculates line and branch coverage, and can also aggregate coverage measurements by method, class or package. JCov, which comes built-in with the OpenJDK [jco], works in a similar way; in addition to line and branch coverage, it also calculates block and field coverage.

Query-based testing, first introduced by Holzer *et al.* [HSTV09], is a generalization of these metrics. In this context, a "query" is an expression from a language called FShell Query Language (FQL), which expresses a condition on a sequence of observations made on a program. For example, a query may impose that a specific line be visited, then that a variable be assigned a specific value, etc. To the best of our knowledge, the only runtime tool measuring coverage in this manner is TestCov [BL19], which supports block, branch, and condition coverage, as well as covering calls to an error-function.

11.2.2 Limitations

Writing custom dynamic analysis tools requires handling two challenging tasks: instrumentation and analysis. Instrumentation is the automated process of modifying the program to extract relevant contextual information during a run. The analysis is the process of analyzing the collected information to answer a desired query. Despite the large array of solutions that touch on dynamic program analysis in some way or another, they all present limitations on either of these facets.

Expressiveness

The first is the relative expressiveness and possibility for customization in each approach. Profilers focus on troubleshooting performance issues; they provide a handful of built-in analytics with little to no customization or support for user-defined calculations. Log analyzers sometimes lack the ability to provide online (i.e. real-time) feedback and also restrict users to a set of predefined analytics. Runtime verification tools allow users to specify their own properties, but this must be done in a relatively simple formal language; moreover, these properties are restricted to a pass/fail verdict. Finally, testing tools can measure coverage and test success, but do not provide insights on real-time system behavior.

To illustrate the issue, consider a data processing application that reads in large datasets, performs a series of computations on the data, and outputs the results to a new file. Suppose the program occasionally crashes during processing, possibly due to a file operation error, but that the cause of the crash or the steps leading to its reproduction are not well understood. One may need to analyze the program from several viewpoints:

- *Runtime verification* checks for errors at runtime: Has the program tried to read or write to a file that is closed? Does it open a file for writing while it is already open for reading?
- *Profiling* provides information about the program's resources: How many files are simultaneously open for read access?
- *Log analysis* can help answer questions such as: Has a component written an error message to the log? Are there log entries that correlate with the occurrence of the error?
- *Coverage* provides information about the program's execution in relation to its source code: What branches of the code are executed when the program crashes?

One can observe that each of the tools above can contribute to part of the solution, but that most of the questions could not be answered using a tool from a different category.

Tight Coupling

Many existing tools tightly couple instrumentation and analysis, leading to several limitations. The first of these is restricted flexibility in customizing the analysis. One example is the inability to define new metrics like indirect coverage [HC16] within JaCoCo. Secondly, the expressiveness of the analysis can be limited. Take the runtime verification tool JavaMOP [CR05b], for example, which relies on AspectJ [Asp] for instrumentation. Although, JavaMOP provide multiple plugins to express properties using different specification languages such as LTL and

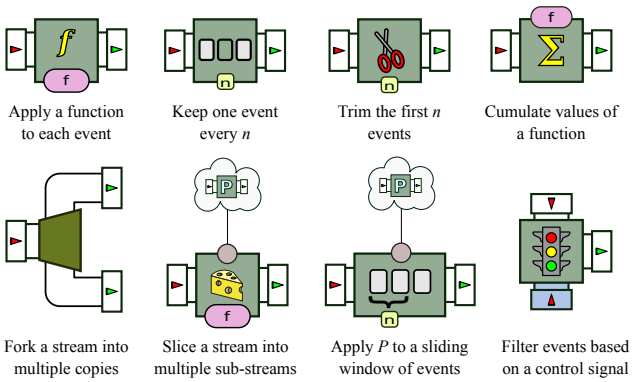


Figure 11.1: BeepBeep's basic processors (adapted from [Hal18]).

extended regular expressions. Its analysis expressiveness is bound by the set of events that AspectJ can extract from the program limiting the ability to write specifications over more complex event patterns (see Section 11.4.4) or over low-level events such as single bytecode instructions as AspectJ does not offer bytecode coverage.

A further limitation is the potential interference caused by integrating the analysis with the program execution. Ideally, one should be able to perform various analyses on the same program execution. This, however, is not achievable when the analysis is closely coupled with the instrumentation. For instance, if one wishes to visualize the dynamic call graph of the program execution and collect coverage information, the program would need to be instrumented twice; whereas it is possible to use the same events for both tasks. This approach is not only inefficient but may also yield different results due to the instrumentation's interference with program execution. Lastly, there is a lack of reusability. When instrumentation and analysis are tightly coupled, reusing the instrumentation code for different analysis types, or vice versa, becomes challenging.

More flexible analysis frameworks have been presented, such as ShadowVM [MKZ+13] for Java and Android, which offers advanced analysis isolation and coverage features. It allows for the execution of multiple analysis tools simultaneously and asynchronously. The analysis is written in unrestricted Java code. ShadowVM is not very portable as it requires running three different processes: one for the observed program, one for the instrumentation, and one for the analysis. These processes operate on three separate VMs, which must be installed on a host machine and communicate using network sockets.

11.3 BeepBeep Overview

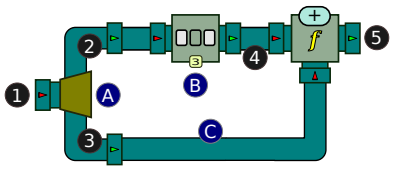
BeepBeep is a generic open source event stream processing library developed in Java [Hal18]. Contrary to most runtime verification frameworks, which offer the user to write specifications using a specific language, BeepBeep's core library provides a handful of generic processor objects performing basic tasks over event streams. These objects can be instantiated directly through Java code, and connected to form networks performing complex calculations. BeepBeep has been used in a variety of case studies over many years [RRKH21, VLGH17b].

Processors

A *processor* is a basic unit of computation that receives one or more event traces as its input, and produces one or more event traces as its output. Processors are represented graphically as a box with "pipes"; the core processors provided by BeepBeep are illustrated in Figure 11.1.

The *ApplyFunction* processor lifts any function f into a processor that applies f on its input events to produce output events. *Fork* is a variant that simply copies its input to multiple outputs, while *TurnInto* transforms each input event into the same constant k. *CountDecimate* is a processor that keeps one event every k. The *Trim* processor removes the first k events of the stream. *Filter* is a processor that discards events based on a stream of Boolean values. The event at position n in the first stream is sent to the output, if and only if the event at the same position in the second stream is the Boolean value *true*.

Some processors can be used to perform aggregations on a set of events. The *Cumulate* processor is designed to



(a) Pipeline

```
1 Fork f = new Fork(2);
2 CountDecimate c = new CountDecimate(3);
3 ApplyFunction a = new ApplyFunction(Numbers.addition);
4 Connector.connect(f, 0, c, 0).connect(f, 1, a, 1)
5   .connect(c, 0, a, 0);
```

(b) Code equivalent

Figure 11.2: Creating pipelines in BeepBeep (taken from [Hal18]).

"accumulate" the successive values of a binary function f. It is often used in conjunction with *Window*, which performs the evaluation of a processor P for each interval of k successive events in the stream. Finally, the *Slice* processor splits an incoming stream into multiple sub-streams, and passes each sub-stream into its own instance of some other processor P.

Pipelines and Palettes

More complex computations can be achieved in two ways. The first is by creating new processors directly as Java objects that are programmed to perform a specific type of processing. Extensions of BeepBeep with predefined custom objects are called *palettes*; there exist palettes for various purposes, such as signal processing, XML manipulation, plotting, and finite-state machines. In addition, BeepBeep also allows existing processors to be *composed*; this means that the output of a processor can be redirected to the input of another, creating complex processor chains. For example, Figure 11.2 shows a simple processor chain, both as a code snippet and as a graphical representation.

11.4 The BISM-BeepBeep Integration

We now describe how BISM and BeepBeep are combined to create a flexible and modular dynamic program analysis framework. We first present the architecture of the integration layer, and then describe how it can be used to implement various analyses. We then present a selection of analyses, that we implemented using our integration layer targeting Java programs. The artifact, containing the source code for the integration layer, the following examples, and experiments, is available at [SFH23c].

The reader will see from these examples that our proposed approach generalizes existing work in two ways. First, no single tool (be it a profiler, a log analyzer or a monitor) would be able to address all of these issues, whereas the BISM-BeepBeep integration can address them all in one uniform framework. Moreover, some of the calculations presented cannot be handled by *any* of the existing tools.

11.4.1 Implementation

The integration between BISM and BeepBeep is enabled via the observation layer described in Section 5.6 where BeepBeep is the analysis engine. Figure 11.3 depicts this integration.

Within each analysis, users specify the instrumentation requirements using the BISM language and the analysis using BeepBeep processors. The integration layer provides facilities such as synchronous and asynchronous observation, event extraction, and event dispatching. Moreover, multiple analyses can be performed on the same program execution.

11.4.2 Runtime Verification: Monitoring and Synthesis

Within our approach, various monitoring scenarios can be implemented. We here present a classical monitoring scenario with events carrying data from the execution.

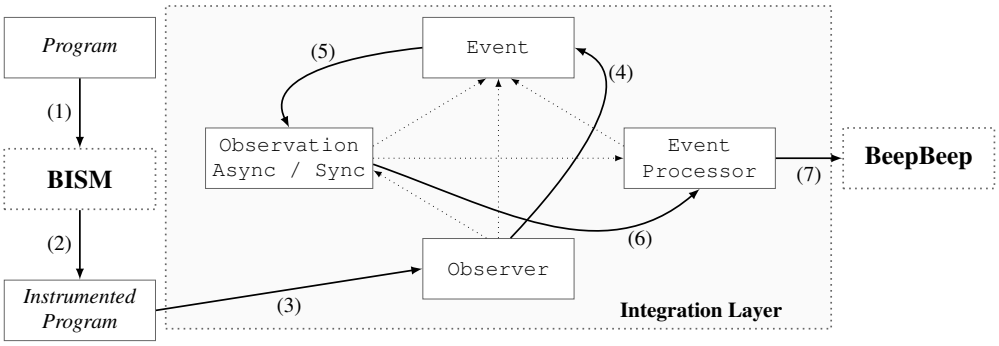


Figure 11.3: Integrating BISM and BeepBeep

```
pointcut pc1 before MethodCall(* *.Iterator.next())
pointcut pc2 before MethodCall(* *.Iterator.hasNext())

event e1(["ev","iterator"],["n",getMethodReceiver]) on pc1
event e2(["ev","iterator"],["h",getMethodReceiver]) on pc2

monitor m1{
 class: bbadapter.Observer
 events: [e1 to observe(List, List), e2 to observe(List, List)]
}
```

Figure 11.4: Instrumentation for **SafeHasNext** property.

Parametric Monitoring

For Java programs, typestate properties [SY86b] are essential for enforcing precise and structured usage of objects; these properties typically constrain the permissible states of certain objects and are monitored by detecting violations of the expected order of method invocations. When monitoring these properties, a *parametric* monitor receives a parameterized trace and spawns multiple monitors for different *trace slices* corresponding to sets of related events. Trace slicing is a natural fit for the `Slice` processor in BeepBeep. This processor takes two parameters: a slicing function and a slice processor (the monitor in our case).

We implemented as a slicing function the algorithm from [CR09]. Figure 11.4 shows the transformer needed to instrument the program with BISM. The integration layer exposes the `Observer.observe` function which accepts 2 lists: one for event keys and one for event values. As such at each relevant join point, such events are extracted. Figure 11.5 shows the pipeline to monitor the **SafeHasNext** property[1]. When an event is received, it is passed to the slicing function. When seeing a new slice, a new monitor instance is automatically created. Else, the event is sent to the appropriate existing monitor. Then after the monitor executes, the results from all monitor instances are accumulated and the conjunction of verdicts is emitted.

[1]The property states that a program does not call the `next` method before calling the `hasNext` method of an iterator.

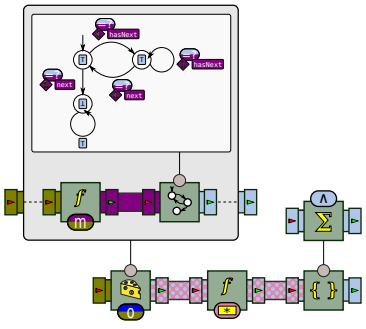


Figure 11.5: Parametric monitoring of the `HasNext` property.

By separating instrumentation from trace analysis, our approach provides enhanced flexibility for runtime verification and monitoring. Firstly, there is a myriad of different approaches to parametric monitoring. They differ in the manner they interpret events with runtime information and project these to instances of monitors. BeepBeep ships with several palettes for constructing monitors using formalisms such as finite-state machines, first-order logic, and temporal logic. This enables users to implement and experiment with various slicing approaches such as [CR09, BFH+12, AAC+05b] using different event granularity levels –something not attainable with other tools. Secondly, within our integration layer, we introduced the asynchronous observation option, which allows the monitor to process events outside the critical path. This design is especially suitable for contexts like real-time systems where monitoring overhead is infeasible [SR94].

Support for Monitor Synthesis

We also added support for the automatic generation of monitors from LTL and regular expressions. For LTL, we used the LamaConv tool [Ins] to translate LTL formulas into equivalent automata and then into a BeepBeep Moore Machine. We also added support for the generation of monitors from regular expressions using the `brics` [bri] library. These regular expressions allow users to specify *bad* or *good* prefixes [KYV01] for safety and co-safety properties. For instance, the language of bad-prefixes for the similar **SafeIterator** property is specified by the user with the following regular expression $c.n^*.u^+.n$. Here, event c denotes the creation of an iterator, event u denotes an update on a collection, and event n denotes using the iterator(`next`). Matching this regular expression with a trace means that the run violates the property.

11.4.3 Profiling: The Dynamic Call Graph

The dynamic *call graph* of a program is a directed graph representing the interprocedural control flow of methods where directed edges indicate a call from one method to another. It is generated at runtime by analyzing the stack trace starting from the program's entry point. Due to the genericity of our approach, we can add two additional properties on the call graph, *invocations count* on edges to count the number of times the caller called the callee, and *time spent* on vertices which reports the total time spent in each method.

The instrumentation is straightforward, we capture the execution of each method with the `onMethodEnter` selector and extract an event carrying the class and method name with its signature. In addition, we inject a timer at the method entry to track the execution time. On method exit, we capture an event with `onMethodExit` carrying with it also the timer value. Figure 11.6a, shows a pipeline for the BeepBeep processor that receives the events and generates the call graph. The processor pushes method enters into a stack to pop them out on exit. The stack processor (top left) receives as input: an event and a boolean flag of *true* or *false* indicating respectively whether the event is a method entry or exit. On *true*, it emits the top element and then pushes the new event into the stack. On *false*, it discards the new event, pops the stack, then emits the top element. The edges are finally aggregated in the rightmost processor which maintains a graph structure that updates the weights of edges and adds up the time spent in each method. Figure 11.6b, shows the dynamic call graph corresponding to a run of the financial transaction system from [BFB+19a].

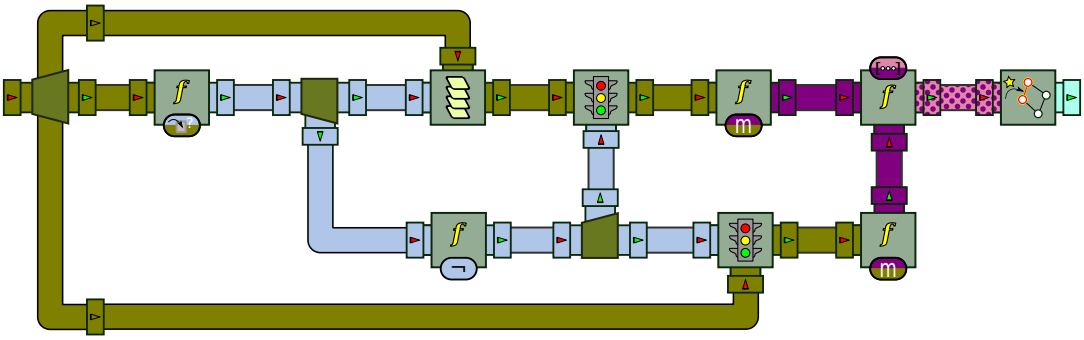
To make the analyses thread-aware, we can include a `Slice` processor and track the execution of each thread separately. These graphs can then be combined into one graph.

The dynamic call graph is a valuable tool for dynamic analysis providing insight into the behavior of a program during execution, and enabling developers to identify and address performance, security, and other issues in the program. Without using a dedicated profiler, extracting such a graph would require some manual work to handle the instrumentation, the graph construction, and extraction. Within BeepBeep, existing abstractions for graph manipulation and exporting to the DOT format allowed us to implement this use case with minimal effort.

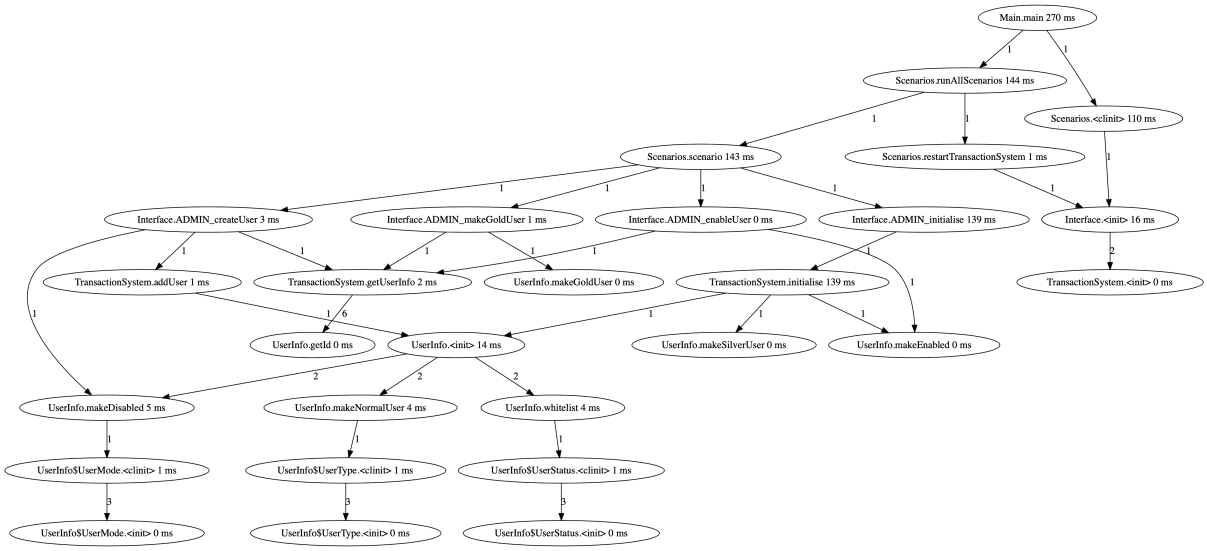
11.4.4 Log Analysis: Complex Instrumented Events

Our next application example leverages the fact that the processing layer of our analysis tool makes use of a full-fledged event stream processing engine, which, in particular, provides functionalities for Complex Event Processing (CEP) [Luc05]. The principle behind CEP is not to merely apply calculations on a stream of events but rather to create so-called "complex" events out of a pattern of multiple lower-level events.

For example, in the context of dynamic program analysis, an `open` operation for a given file handle h, followed by

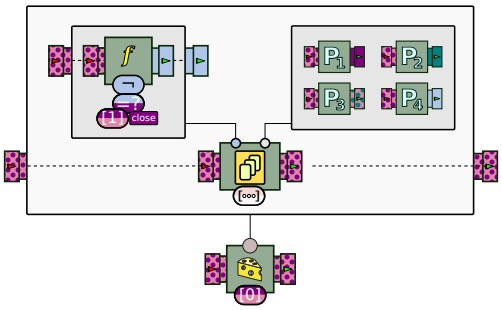


(a) Pipeline

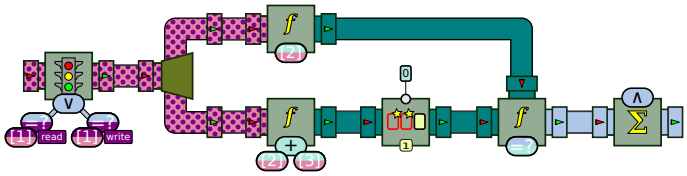


(b) Call graph

Figure 11.6: Generating the call graph of an instrumented program.



(a) High-level pipeline



(b) Processor P_4 checking if accesses are contiguous

Figure 11.7: A complex instrumented event for file operations.

multiple `read` actions and ended by a `close` on that same handle, could be summarized as a single "access to file h" complex event. This event could summarize the access to the file by providing the number of bytes read, the total time during which the file was open, or whether the successive `read` operations were contiguous (i.e. each successive `read` starts at the position where the previous left off). In a sense, complex instrumented events generalize runtime verification; the point here is not to merely detect the presence of a sequential pattern inside an event trace (classical monitors suffice for this task), but rather to produce a higher-level *composite* event that summarizes the occurrence of this pattern.

The BeepBeep engine has an extension named *Complex* containing processors suited for that task. In particular, this extension provides a processor named `RangeCep`, which creates complex events out of a range of simple events. The processor is parameterized by the following elements: 1) A processor π_R signaling the range of contiguous events of the input stream that should be taken into account in the construction of the complex event. 2) An array of processors $\pi_1, \ldots, \pi_n$ that are fed the range of events identified by π_R, and execute an arbitrary calculation. 3) A function f, which ingests the output of each of the π_i and produces a complex event out of them. This construct generalizes the stream quantitative regular expressions of the StreamQRE system [MRA+17] by allowing patterns not expressible as regular expressions.

In the case of the file access complex events described above, Figure 11.7a shows a summary of the high-level pipeline using the `RangeCep` processor. Processor π_R is defined to return $\top$ between the calls to `open` and `close` for a given file handle. The π_i are processors fetching the name of the file, the min/max of the range of bytes read in this interval, the number of bytes read, and whether accesses are contiguous, respectively. The latter processor pipeline is illustrated in Figure 11.7b. Finally, the function f simply aggregates these values into a tuple mapping each attribute to the value calculated by the corresponding π_i.

The presented complex events can be used in conjunction with the previously presented monitoring features, providing richer semantics and allowing for new types of analyses. For example, complex events can be used to detect and analyze patterns of events where the monitor can potentially ignore a lot of noise in the system, focusing only on the significant patterns. Moreover, usages like anomaly detection can be implemented using such events.

11.4.5 Coverage: Versatile Metrics

Coverage analysis is a technique used to measure the effectiveness of a test suite. We here demonstrate two coverage analyses that can help developers effectively test their code and improve their test suites. Both presented analyses are implemented using the same instrumentation specification, which is a testament to the flexibility of decoupling instrumentation from analysis. Notably, both analyses are unattainable within widely used frameworks like JaCoCo where the tight coupling of instrumentation and analysis restricts any customization. Furthermore, such analyses

can be not achieved with AspectJ instrumentation as they require access to low-level events.

Branch Coverage Under Different Execution Paths

Utilizing the control flow information that BISM can extract from a program, we can calculate the branch coverage for each executing method. A high percentage of branch coverage indicates that the test suite is comprehensive and covers a significant portion of the possible code execution paths, thereby reducing the likelihood of undetected bugs or issues. One often overlooked aspect of branch coverage is aggregating the coverage statistics for a method under different calling contexts which may aid developers to isolate problems tied to specific execution paths.

To measure branch coverage, we instrument the program to emit method *access* events on the first access of a method and extract its control flow graph (CFG). The graph is extracted as a list of edges using the control-flow static context objects provided by BISM. Also on each execution of a basic block in a method, we emit a block *entry* event that tracks the transition edge from last executed block and the new one.

Figure 11.8 shows the pipeline that handles these events and performs the analysis. Method access events containing the CFG edges are used to create a reference graph for each method. The `Slice` processor extracts the edges from the event, unpacks them and generates a new graph. Then on each block entry event, the graph associated with the stack trace ID `sid` is updated with the edge incoming with the event. The slice function maintains a stack of the currently executing methods and tags each event with a stack trace ID which is a unique identifier for a specific path in the dynamic call graph. Finally, on each graph we calculate the coverage percentages.

The above analysis allows for the generation of a comprehensive report, enabling users to filter coverage metrics by methods and specific execution paths. Finally, a dashboard is generated that provides users the option to select a

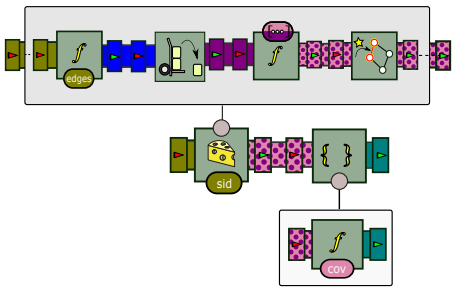


Figure 11.8: Calculating branch coverage pipeline.

path from the stack trace and view a detailed representation of the corresponding method's CFG (at the leaf node). Figure 11.9 shows a partial screenshot of the dashboard for a sample program. The nodes display the basic block ID and line numbers of the source code.

Indirect Coverage

Indirect coverage is a metric introduced by [HC16], which is often overlooked in conventional coverage analysis. A covered entity *e* (e.g. statement, branch, function) is said to be directly covered if there exists a test that directly invokes the method *m* that contains *e*; otherwise *e* is said to be indirectly covered. To the best of our knowledge, no existing tool can compute indirect coverage of a test suite (the authors of [HC16] provide none). Reusing the *same* instrumentation specification from Section 11.4.5, we implemented a pipeline that perform this task; the analysis yields a comprehensive report, outlining both direct and indirect coverage of basic blocks and instructions for every executed method. It further provides the ratio of direct to indirect coverage for each method, enhancing the visibility of test effectiveness. The following output illustrates the calculation of direct (`DC`) and indirect branch (`IC`) coverage for a simple calculator program.

```
IC       =  {Calculator.genericAdd=[1(36-36)]}
DC       =  {Calculator.add=[1(5-5), 3(8-9) ]}
DC Ratio =  {Calculator.add=1.0, Calculator.genericAdd=0.0}
IC Ratio =  {Calculator.add=0.0, Calculator.genericAdd=1.0}
```

This analysis reports for each method under the test suite the directly and indirectly executed basic blocks and instructions. For example, the method `genericAdd` which is called by `add` was indirectly covered by the test

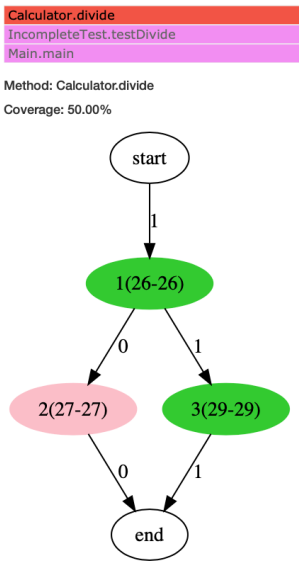


Figure 11.9: Screenshot from the branch coverage dashboard.

suite with a ratio of 1, meaning that none of its entities where directly covered.

11.5 Experimental Evaluation

In the following, we provide experimental measurements of the performance of our proposed tool for some of the analyses presented in the previous section.

11.5.1 Monitoring

We compared the performance of our parametric monitors with well-established monitoring tools such as JavaMOP [CR05b], and MARQ [RCR15a] on a program that creates lists of elements and iterates over them. Varying the number of generated events per execution from 10^3 to 10^6, we report the total execution time and memory used in Table 11.1. As expected, our approach performed slower in comparison to very well-optimized tools such as JavaMOP and MARQ; however, a detailed time overhead and memory overhead shows a linear growth with the size of the trace, as is the case for other tools.

Tool	Execution Time (s)	Used Memory (MB)
BISM-BeepBeep	78.36	9493
JavaMOP	11.31	2045
MARQ	4.159	828
Original	0.874	603

Table 11.1: Execution time and memory usage for each tool.

Figure 11.10 depicts the memory and execution time overhead for multiple executions with varying numbers of events, comparing both an empty analysis and the aforementioned monitoring scenario within BISM-BeepBeep. A notable dip in execution time is due to the JIT compiler. We observe that instrumentation contributes to 10% of the overhead; almost all the running time is spent in BeepBeep's slicing processor. Further inspection revealed that this function doesn't free memory of slices no longer in use, potentially explaining the slowdown, which could be optimized.

In counterpart to this higher overhead, we highlight that our approach brings increased flexibility and modularity. While the existing tools impose a limited specification language (finite-state automata, temporal logic or regular expressions), one can use any slicing function from the literature and any monitor that can be constructed as a BeepBeep pipeline, resulting in a much higher flexibility. For example, instead of checking that the **SafeIterator**

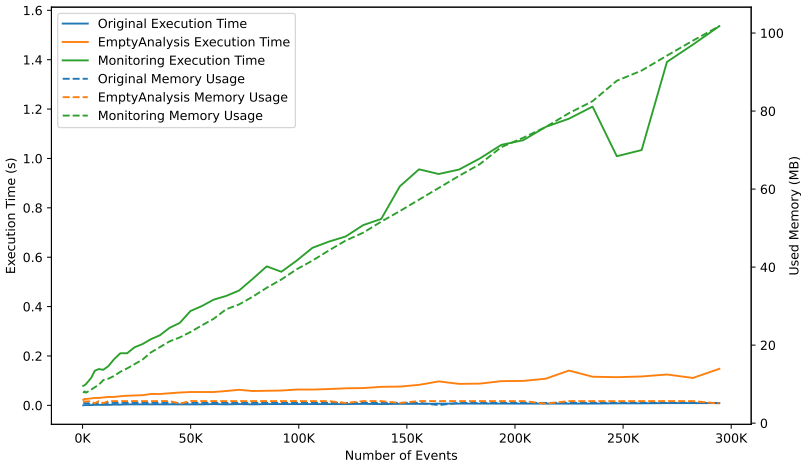


Figure 11.10: Overhead comparison in BISM-BeepBeep: empty analysis vs. monitoring scenario.

property is respected for every iterator instance (a yes/no verdict), one could calculate the fraction of all iterators that violate it in the last n program steps, something that is out of reach of JavaMOP and MARQ.

11.5.2 Coverage

We compared the performance of our branch coverage analysis with JaCoCo [JaC] using a benchmark that simulates a financial transaction system [BFB+19a]. This comparison varied the number of produced events across different runs. As depicted in Table 11.2, our approach's execution time and memory usage were consistently slower than the optimized JaCoCo tool, as anticipated. Yet, it's crucial to put this difference in perspective: the absolute difference is less than 1.5 seconds, which, for a task likely run only a few times daily, may be inconsequential for most users. Beyond this, our method offers significant advantages, providing a much richer dataset. It captures the number of times each branch is taken (transition frequency) and the coverage under unique execution paths, yielding a deeper insight into system behavior. This extensive data means our method analyzes over double the paths than JaCoCo and measures indirect coverage, which JaCoCo does not.

While JaCoCo instruments `Boolean` arrays into the program to track executed lines and subsequently generates a report from these, it does require access to the compiled program for report creation. In contrast, our approach extracts comprehensive events directly and eliminates the need for later access to the source code. Furthermore, our method allows users to narrow down the instrumentation scope, allowing a focused analysis on specific modules of interest, hence leading to a more targeted and efficient coverage assessment.

Metric	Original	JaCoCo	BISM-BeepBeep
Execution Time (ms)	29	655	2080
Memory Usage (MB)	5.03	8.07	15.7
Unique Paths Analysed		71	189
Transition Frequency		✗	✓

Table 11.2: Comparison of JaCoCo and BISM-BeepBeep.

11.5.3 Profiling

During our evaluation, we struggled to adapt existing profilers to solely generate dynamic call graphs. While many of these tools inherently generate these graphs, their extensive suite of features and functionalities makes it exceptionally challenging to isolate and report solely on this particular aspect. As such, we benchmarked BISM-BeepBeep against a manual analysis we wrote with instrumentation with AspectJ, with results in Table 11.3, using the same financial transaction system [BFB+19a]. Our approach utilized BeepBeep's existing abstractions for

Method	Execution Time (ms)	Processing (ms)
Manual Code with AspectJ	2400	110
BISM-BeepBeep	1750	290
Original	800	-

Table 11.3: Benchmarking results for execution time.

graph handling and DOT format export, whereas the manual analysis needed tailored graph code and DOT format expertise. Our method was quicker overall, but slower in isolated processing due to the competitor's single-task focus.

11.6 Conclusion

This chapter introduced a new approach that leverages the capabilities of BISM and BeepBeep, providing a flexible and comprehensive framework for dynamic program analysis. We overcame many limitations of existing methods that couple instrumentation and analysis processes, thereby increasing expressiveness, allowing synchronous and asynchronous analyses, and promoting modularity and reusability. Our approach allows users to express instrumentation requirements with a high level of abstraction allowing the specification of analyses with various granularity levels of events. Moreover, the seamless integration and the minimal setup and configuration requirements facilitate its incorporation into development workflows. Nevertheless, our approach has its limitations. First, there is an overhead due to the abstraction and flexibility it provides, and it may not be as optimized as dedicated tools. Further, compared to other frameworks that implement program observation techniques other than instrumentation, our approach cannot capture internal events that the runtime environment executes such as `dispose` events of objects. Looking ahead, there is substantial scope for future work. One possibility is extending the application of our approach to blockchain languages like Ethereum. Given the increasing prevalence and importance of blockchain technologies, this could open up new avenues for understanding and improving blockchain-based systems.

Part V

Conclusion and Perspectives

CHAPTER 12

Conclusion and Perspectives

In this chapter, we conclude this thesis by summarizing the contributions made in this work and presenting some perspectives for future work.

12.1 Contributions

We here present a summary of the contributions made in this thesis. We start with the instrumentation framework, BISM, which enables the following contributions. We then present the contributions related to combining static and dynamic analysis, the monitoring of concurrent programs and the contributions related to the broader scope of dynamic program analysis.

12.1.1 Program Instrumentation

In response to the need for a comprehensive instrumentation framework, we presented a new dedicated, versatile, and expressive instrumentation framework for runtime verification. This framework is designed to provide three important features we identified as lacking together in existing instrumentation frameworks: the ability to capture events at different levels of granularity, the ability to guide the instrumentation process with weave-time analyses, and the ability to modify the program in an unrestricted manner. This instrumentation framework is implemented in a new state-of-the-art tool for JVM languages, named BISM (Bytecode-Level Instrumentation for Software Monitoring). BISM addresses the mentioned expressiveness requirements while providing a high-level instrumentation language that can be used by various users.

BISM is designed to combine the expressiveness of bytecode manipulation frameworks and the abstraction of aspect-oriented programming (AOP) frameworks. It captures events at the bytecode level, allowing for the monitoring of properties specified over events with various granularity levels. It is designed to support writing weave-time analyses within the instrumentation specification, enabling the guiding of the instrumentation process to consider property and program semantics in order to optimize instrumentation. To achieve this, writing instrumentation is achieved through *transformers*, which are classes that encapsulate join point selection and advice inlining. However, users specify advice using *advice methods* rather than directly writing code snippets. This approach enables users to write weave-time analyses that can be executed at the time of instrumentation along with advice code. Hence both the instrumentation and the analysis are performed at the same place using the same abstractions provided by the framework. It also provides advanced users with the flexibility of unrestricted code modification, which is essential for deploying inline monitors or enforcing certain properties such as a sequential order of concurrent events.

BISM provides constructs, mapping directly to the requirements of runtime verification, to handle three key functions: identification of points of interest, access to contextual information from the program, and the extraction of this information. BISM also performs out-of-the-box analysis on the bytecode to provide additional information about the program methods such as control-flow information and the states of the stack frames. BISM also allows

the composition of these transformers, enabling users to specify instrumentation in a modular and reusable manner. Moreover, transformers in composition are capable of controlling the visibility of program parts, allowing for the integration of various weave-time static analyzers within the instrumentation process.

BISM provides two distinct approaches for implementing transformers: an API-based and an external DSL approach. With the API approach, users define transformers in Java classes, offering them a high degree of control over the instrumentation process. The DSL approach, on the other hand, provides a declarative way of specifying instrumentation directives, offering a subset of the language constructs available in the API approach, but in a more concise and user-friendly manner.

We evaluated the performance of BISM and compared it to AspectJ and DiSL, two state-of-the-art instrumentation frameworks, and showed that BISM can be used as an alternative to these tools for expressive runtime verification. BISM incurs lower overhead in terms of bytecode size, memory usage, and execution time. We also demonstrated that BISM was capable of handling instrumentation requirements that were not possible with AspectJ or DiSL. These are mainly scenarios where unrestricted code modification is needed.

We also demonstrated the applicability of BISM in various static and dynamic analyses. For static analyses, BISM provides the proper abstractions to traverse the program and extract information. Such analyses can be then easily integrated into the instrumentation process.

12.1.2 Guiding the Instrumentation Process with Residual Analysis

Motivated by the new capabilities within the instrumentation framework, we presented a novel approach to residual runtime verification of parametric properties. We instantiated it at the intra-procedural level using novel overapproximation approaches for the program behavior. This technique integrated both the semantics of the property under verification and the program to optimize the instrumentation process and reduce instrumentation points (C2). The presented method applies to both bad and good prefixes, making it suitable for monitoring various kinds of safety and co-safety properties. Unlike existing methods, our approach is independent of any specific static analysis technique. This design decision allows for the modular integration of various static analyses without altering the core residual analysis algorithm, thereby increasing the framework's adaptability to diverse runtime verification scenarios.

The proposed residual analysis is fully implemented and integrated into BISM. We also evaluated the scalability of the approach using a set of benchmarks and showed that it could be used to significantly reduce the runtime overhead of monitoring parametric properties. Overall, this work establishes a foundational framework for residual runtime verification that is agnostic to the choice of static analysis, providing a robust and flexible mechanism for performance optimization in runtime monitoring tasks.

12.1.3 Monitoring of Concurrent Programs

In this thesis, we focused on monitoring general behavioral properties of concurrent programs. These properties target violations that cannot be traced back to classical concurrency errors, and they typically include order violations such as null-pointer dereferences [FPRS12], and typestate violations [JS08, HLR15, SCR12].

We formalized two critical properties that a trace should fulfill to be considered a *representative* of a concurrent program's behavior: *soundness* and *faithfulness*. These qualities established a foundation for subsequent analyses and methodologies. We introduced a generalized approach for collecting traces of concurrent programs aimed at runtime monitoring. This approach was distinct in its use of a real-time vector clock algorithm, designed to establish the causal ordering of events in a non-blocking manner. The non-blocking characteristic was crucial, as it minimized interference with the program's natural execution flow, thus preserving its functional and performance characteristics to a higher degree compared to blocking alternatives.

For behavioral properties expressed in automata-based formalisms, we redefined the notion of monitorability for concurrent traces. We extracted a *causal dependence relation* from a given property to know which events cannot permute in a trace and checked whether a trace contained enough order information (Section 7.6). We then redefined trace *monitorability* for concurrent executions with a necessary condition on the trace to guarantee a sound verdict when monitoring.

To validate our theoretical contributions, we implemented a BISM extension named FACTS. This allowed us to capture concurrent traces from programs and assess their monitorability. We evaluated the performance of FACTS

using a set of benchmarks and real-world applications. The results showed that FACTS was capable of capturing concurrent traces with low overhead, making it suitable for deployment in production environments. We also evaluated the monitorability of the captured traces and showed that the majority of the traces were monitorable. This demonstrated the feasibility of monitoring concurrent programs using automata-based formalisms.

We also presented *opportunistic monitoring*, an approach for the online monitoring of multithreaded programs. We deployed monitoring at two levels. At the first level, thread-local monitors were employed to monitor the execution of individual threads. The second level introduced *scope* monitors which monitored global properties shared across threads. This approach introduces a novel way of instrumenting multithreaded programs, taking advantage of existing synchronization points in a program to monitor it, rather than introducing additional synchronization points, which might interfere with the program's behavior and introduce additional overhead. Scope monitors are evaluated at the end of scope regions which are assumed to be atomically executing. Hence, by assuming the atomicity of scope regions, we ensure that the thread-local monitors can accurately observe and report the state of the thread within the region. Such a hierarchical setup enabled the expression of complex global properties that could not be expressed by classical monitors that rely on linear traces. Moreover, by decentralizing the specification over local and global monitors, various interesting behavioral properties can now be checked which was not possible before. The evaluation of the opportunistic monitoring approach was performed using a set of benchmarks and real-world applications. The results showed that the approach was effective in monitoring global properties without the need for additional synchronization mechanisms that could disrupt program execution. The opportunistic approach incurred a low overhead in terms of memory usage and execution time making it suitable for deployment in production environments.

Overall, the contributions presented throughout this thesis aim to improve the reliability, efficiency, and reusability of existing runtime verification techniques when monitoring concurrent programs.

12.1.4 Contributions to the Broader Scope of Dynamic Program Analysis

By utilizing the instrumentation framework, we presented a modular approach to dynamic program analysis for JVM-based languages. This approach combined the instrumentation framework with a complex event-processing engine to provide a comprehensive framework for dynamic program analysis. BISM provided the ability to capture fine-grained events and perform pre-instrumentation analyses. The complex event processing engine provided the ability to perform synchronous and asynchronous analysis, as well as the ability to combine multiple analyses. The combination of these two components provides a flexible and expressive framework for dynamic program analysis.

Our approach allows users to express instrumentation requirements for dynamic analyses with a high level of abstraction. It also facilitates the specification of analyses with various granularity levels of events. The capability of performing synchronous or asynchronous analysis gives users additional flexibility, making it feasible to adapt to the demands of different analysis scenarios. Moreover, the seamless integration with existing tools and the minimal setup and configuration requirements facilitate its incorporation into development workflows.

12.2 Perspectives

In this section, we present some perspectives for future work.

12.2.1 BISM

We plan on extending the BISM language by adding more features to it. For the core BISM API, we plan to add guards to selectors that will facilitate the filtering of join points to the user. Guards can be annotations that decorate selectors. They allow users to specify a filter on important static information such as scope, method signature, opcode for instruction, and others. Also, the dynamic pointcut *if* can also be considered to allow users to specify guards based on runtime information. This would also be specified as annotations on selectors in order to avoid adding conditional advice methods.

As for the DSL, it currently offers a focused syntax for instrumentation and runtime verification. However, it only supports a subset of the BISM API features and lacks support for inserting arbitrary bytecode instructions. A full integration of the DSL and the API can considered to provide a more expressive language. Moreover, we aim to add enforcement constructs to the DSL (discussed below), allowing users to specify inlined enforcers.

Runtime enforcement. Finally, using the bytecode insertion capabilities of BISM, the language of BISM can be extended with abstractions targeting enforcement capabilities. This will allow users to specify enforcement directives in the same way they specify instrumentation advice. For instance, a user can specify that an instruction such as a method invocation in the base program can only be executed if a certain condition is satisfied. Otherwise, the program is driven to skip the instruction, exit the method, or terminate the execution. Another enforcement directive can be blocking a thread from executing before some instructions until a certain condition is satisfied. Here BISM will be responsible for generating the bytecode for the enforcement specified with the high-level directives and weaving it into the base program. As such effective runtime enforcement [FMRS18, FP19] tools can be implemented.

Adapter for AspectJ. Many of the currently existing runtime verification tools such as JavaMOP [CR05a], Tracematches [AAC+05a, BHL+10], MarQ [RCR15b], and LARVA [CPS09b] rely on AspectJ for instrumentation. As such their analysis expressiveness is bound by the set of events that AspectJ can extract from the program limiting the ability to write specifications over low-level events. Motivating these tools to move to BISM can be considered future work. Developing an adapter that can convert AspectJ aspects to BISM transformers is a first step towards enabling the use of BISM with these tools. This would allow the usage of pre-existing specifications written in AspectJ with BISM. More importantly the use of BISM as a general-purpose instrumentation tool for runtime verification.

Stateful instrumentation. BISM currently provides the ability to capture events at multiple levels of granularity. These events are often observations of some program action. In state-based monitoring, each event might include observations that might span several locations in the program. Consider a property that specifies that a certain variable `x` should be greater than variable `y` at some point in the program. This property requires knowing the values of `x` and `y` at different points in the program where either of the variables is updated since the values of `x` and `y` need to be compared to check if the property is satisfied. As such, the values of `x` and `y` need to be maintained across different locations in the program. This would require an additional layer that maintains shadow variables for the relevant program variables. This layer would be responsible for updating the shadow variables at the appropriate points in the program. BISM can be equipped with such instrumentation directives to support state-based monitoring.

Wider targeting of JVM languages. While various JVM languages all compile down to the same JVM bytecode, modern languages such as Scala, Kotlin, and Groovy provide unique features and constructs. These languages exhibit distinctive features like functional programming constructs, pattern matching, and unique naming conventions, resulting in bytecode that is complex and non-obvious. In order to instrument these programs with a bytecode instrumentation tool, the users need to be fully aware of how these languages compile down to bytecode. This can be challenging for users who are not familiar with the compilation process of these languages. To address this issue, BISM can be equipped with dedicated APIs for these languages. This API can provide language-specific constructs that can be used to specify instrumentation directives. For instance, the API can provide constructs for specifying instrumentation directives for Scala's `match` expressions. This will allow users to specify instrumentation directives more concisely and naturally.

12.2.2 Combinining Static and Dynamic Analysis

Our work on the residual analysis showed promising results in reducing the number of instrumentation points. At its core, our approach depends on guiding the instrumentation with fairly simple static analyzers that avoid data-flow analysis. Nevertheless, to increase precision and restrict over-approximations, it can be extended to include specific static program analysis techniques. The user might opt to include static call graph construction and escape analysis for a more precise approximation of parametric traces, as well as pointer analysis for better approximations of projected traces. For instance, demand-driven pointer analysis [SNQDAB16] can be used to refine the safe list of instructions by making sure that references passed to unrelated method invocations do not point to objects that the monitor is interested in.

Moreover, in cases where pointer analysis does not return a result, the program can be appended with *runtime assertions* to check the aliasing relationship between references. For example, in Ex. 25, if `someflag` (at Line 9) evaluates to `true` and the program enters the branch, the program can be appended with an assertion at Line 10 to check that `l1` and `l2` do not point to the same object. If the assertion fails, the program can be driven to

take a branch where both list updates in Lines 10 and 11 are captured by instrumentation, else the program can be driven to take the branch where these updates are ignored by the instrumentation. As such, some paths of the program need to be duplicated, ones with advice to capture the events and others without advice to ignore them. Given the unrestricted ability to control the program with BISM, such assertions and copying of paths can be easily implemented.

Our current analysis is targeted toward single-threaded programs. Another interesting extension can be to handle concurrent programs and handle thread-escaping references.

12.2.3 Monitoring of Concurrent Programs

We here present perspectives for future work related to the monitoring of concurrent programs.

Collection of Concurrent traces

Collecting concurrent traces for runtime monitoring opens the path to interesting research directions. Firstly, monitoring techniques can be revisited and extended to take into account the partial order. Tools relying on total order in traces can use concurrent traces to check if a trace has the needed causality and, if not, produce warnings. Moreover, specifications (and formalisms) that can match traces obtained from our approach can be elaborated to extend the expressiveness of monitoring to check concurrency-related behavioral properties. For example, our approach is applicable for OpenMP runtimes; it can be used to verify the correctness of the scheduling algorithms of tasks with data dependencies (e.g., [VBGR16]). We can verify that the runtime never schedules two dependent tasks in parallel.

Secondly, it is now possible to define and quantify optimizations for capturing sound and faithful traces. How to obtain optimal faithfulness (Section 7.6.3) with instrumentation is an interesting challenge. Approaches such as sliced causality [CR07] can be inspiring. Sliced causality aims to build a more relaxed causal model that allows exploring interleavings. It does so by applying static analysis with the help of a collected trace to determine data and control dependent actions that should be kept in the causal model. We envision an approach where multiple runs can guide instrumentation to reach the optimal faithfulness ratio. Thirdly, for scenarios where the execution often produces traces that lack the needed causality, the user can be given facilities to enforce the needed causality. Linearization of concurrent events can be achieved for instance on the fly by JVM hot swapping.

Finally, to reduce the overhead of capturing synchronization actions, instrumentation can be tailored to the concurrency framework used in the program. Higher-level concurrency frameworks such as fork-join [Lea00] and software transactional memory [HMPJH05] can be targeted taking advantage of the assumptions they make about the correctness of the program. Tailoring instrumentation to these frameworks reduces the number of collected synchronization actions. It requires assumptions about the correctness of these frameworks. These assumptions can be checked using static and dynamic analysis.

Opportunistic Monitoring

While the preliminary results are promising, additional work needs to be invested to complete the automatic synthesis and instrumentation of monitors. So far, splitting the property over local and scope monitors is achieved manually and scope regions are guaranteed by the user to follow a total order. Analyzing the program to find and suggest scopes suitable for splitting and monitoring a given property is an interesting challenge that we leave for future work. The program can be run, for instance, to capture its causality and recommend suitable synchronization actions for delimiting scope regions. Furthermore, the expressiveness of the specification can be increased by extending scopes to contain other scopes and adding more levels of monitors. This allows for properties that target not just thread-local properties, but also concurrent regions enclosed in other concurrent regions, thus creating a hierarchical setting.

In [BSB17, BSS18], the authors present monitoring for *hyperproperties* written in alternation-free fragments of HyperLTL [CS10]. Hyperproperties are specified over sets of execution traces instead of a single trace. In the opportunistic monitoring setup, each thread produces its trace and thus scope properties we monitor can be expressed in HyperLTL for instance. The time occurrence of events will be delimited by concurrency regions and thus traces will consist of propositions that summarize the concurrency region. We have yet to explore the applicability of specifying and monitoring hyperproperties within our opportunistic approach.

Bibliography

[133] Java Specification Request (JSR) 133. Java Memory Model and Thread Specification Revision, 2004. http://jcp.org/jsr/detail/133.jsp.

[AAC+05a] 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. ACM, 2005.

[AAC+05b] 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. *SIGPLAN Not.*, 40(10):345–364, October 2005.

[ABF+10] Laksono Adhianto, S. Banerjee, Michael W. Fagan, Mark Krentel, Gabriel Marin, John M. Mellor-Crummey, and Nathan R. Tallent. HPCTOOLKIT: tools for performance analysis of optimized parallel programs. *Concurr. Comput. Pract. Exp.*, 22(6):685–701, 2010.

[ABH+06] Wolfgang Ahrendt, Bernhard Beckert, Reiner Hähnle, Philipp Rümmer, and Peter Schmitt. Verifying object-oriented programs with key: A tutorial. volume 4709, pages 70–101, 01 2006.

[AC76] Frances E. Allen and John Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137, 1976.

[ACF+22] Duncan Paul Attard, Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. A runtime monitoring tool for actor-based systems, 2022.

[ACH+05] 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*, pages 87–98, 2005.

[ACP20] Shaun Azzopardi., Christian Colombo., and Gordon Pace. Clarva: Model-based residual verification of java programs. In *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development - MODELSWARD,*, pages 352–359. INSTICC, SciTePress, 2020.

[AG96] S. V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, December 1996.

[AG05] Anurag Agarwal and Vijay K. Garg. Efficient dependency tracking for relevant events in shared-memory systems. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing*, PODC '05, page 19–28, New York, NY, USA, 2005. Association for Computing Machinery.

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[AGH+15] Raja Wasim Ahmad, Abdullah Gani, Siti Hafizah Ab. Hamid, Feng Xia, and Muhammad Shiraz. A review on mobile application energy profiling: Taxonomy, state-of-the-art, and open research issues. *J. of Netw. and Comp. App.*, 58:42 – 59, 2015.

[AGVY11] Edward Aftandilian, Samuel Z. Guyer, Martin T. Vechev, and Eran Yahav. Asynchronous assertions. In Cristina Videira Lopes and Kathleen Fisher, editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011*, pages 275–288. ACM, 2011.

[Akk22] Akka documentation. http://akka.io/docs/, 2022.

[AM07] Tomoyuki Aotani and Hidehiko Masuhara. Scope: an aspectj compiler for supporting user-defined analysis-based pointcuts. In *Proceedings of the 6th International Conference on Aspect-oriented software development*, pages 161–172, 2007.

[ANB+95] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, March 1995.

[Apa] Apache Commons. BCEL (byte code engineering library). https://commons.apache.org/proper/commons-bcel. Accessed: 2020-06-18.

[Asp] Aspectj. https://www.eclipse.org/aspectj/. Accessed: 2023-05-01.

[BCC+09] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Handbook of satisfiability*, 185(99):457–481, 2009.

[BCMD91] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, and David L Dill. Sequential circuit verification using symbolic model checking. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 46–51, 1991.

[BDDF16] Titus Barik, Robert DeLine, Steven Mark Drucker, and Danyel Fisher. The bones of the system: a case study of logging and telemetry at Microsoft. In Laura K. Dillon, Willem Visser, and Laurie A. Williams, editors, *ICSE*, pages 92–101. ACM, 2016.

[Bec04] Kent L. Beck. *JUnit - pocket guide: quick lookup and advice*. O'Reilly, 2004.

[BF72] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Comput.*, 21(6):592–597, June 1972.

[BFB12] Jan Olaf Blech, Ylies Falcone, and Klaus Becker. Towards certified runtime verification. In *International Conference on Formal Engineering Methods*, pages 494–509. Springer, 2012.

[BFB+17] Ezio Bartocci, Yliès Falcone, Borzoo Bonakdarpour, Christian Colombo, Normann Decker, Klaus Havelund, Yogi Joshi, Felix Klaedtke, Reed Milewicz, Giles Reger, Grigore Rosu, Julien Signoles, Daniel Thoma, Eugen Zalinescu, and Yi Zhang. First international competition on runtime verification: rules, benchmarks, tools, and final results of CRV 2014. *International Journal on Software Tools for Technology Transfer*, April 2017.

[BFB+19a] Ezio Bartocci, Yliès Falcone, Borzoo Bonakdarpour, Christian Colombo, Normann Decker, Klaus Havelund, Yogi Joshi, Felix Klaedtke, Reed Milewicz, Giles Reger, Grigore Rosu, Julien Signoles, Daniel Thoma, Eugen Zalinescu, and Yi Zhang. First international competition on runtime verification: rules, benchmarks, tools, and final results of CRV 2014. *Int. J. Softw. Tools Technol. Transf.*, 21(1):31–70, 2019.

[BFB+19b] Ezio Bartocci, Yliès Falcone, Borzoo Bonakdarpour, Christian Colombo, Normann Decker, Klaus Havelund, Yogi Joshi, Felix Klaedtke, Reed Milewicz, Giles Reger, Grigore Rosu, Julien Signoles, Daniel Thoma, Eugen Zalinescu, and Yi Zhang. First international competition on runtime verification: Rules, benchmarks, tools, and final results of crv 2014. *Int. J. Softw. Tools Technol. Transf.*, 21(1):31–70, February 2019.

[BFFR18a] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to runtime verification. In Ezio Bartocci and Yliès Falcone, editors, *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 1–33. Springer, 2018.

[BFFR18b] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to runtime verification. In Ezio Bartocci and Yliès Falcone, editors, *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *LNCS*, pages 1–33. Springer, 2018.

[BFH+12] Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David E. Rydeheard. Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*, pages 68–84. Springer, 2012.

[BGH+06a] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In Peri L. Tarr and William R. Cook, editors, *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, volume 41, pages 169–190, New York, NY, USA, October 2006. ACM.

[BGH+06b] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben 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*, volume 41 of *OOPSLA '06*, pages 169–190. ACM, October 2006.

[BH05] Saddek Bensalem and Klaus Havelund. Dynamic deadlock analysis of multi-threaded programs. In *Proceedings of the First Haifa International Conference on Hardware and Software Verification and Testing*, HVC'05, page 208–223, Berlin, Heidelberg, 2005. Springer-Verlag.

[BH10] Eric Bodden and Klaus Havelund. Aspect-oriented race detection in Java. *IEEE Transactions on Software Engineering (TSE)*, 36(4):509–527, July 2010.

[BHL+10] Eric Bodden, Laurie Hendren, Patrick Lam, Ondřej Lhoták, and Nomair A. Naeem. Collaborative Runtime Verification with Tracematches. *Journal of Logic and Computation*, 20(3):707–723, June 2010.

[BL19] Dirk Beyer and Thomas Lemberger. Testcov: Robust test-suite execution and coverage measurement. In *ASE*, pages 1074–1077. IEEE, 2019.

[BLC02] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *Adaptable and extensible component systems*, 2002.

[BLH10] Eric Bodden, Patrick Lam, and Laurie Hendren. Clara: A framework for partially evaluating finite-state runtime monitors ahead of time. pages 183–197, 01 2010.

[BLH12] Eric Bodden, Patrick Lam, and Laurie J. Hendren. Partially evaluating finite-state runtime monitors ahead of time. *ACM Trans. Program. Lang. Syst.*, 34(2):7:1–7:52, 2012.

[BLS11a] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, September 2011.

[BLS11b] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for ltl and tltl. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, September 2011.

[BMP18] Francesco Adalberto Bianchi, Alessandro Margara, and Mauro Pezzè. A survey of recent trends in testing concurrent software systems. *IEEE Transactions on Software Engineering*, 44(8):747–783, 2018.

[BMTA16] Walter Binder, Philippe Moret, Éric Tanter, and Danilo Ansaloni. Polymorphic bytecode instrumentation. *Softw. Pract. Exp.*, 46(10):1351–1380, 2016.

[bri] BRICS Automaton. http://www.brics.dk/automaton/. Accessed: 2023-05-30.

[BS93] Jonathan Bowen and Victoria Stavridou. Safety-critical systems, formal methods and standards. *Software engineering journal*, 8(4):189–209, 1993.

[BSB17] Noel Brett, Umair Siddique, and Borzoo Bonakdarpour. Rewriting-based runtime verification for alternation-free hyperltl. In *Proceedings, Part II, of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 10206*, page 77–93, Berlin, Heidelberg, 2017. Springer-Verlag.

[BSS18] Borzoo Bonakdarpour, Cesar Sanchez, and Gerardo Schneider. Monitoring hyperproperties by combining static analysis and runtime verification. In *Leveraging Applications of Formal Methods, Verification and Validation. Verification: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part II*, page 8–27, Berlin, Heidelberg, 2018. Springer-Verlag.

[CAPS15] Jesus Chimento, Wolfgang Ahrendt, Gordon Pace, and Gerardo Schneider. Starvoors: A tool for combined static and runtime verification of java. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9333(2012), 09 2015.

[CAS18] Jesús Mauricio Chimento, Wolfgang Ahrendt, and Gerardo Schneider. Testing meets static and runtime verification. In *Proceedings of the 6th Conference on Formal Methods in Software Engineering*, FormaliSE '18, page 30–39, New York, NY, USA, 2018. Association for Computing Machinery.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logics of Programs, Workshop, 1981*, pages 52–71, 1981.

[CF16] Ian Cassar and Adrian Francalanza. On implementing a monitor-oriented programming framework for actor systems. In *Integrated Formal Methods - 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings*, pages 176–192, 2016.

[CFA+17] I Cassar, A Francalanza, L Aceto, A Ingólfsdóttir, et al. A survey of runtime monitoring instrumentation techniques. In *Proceedings Second International Workshop on Pre-and Post-Deployment Verification Techniques, PrePost@ iFM 2017*, pages 15–28, 2017.

[CGS+99] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. *SIGPLAN Not.*, 34(10):1–19, October 1999.

[Chi00] Shigeru Chiba. Load-time structural reflection in java. In Elisa Bertino, editor, *ECOOP 2000 - Object-Oriented Programming, 14th European Conference, Sophia Antipolis and Cannes, France, June 12-16, 2000, Proceedings*, volume 1850 of *Lecture Notes in Computer Science*, pages 313–336. Springer, 2000.

[CHV+18] Edmund M Clarke, Thomas A Henzinger, Helmut Veith, Roderick Bloem, et al. *Handbook of model checking*, volume 10. Springer, 2018.

[CHVB18] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. *Handbook of Model Checking*. Springer Publishing Company, Incorporated, 1st edition, 2018.

[CJ21] Boyuan Chen and Zhen Ming (Jack) Jiang. A survey of software log instrumentation. *ACM Comput. Surv.*, 54(4), May 2021.

[CKFS01] Yvonne Coady, Gregor Kiczales, Michael J. Feeley, and Greg Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In A Min Tjoa and Volker Gruhn, editors, *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering 2001, Vienna, Austria, September 10-14, 2001*, pages 88–98. ACM, 2001.

[CL02] Harold W. Cain and Mikko H. Lipasti. Verifying sequential consistency using vector clocks. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, page 153–154, New York, NY, USA, 2002. Association for Computing Machinery.

[CP17] Christian Colombo and Gordon J. Pace. Runtime verification using LARVA. In Giles Reger and Klaus Havelund, editors, *RV-CuBES*, volume 3 of *Kalpa Publications in Computing*, pages 55–63. EasyChair, 2017.

[CPS09a] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In Darren Cofer and Alessandro Fantechi, editors, *Formal Methods for Industrial Critical Systems*, volume 7609, pages 135–149, Berlin, Heidelberg, October 2009. Springer Berlin Heidelberg.

[CPS09b] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. LARVA — Safer Monitoring of Real-Time Java Programs (Tool Paper). In Dang Van Hung and Padmanabhan Krishnan, editors, *Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, Hanoi, Vietnam, 23-27 November 2009*, pages 33–37. IEEE Computer Society, 2009.

[CR05a] Feng Chen and Grigore Roşu. Java-MOP: A Monitoring Oriented Programming Environment for Java. In *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 546–550. Springer, April 2005.

[CR05b] Feng Chen and Grigore Roşu. Java-MOP: A monitoring oriented programming environment for java. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 546–550, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[CR06] Lori A Clarke and David S Rosenblum. A historical perspective on runtime assertion checking in software development. *ACM SIGSOFT Software Engineering Notes*, 31(3):25–37, 2006.

[CR07] Feng Chen and Grigore Roşu. Parametric and sliced causality. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, page 240–253, Berlin, Heidelberg, 2007. Springer-Verlag.

[CR09] Feng Chen and Grigore Roşu. Parametric trace slicing and monitoring. In Stefan Kowalewski and Anna Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 246–261, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[CS10] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, September 2010.

[CSR08] Feng Chen, Traian Florin Serbanuta, and Grigore Rosu. Jpredictor: A predictive runtime analysis tool for java. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, page 221–230, New York, NY, USA, 2008. Association for Computing Machinery.

[CTL97] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. *https://researchspace.auckland.ac.nz/handle/2292/3491*, 01 1997.

[CTL98] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery, 1998.

[CWG+17] R. Calinescu, D. Weyns, Simos Gerasimou, M. Iftikhar, I. Habli, and T. Kelly. Engineering trustworthy self-adaptive software with dynamic assurance cases. *ACM*, 2017.

[CZS+23] Ming Chai, Xinyi Zhang, Bernd-Holger Schlingloff, Tao Tang, and Hongjie Liu. Online hazard prediction of train operations with parametric hybrid automata based runtime verification. *Reliability Engineering & System Safety*, page 109621, 2023.

[DAC99] 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, page 411–420, New York, NY, USA, 1999. Association for Computing Machinery.

[DFS02] Rémi Douence, Pascal Fradet, and Mario Sûdholt. A framework for the detection and resolution of aspect interactions. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2487:173–188, 2002.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[Dij72] Edsger W Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[DKS13] Mickaël Delahaye, Nikolai Kosmatov, and Julien Signoles. Common specification language for static and dynamic analysis of c programs. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1230–1235, 2013.

[DLK+14] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*, pages 475–488, 2014.

[DLS78] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 1978.

[DP07] Matthew B. Dwyer and Rahul Purandare. Residual dynamic typestate analysis exploiting static analysis. page 124, 2007.

[EF18] Antoine El-Hokayem and Yliès Falcone. In Christian Colombo and Martin Leucker, editors, *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*, volume 11237 of *Lecture Notes in Computer Science*, pages 64–89. Springer, 2018.

[EH18] Antoine El Hokayem. *Runtime Verification of Hierarchical Decentralized Specifications*. Theses, Université Grenoble Alpes, December 2018.

[EHF22] Antoine El-Hokayem and Yliès Falcone. Bringing runtime verification home: a case study on the hierarchical monitoring of smart homes using decentralized specifications. *International Journal on Software Tools for Technology Transfer*, 24(2):159–181, 2022.

[EMN12] Frank Elberzhager, Jürgen Münch, and Vi Tran Ngoc Nha. A systematic mapping study on the combination of static and dynamic quality assurance techniques. *Information and Software Technology*, 54(1):1–15, 2012.

[esp] Esper. https://www.espertech.com/esper, Accessed May 31st, 2023.

[eve] ManageEngine EventLog Analyzer. https://www.manageengine.com/products/eventlog, Accessed May 31st, 2023.

[FCT10] Marco Montali Federico Chesani, Paola Mello and Paolo Torroni. A logic-based, reactive calculus of events. *Fundam. Informaticae*, 105(1-2):135–161, 2010.

[FF04] Cormac Flanagan and Stephen N Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. *SIGPLAN Not.*, 39(1):256–267, January 2004.

[FF09] Cormac Flanagan and Stephen N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 121–133, New York, NY, USA, 2009. Association for Computing Machinery.

[FFM12] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. What can you verify and enforce at runtime? *Int. J. Softw. Tools Technol. Transf.*, 14(3):349–382, 2012.

[FHR13a] Yliès Falcone, Klaus Havelund, and Giles Reger. A tutorial on runtime verification. In Manfred Broy, Doron A. Peled, and Georg Kalus, editors, *Engineering Dependable Software Systems*, volume 34 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 141–175. IOS Press, 2013.

[FHR13b] Yliès Falcone, Klaus Havelund, and Giles Reger. A Tutorial on Runtime Verification. In *Engineering Dependable Software Systems*, pages 141–175. IOS Press, 2013.

[Fit00] Jerry Fitzpatrick. Applying the ABC metric to C, C++, and Java. In *C++ Report*, AOSD '05, pages 245–264, New York, NY, USA, 01 2000. Association for Computing Machinery.

[FKRT18] Yliès Falcone, Srdan Krstic, Giles Reger, and Dmitriy Traytel. A taxonomy for classifying runtime verification tools. In Christian Colombo and Martin Leucker, editors, *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*, volume 23 of *Lecture Notes in Computer Science*, pages 241–262. Springer, 2018.

[FKRT21] Yliès Falcone, Srdan Krstic, Giles Reger, and Dmitriy Traytel. A taxonomy for classifying runtime verification tools. *Int. J. Softw. Tools Technol. Transf.*, 23(2):255–284, 2021.

[FMRS12] Azadeh Farzan, P. Madhusudan, Niloofar Razavi, and Francesco Sorrentino. Predicting null-pointer dereferences in concurrent programs. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, New York, NY, USA, 2012. Association for Computing Machinery.

[FMRS18] Yliès Falcone, Leonardo Mariani, Antoine Rollet, and Saikat Saha. Runtime failure prevention and reaction. In Ezio Bartocci and Yliès Falcone, editors, *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 103–134. Springer, 2018.

[FNRT15] Yliès Falcone, Dejan Nickovic, Giles Reger, and Daniel Thoma. Second international competition on runtime verification CRV 2015. In Ezio Bartocci and Rupak Majumdar, editors, *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*, volume 9333 of *Lecture Notes in Computer Science*, pages 405–422. Springer, 2015.

[FP19] Yliès Falcone and Srinivas Pinisetty. On the runtime enforcement of timed properties. In Bernd Finkbeiner and Leonardo Mariani, editors, *Runtime Verification - 19th International Conference, RV 2019, Porto, Portugal, October 8-11, 2019, Proceedings*, volume 11757 of *Lecture Notes in Computer Science*, pages 48–69. Springer, 2019.

[FPRS12] Azadeh Farzan, Madhusudan Parthasarathy, Niloofar Razavi, and Francesco Sorrentino. Predicting null-pointer dereferences in concurrent programs. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, New York, NY, USA, November 2012. Association for Computing Machinery.

[GA14] Stefan J. Galler and Bernhard K. Aichernig. Survey on test data generation tools - an evaluation of white- and gray-box testing tools for c#, c++, eiffel, and java. *STTT*, 16(6):727–751, 2014.

[GK10] Paul Gastin and Dietrich Kuske. Uniform satisfiability problem for local temporal logics over Mazurkiewicz traces. *Inf. Comput.*, 208(7):797–816, 2010.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[goa] GoAccess. https://goaccess.io, Accessed May 31st, 2023.

[GR92] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.

[GS14] Anjana Gosain and Ganga Sharma. A survey of dynamic program analysis techniques and tools. In Suresh Chandra Satapathy, Bhabendra Narayan Biswal, Siba K. Udgata, and J. K. Mandal, editors, *FICTA*, volume 327 of *Advances in Intelligent Systems and Computing*, pages 113–122. Springer, 2014.

[GS21] Felipe Gorostiaga and César Sánchez. HLola: a very functional tool for extensible stream runtime verification. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *TACAS*, volume 12652 of *LNCS*, pages 349–356. Springer, 2021.

[GZC+11] Qi Gao, Wenbin Zhang, Zhezhe Chen, Mai Zheng, and Feng Qin. 2ndstrike: Toward manifesting hidden concurrency typestate bugs. *SIGPLAN Not.*, 46(3):239–250, March 2011.

[Hal18] Sylvain Hallé. *Event Stream Processing With BeepBeep 3: Log Crunching and Analysis Made Easy*. Presses de l'Université du Québec, 2018.

[Hal20] Sylvain Hallé. Explainable queries over event logs. In *EDOC*, pages 171–180, 2020.

[HBM20] Joseph Herkert, Jason Borenstein, and Keith Miller. The boeing 737 max: Lessons for engineering ethics. *Science and Engineering Ethics*, 26(6):2957–2974, 2020.

[HC16] Chen Huo and James Clause. Interpreting coverage information using direct and indirect coverage. In *ICST*, pages 234–243. IEEE Comp. Soc. 2016.

[Hec77] Matthew S Hecht. Data flow analysis of computer programs, 1977.

[HLR15] Jeff Huang, Qingzhou Luo, and Grigore Rosu. Gpredict: Generic predictive concurrency analysis. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Volume 1*, pages 847–857, 2015.

[HMPJH05] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, page 48–60, New York, NY, USA, 2005. Association for Computing Machinery.

[HMR14a] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. Maximal sound predictive race detection with control flow abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 337–348, New York, NY, USA, 2014. Association for Computing Machinery.

[HMR14b] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. Maximal sound predictive race detection with control flow abstraction. *SIGPLAN Not.*, 49(6):337–348, June 2014.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[HNB06] Wilke Havinga, I Nagy, and Lodewijk M J Bergmans. An Analysis of Aspect Composition Problems. Technical Report Technical Report IAI-TR-2006-6, 2006.

[HNBA07] Wilke Havinga, Istvan Nagy, Lodewijk Bergmans, and Mehmet Aksit. A graph-based approach to modeling and detecting composition conflicts related to introductions. *ACM International Conference Proceeding Series*, 208:85–95, 2007.

[Hoa69] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[HR01] Klaus Havelund and Grigore Rosu. Monitoring programs using rewriting. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 135–143. IEEE, 2001.

[HRTZ18a] Klaus Havelund, Giles Reger, Daniel Thoma, and Eugen Zalinescu. Monitoring events that carry data. In Ezio Bartocci and Yliès Falcone, editors, *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 61–102. Springer, Cham, 2018.

[HRTZ18b] Klaus Havelund, Giles Reger, Daniel Thoma, and Eugen Zălinescu. *Monitoring Events that Carry Data*, pages 61–102. Springer International Publishing, Cham, 2018.

[HS12a] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[HS12b] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[HSF23] Sylvain Hallé, Chukri Soueidi, and Yliès Falcone. Leveraging runtime verification for the monitoring of digital twins. In *Preproceedings of the Workshop on Applications of Formal Methods and Digital Twins*, Lübeck, Germany, 3 2023. Informatics Library, University of Oslo. Co-located with the 25th International Symposium on Formal Methods.

[HSTV09] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. Query-driven program testing. In Neil D. Jones and Markus Müller-Olm, editors, *VMCAI*, volume 5403 of *LNCS*, pages 151–166. Springer, 2009.

[HWY09] T. Honglei, S. Wei, and Z. Yanan. The research on software metrics and software complexity metrics. In *2009 International Forum on Computer Science-Technology and Applications*, 2009.

[HZHL16] Shilin He, Jieming Zhu, Pinjia He, and Michael R. Lyu. Experience report: System log analysis for anomaly detection. In *ISSRE*, pages 207–218. IEEE Comp. Soc. 2016.

[IEE17] IEEE. Ieee standard for system, software, and hardware verification and validation. *IEEE Std 1012-2016 (Revision of IEEE Std 1012-2012/ Incorporates IEEE Std 1012-2016/Cor1-2017)*, pages 1–260, 2017.

[IML09] Juha Itkonen, Mika Mäntylä, and Casper Lassenius. How do testers do it? an exploratory study on manual testing practices. In *Proceedings of the Third International Symposium on Empirical Software Engineering and Measurement, ESEM 2009*, pages 494–497. ACM / IEEE Computer Society, 2009.

[Ins] Institute for Software Engineering and Programming Languages. LamaConv - Logics and Automata Converter Library. www.isp.uni-luebeck.de/lamaconv.

[JaC] JaCoCo Java code coverage library. https://www.jacoco.org/. Accessed: 2023-05-01.

[jco] JCov. https://wiki.openjdk.org/display/CodeTools/jcov, Accessed May 31st, 2023.

[Jen18] Open-source Jenetics repository. https://github.com/jenetics/jenetics, 2018.

[Jet23] JetBrains s.r.o. Kotlin language documentation. https://kotlinlang.org/docs/home.html, 2023. Accessed: 2023-04-30.

[JFMP17] Raphaël Jakse, Yliès Falcone, Jean-François Méhaut, and Kevin Pouget. Interactive runtime verification - when interactive debugging meets runtime verification. In *28th IEEE International Symposium on Software Reliability Engineering, ISSRE 2017, Toulouse, France, October 23-26, 2017*, pages 182–193. IEEE Computer Society, 2017.

[JH11] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 2011.

[jpra] JProbe suite: Complete Java performance tools suite. http://tan.com/jprobe, Accessed May 30th, 2023.

[jprb] JProfiler. https://www.ej-technologies.com/products/jprofiler/overview.html. Accessed: 2023-06-01.

[JR22] Jisha M. Jose and S. R. Reeja. Anomaly detection on system generated logs—a survey study. In Subarna Shakya, Robert Bestak, Ram Palanisamy, and Khaled A. Kamel, editors, *Mobile Comp. and Sustainable Inf.*, pages 779–793, Singapore, 2022. Springer Singapore.

[JS08] P. Joshi and K. Sen. Predictive typestate checking of multithreaded java programs. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, page 288–296, USA, 2008. IEEE Computer Society.

[JSW+16] Yu Jiang, Houbing Song, Rui Wang, Ming Gu, Jiaguang Sun, and Lui Sha. Data-centered runtime verification of wireless medical cyber-physical system. *IEEE transactions on industrial informatics*, 13(4):1900–1909, 2016.

[JTF17] Yogi Joshi, Guy Martin Tchamgoue, and Sebastian Fischmeister. Runtime verification of ltl on lossy traces. *Proceedings of the Symposium on Applied Computing*, 2017.

[JVCS07] Jonathan Jacky, Margus Veanes, Colin Campbell, and Wolfram Schulte. *Model-based software testing and analysis with C*. Cambridge University Press, 2007.

[KABM12] Stephen Kell, Danilo Ansaloni, Walter Binder, and Lukáš Marek. The jvm is not observable enough (and what to do about it). In *Proceedings of the Sixth ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '12, page 33–38, New York, NY, USA, 2012. Association for Computing Machinery.

[KAX+99] T. M. Khoshgoftaar, E. B. Allen, Xiaojing Yuan, W. D. Jones, and J. P. Hudepohl. Assessing uncertain predictions of software quality. In *Proceedings Sixth International Software Metrics Symposium (Cat. No.PR00403)*, 1999.

[KBM14] Ivo Krka, Yuriy Brun, and Nenad Medvidovic. Automatic mining of specifications from invocation traces and method invariants. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 178–189, New York, NY, USA, 2014. Association for Computing Machinery.

[KDG+21] Sean Kauffman, Murray Dunne, Giovani Gracioli, Waleed Khan, Nirmal Benann, and Sebastian Fischmeister. Palisade: A framework for anomaly detection in embedded systems. *J. Syst. Archit.*, 113:101876, 2021.

[KF19] Ali Kassem and Yliès Falcone. Detecting fault injection attacks with runtime verification. In *Proceedings of the 3rd ACM Workshop on Software Protection*, SPRO'19, page 65–76, New York, NY, USA, 2019. Association for Computing Machinery.

[KHH+01a] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. Getting started with ASPECTJ. *Com. ACM*, 44(10):59–65, 2001.

[KHH+01b] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. Getting started with AspectJ. *Commun. ACM*, 44(10):59–65, 2001.

[KKST17] Andrei Kirilenko, Albert S Kyle, Mehrdad Samadi, and Tugkan Tuzun. The flash crash: High-frequency trading in an electronic market. *The Journal of Finance*, 72(3):967–998, 2017.

[KNS16] P. Karandikar, M. Niewerth, and Ph Schnoebelen. On the state complexity of closures and interiors of regular languages with subwords and superwords. *Theoretical Computer Science*, 610:91–107, 2016.

[KYV01] Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Form. Methods Syst. Des.*, 19(3):291–314, October 2001.

[LA04] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88. IEEE Computer Society, 2004.

[Lam74] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, August 1974.

[Lam78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.

[Lam79] Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[LBM+] Doug Lea, Joshua Bloch, Sam Midkiff, David Holmes, Joseph Bowbeer, and Tim Peierls. Jsr 166: Concurrency utilities. https://www.jcp.org/en/jsr/detail?id=166.

[LBM15] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. Accurate and efficient object tracing for java applications. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, page 51–62, New York, NY, USA, 2015. Association for Computing Machinery.

[Lea00] Doug Lea. A java fork/join framework. In *Proceedings of the ACM 2000 Java Grande Conference, San Francisco, CA, USA, June 3-5, 2000*, pages 36–43, 2000.

[Leu12] Martin Leucker. Sliding between model checking and runtime verification. In Shaz Qadeer and Serdar Tasiran, editors, *Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers*, volume 7687 of *Lecture Notes in Computer Science*, pages 82–87. Springer, 2012.

[LHX+16] Owolabi Legunsen, Wajih Ul Hassan, Xinyue Xu, G. Roșu, and D. Marinov. How good are the specs? a study of the bug-finding effectiveness of existing java api specifications. *ACM*, 2016.

[Lie89] K. J. Lienberherr. Formulations and benefits of the law of Demeter. *SIGPLAN Not.*, 24(3):67–78, March 1989.

[Lig23] Lightbend Inc. Scala documentation. https://docs.scala-lang.org/, 2023. Accessed: 2023-04-30.

[LR13] Qingzhou Luo and Grigore Rosu. Enforcemop: A runtime property enforcement system for multithreaded programs. In *Proceedings of International Symposium in Software Testing and Analysis (ISSTA'13)*, pages 156–166. ACM, July 2013.

[LS09] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, May 2009.

[LSS+18] Martin Leucker, César Sánchez, Torben Scheffel, Malte Schmitz, and Alexander Schramm. TeSSLa: runtime verification of non-synchronized real-time streams. In *SAC*, pages 1925–1933, 2018.

[LT93] N.G. Leveson and C.S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[Luc05] David C. Luckham. *The power of events – An introduction to complex event processing in distributed enterprise systems*. ACM, 2005.

[Mat88] Friedemann Mattern. Virtual time and global states of distributed systems. In *PARALLEL AND DISTRIBUTED ALGORITHMS*, pages 215–226. North-Holland, 1988.

[Maz86] Antoni W. Mazurkiewicz. Trace theory. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, Germany, 8-19 September 1986*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer, 1986.

[Maz87] A Mazurkiewicz. Trace theory. In *Advances in Petri Nets 1986, Part II on Petri Nets: Applications and Relationships to Other Models of Concurrency*, page 279–324, Berlin, Heidelberg, 1987. Springer-Verlag.

[McC59] John McCarthy. A basis for a mathematical theory of computation. In *Studies in Logic and the Foundations of Mathematics*, volume 26, pages 33–70. Elsevier, 1959.

[McC76] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 1976.

[McM04] Phil McMinn. Search-based software test data generation: a survey. *Softw. Test., Verif. Reliab.*, 14(2):105–156, 2004.

[MKZ+13] Lukáš Marek, Stephen Kell, Yudi Zheng, Lubomír Bulej, Walter Binder, Petr Tůma, Danilo Ansaloni, Aibek Sarimbekov, and Andreas Sewe. Shadowvm: Robust and comprehensive dynamic program analysis for the java platform. *SIGPLAN Not.*, 49(3):105–114, October 2013.

[MP90] Zohar Manna and Amir Pnueli. A hierarchy of temporal properties (invited paper, 1989). In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, PODC '90, page 377–410, New York, NY, USA, 1990. Association for Computing Machinery.

[MP95] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag New York, Inc., 1995.

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java Memory Model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 378–391. ACM, 2005.

[MPTV22] Umang Mathur, Andreas Pavlogiannis, Hünkar Can Tunç, and Mahesh Viswanathan. A tree clock data structure for causal orderings in concurrent executions. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, page 710–725, New York, NY, USA, 2022. Association for Computing Machinery.

[MRA+17] Konstantinos Mamouras, Mukund Raghothaman, Rajeev Alur, Zachary G. Ives, and Sanjeev Khanna. StreamQRE: modular specification and efficient evaluation of quantitative queries over streaming data. In *PLDI*, pages 693–708, 2017.

[MV20] Umang Mathur and Mahesh Viswanathan. *Atomicity Checking in Linear Time Using Vector Clocks*, page 183–199. Association for Computing Machinery, New York, NY, USA, 2020.

[MVZ+12] Lukás Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. DiSL: a domain-specific language for bytecode instrumentation. In Robert Hirschfeld, Éric Tanter, Kevin J. Sullivan, and Richard P. Gabriel, editors, *Proceedings of the 11th International Conference on Aspect-oriented Software Development, AOSD, Potsdam, Germany*, pages 239–250. ACM, 2012.

[NNH15] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2015.

[OHF+14] Tony Ohmann, Michael Herzberg, Sebastian Fiss, Armand Halbert, Marc Palyart, Ivan Beschastnikh, and Yuriy Brun. Behavioral resource-aware model inference. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, page 19–30, New York, NY, USA, 2014. Association for Computing Machinery.

[Ora23a] Oracle Corporation. The java native interface specification. Technical report, 2023. Accessed: 2023-04-30.

[Ora23b] Oracle Corporation. Java platform, standard edition. https://docs.oracle.com/en/java/, 2023. Accessed: 2023-04-30.

[Ora23c] Oracle Corporation. The java virtual machine specification, java se. Technical report, 2023. Accessed: 2023-04-30.

[OU01] A. Jefferson Offutt and Roland H. Untch. *Mutation 2000: Uniting the Orthogonal*, pages 34–44. Springer US, Boston, MA, 2001.

[Pat] Patterns in property specifications for finite-state verification home page. https://matthewbdwyer.github.io/psp/patterns.html.

[PH18] Doron Peled and Klaus Havelund. Refining the safety-liveness classification of temporal properties according to monitorability. In Tiziana Margaria, Susanne Graf, and Kim G. Larsen, editors, *Models, Mindsets, Meta: The What, the How, and the Why Not? - Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday*, volume 11200 of *Lecture Notes in Computer Science*, pages 218–234. Springer, 2018.

[PRL+19] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. Renaissance: A modern benchmark suite for parallel applications on the jvm. In *Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, SPLASH Companion 2019, page 11–12, New York, NY, USA, 2019. Association for Computing Machinery.

[PWNG13] Lee Pike, Nis Wegmann, Sebastian Niller, and Alwyn E. Goodloe. Copilot: monitoring embedded systems. *Springer*, 2013.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium on programming*, pages 337–351. Springer, 1982.

[RB20] Andrea Rosà and Walter Binder. P^3: A profiler suite for parallel applications on the java virtual machine. In Bruno C. d. S. Oliveira, editor, *APLAS*, volume 12470 of *LNCS*, pages 364–372. Springer, 2020.

[RC17] Nisha Rathee and Rajender Singh Chhillar. A survey on test case generation techniques using UML diagrams. *JSW*, 12(8):643–648, 2017.

[RCR15a] Giles Reger, Helena Cuenca Cruz, and David E. Rydeheard. Marq: Monitoring at runtime with QEA. In Christel Baier and Cesare Tinelli, editors, *TACAS*, volume 9035 of *LNCS*, pages 596–610. Springer, 2015.

[RCR15b] Giles Reger, Helena Cuenca Cruz, and David E. Rydeheard. MarQ: Monitoring at Runtime with QEA. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9035 of *Lecture Notes in Computer Science*, pages 596–610. Springer, 2015.

[RGB20] Jake Roemer, Kaan Genç, and Michael D. Bond. Smarttrack: Efficient predictive race detection. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 747–762, New York, NY, USA, 2020. Association for Computing Machinery.

[RHF16] Giles Reger, Sylvain Hallé, and Yliès Falcone. Third international competition on runtime verification - CRV 2016. In Yliès Falcone and César Sánchez, editors, *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings*, volume 10012 of *Lecture Notes in Computer Science*, pages 21–37. Springer, 2016.

[Ric53] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society*, 74(2):358–366, 1953.

[RLL+13] Henrique Rebêlo, Ricardo Massa Ferreira Lima, Gary T. Leavens, Márcio Cornélio, Alexandre Mota, and César A. L. Oliveira. Optimizing generated aspect-oriented assertion checking code for JML using program transformations: An empirical study. *Sci. Comput. Program.*, 78(8):1137–1156, 2013.

[RRKH21] Massiva Roudjane, Djamal Rebaïne, Raphaël Khoury, and Sylvain Hallé. Detecting trend deviations with generic stream processing patterns. *Inf. Syst.*, 101:101446, 2021.

[RS04] G. Rosu and K. Sen. An instrumentation technique for online analysis of multithreaded programs. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pages 268–, 2004.

[Run06] Per Runeson. A survey of unit testing practices. *IEEE Software*, 23(4):22–29, 2006.

[SC] Sharkdp and Contributors. Hyperfine. https://github.com/sharkdp/hyperfine. Accessed: 2023-06-01.

[SCR12] Traian-Florin Serbanuta, Feng Chen, and Grigore Rosu. Maximal causal models for sequentially consistent systems. In *Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers*, pages 136–150, 2012.

[SEF23] Chukri Soueidi, Antoine El-Hokayem, and Yliès Falcone. Opportunistic monitoring of multithreaded programs. In Leen Lambers and Sebastián Uchitel, editors, *Fundamental Approaches to Software Engineering - 26th International Conference, FASE 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings*, volume 13991 of *Lecture Notes in Computer Science*, pages 173–194. Springer, 2023.

[Sen07] Koushik Sen. Concolic testing. In *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, pages 571–572, 2007.

[SFa] Chukri Soueidi and Yliès Falcone. Artifact for sound concurrent traces for online monitoring. https://gitlab.inria.fr/monitoring/sound-concurrent-traces-for-online-monitoring-artifact. Accessed: 2023-09-01.

[SFb] Chukri Soueidi and Yliès Falcone. Bism documentation. https://gitlab.inria.fr/bism/bism-public. Accessed: 2023-09-01.

[SFc] Chukri Soueidi and Yliès Falcone. Bism dsl experiments. https://gitlab.inria.fr/bism/bism-dsl-experiments. Accessed: 2023-09-01.

[SFd] Chukri Soueidi and Yliès Falcone. Residual runtime verification with bism. https://gitlab.inria.fr/monitoring/residual-runtime-verification-with-bism. Accessed: 2023-09-01.

[SF22a] Chukri Soueidi and Yliès Falcone. Capturing program models with bism. In Jiman Hong, Miroslav Bures, Juw Won Park, and Tomás Cerný, editors, *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, SAC '22, page 1857–1861, New York, NY, USA, 2022. Association for Computing Machinery.

[SF22b] Chukri Soueidi and Yliès Falcone. Residual runtime verification via reachability analysis. In Akash Lal and Stefano Tonetta, editors, *Verified Software. Theories, Tools and Experiments - 14th International Conference, VSTTE 2022, Trento, Italy, October 17-18, 2022, Revised Selected Papers*, volume 13800 of *Lecture Notes in Computer Science*, pages 148–166. Springer, 2022.

[SF23a] Chukri Soueidi and Ylies Falcone. Artifact Repostiory - Opportunistic Monitoring of Multithreaded Programs. https://doi.org/10.6084/m9.figshare.21828570, 2023.

[SF23b] Chukri Soueidi and Yliès Falcone. Bridging the gap: A focused dsl for rv-oriented instrumentation with bism. In *International Conference on Runtime Verification*, pages 327–338. Springer, 2023.

[SF23c] Chukri Soueidi and Yliès Falcone. Instrumentation for rv: From basic monitoring to advanced use cases. In *International Conference on Runtime Verification*, pages 403–427. Springer, 2023.

[SF23d] Chukri Soueidi and Yliès Falcone. Sound concurrent traces for online monitoring. In Georgiana Caltais and Christian Schilling, editors, *Model Checking Software - 29th International Symposium, SPIN 2023, Paris, France, April 26-27, 2023, Proceedings*, volume 13872 of *Lecture Notes in Computer Science*, pages 59–80. Springer, 2023.

[SFH23a] Chukri Soueidi, Yliès Falcone, and Sylvain Hallé. Dynamic program analysis with flexible instrumentation and complex event processing. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 742–751. IEEE, 2023.

[SFH23b] Chukri Soueidi, Yliès Falcone, and Sylvain Hallé. Monitoring business process compliance across multiple executions with stream processing. In *Proceedings of the 27th International EDOC Conference (EDOC 2023): Enterprise Design, Operations, and Computing*, 2023. To appear.

[SFH23c] Chukri Soueidi, Yliès Falcone, and Sylvain Hallé. Artifact for Dynamic Program Analysis with Flexible Instrumentation and Complex Event Processing. https://doi.org/10.5281/zenodo.8271121, August 2023.

[SGLN+11] Sriskandarajah Suhothayan, Kasun Gajasinghe, Isuru Loku Narangoda, Subash Chaturanga, Srinath Perera, and Vishaka Nanayakkara. Siddhi: A second look at complex event processing architectures. In *GCE*, page 43–50. ACM, 2011.

[SKF20] Chukri Soueidi, Ali Kassem, and Yliès Falcone. BISM: bytecode-level instrumentation for software monitoring. In Jyotirmoy Deshmukh and Dejan Nickovic, editors, *Runtime Verification - 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6-9, 2020, Proceedings*, volume 12399 of *Lecture Notes in Computer Science*, pages 323–335. Springer, 2020.

[SKV17] Julien Signoles, Nikolai Kosmatov, and Kostyantyn Vorobyov. E-ACSL, a runtime verification tool for safety and security of C programs (tool paper). In Giles Reger and Klaus Havelund, editors, *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, 2017*, volume 3 of *Kalpa Publications in Computing*, pages 164–173. EasyChair, 2017.

[SLU05] O. Spinczyk, Daniel Lohmann, and M. Urban. AspectC++: An AOP extension for C. *Software Developer's Journal*, 01 2005.

[SMF23] Chukri Soueidi, Marius Monnier, and Yliès Falcone. *International Journal on Software Tools for Technology Transfer*, pages 1–27, 2023.

[SNQDAB16] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow-and context-sensitive pointer analysis for java. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[spl] Splunk. https://www.splunk.com, Accessed May 31st, 2023.

[SR94] K.G. Shin and P. Ramanathan. Real-time computing: a new discipline of computer science and engineering. *Proceedings of the IEEE*, 82(1):6–24, 1994.

[SRA03] Koushik Sen, Grigore Rosu, and Gul Agha. Runtime safety analysis of multithreaded programs. *SIGSOFT Softw. Eng. Notes*, 28(5):337–346, September 2003.

[SY86a] R E Strom and S Yemini. Typestate: A programming language concept for enhancing software reliability. 12(1):157–171, January 1986.

[SY86b] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. on Softw. Eng.*, 12(1):157–171, 1986.

[TC10] Fuminobu Takeyama and Shigeru Chiba. An advice for advice composition in AspectJ. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6144 LNCS:122–137, 2010.

[TKH21] Rania Taleb, Raphaël Khoury, and Sylvain Hallé. Runtime verification under access restrictions. In *2021 IEEE/ACM 9th International Conference on Formal Methods in Software Engineering (FormaliSE)*, pages 31–41, 2021.

[tra] Trace Compass. https://www.eclipse.org/tracecompass, Accessed May 30th, 2023.

[Tur36] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[VBGR16] Philippe Virouleau, François Broquedis, Thierry Gautier, and Fabrice Rastello. Using data dependencies to improve task-based scheduling strategies on NUMA architectures. In *Euro-Par 2016: Parallel Processing - 22nd International Conference on Parallel and Distributed Computing, 2016, Proceedings*, volume 9833 of *Lecture Notes in Computer Science*, pages 531–544, Cham, 2016. Springer.

[vis] VisualVM: All-in-one Java troubleshooting tool. https://visualvm.github.io/, Accessed May 30th, 2023.

[VLGH17a] Simon Varvaressos, Kim Lavoie, Sébastien Gaboury, and Sylvain Hallé. Automated bug finding in video games: A case study for runtime monitoring. *Computers in Entertainment (CIE)*, 15(1):1–28, 2017.

[VLGH17b] Simon Varvaressos, Kim Lavoie, Sébastien Gaboury, and Sylvain Hallé. Automated bug finding in video games: A case study for runtime monitoring. *Comput. Entertain.*, 15(1):1:1–1:28, 2017.

[VRCG+99] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, page 13, USA, 1999. IBM Press.

[WCM13] Chengsong Wang, Zhenbang Chen, and Xiaoguang Mao. Optimizing nop-shadows typestate analysis by filtering interferential configurations. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8174 LNCS:269–284, 2013.

[WL05] Pengcheng Wu and Karl Lieberherr. Shadow programming: Reasoning about programs using lexical join point information. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3676 LNCS:141–156, 2005.

[WS06] L. Wang and S.D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Transactions on Software Engineering*, 32(2):93–110, 2006.

[ZAD+23] Pei Zhang, Alexis Aurandt, Rohit Dureja, Phillip H Jones, and Kristin Yvonne Rozier. Model predictive runtime verification for cyber-physical systems with real-time deadlines. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 158–180. Springer, 2023.

[ZCB21] Tommaso Zoppi, Andrea Ceccarelli, and Andrea Bondavalli. MADneSs: A multi-layer anomaly detection framework for complex dynamic systems. *IEEE Trans. Dependable Secur. Comput.*, 18(2):796–809, 2021.

[ZLD12] Xian Zhang, Martin Leucker, and Wei Dong. Runtime verification with predictive semantics. In *Proceedings of the 4th International Conference on NASA Formal Methods*, NFM'12, page 418–432, Berlin, Heidelberg, 2012. Springer-Verlag.

Part VI

Appendix

APPENDIX A

Other Works

We here present other works that are not directly related to the main contributions of this thesis but can benefit in later stages from the instrumentation framework presented in this thesis.

A.1 Leveraging Runtime Verification for the Monitoring of Digital Twins

[HSF23] S. Hallé, C. Soueidi, and Y. Falcone, "Leveraging Runtime Verification for the Monitoring of Digital Twins," in *Preproceedings of the Workshop on Applications of Formal Methods and Digital Twins*, Informatics Library, University of Oslo, Lübeck, Germany, Mar. 2023, Co-located with the 25th International Symposium on Formal Methods. [Online]. Available: https://www.duo.uio.no/handle/10852/101662

Abstract. The paper considers the problem of discovering divergences between the actions of a digital twin and those of its real-world counterpart. It observes the similarities between this problem and an existing field of formal methods called Runtime Verification (RV), and suggests leveraging and adapting RV techniques to this effect. Concretely, three important aspects of the problem are identified and for which both theoretical and practical challenges must be addressed.

A.2 Monitoring Business Process Compliance Across Multiple Executions with Stream Processing

[SFH23b] C. Soueidi, Y. Falcone, and S. Hallé, "Monitoring Business Process Compliance Across Multiple Executions with Stream Processing," in *Proceedings of the 27th International EDOC Conference (EDOC 2023): Enterprise Design, Operations, and Computing*, 2023, to appear.

Abstract. Compliance checking is the operation that consists of assessing whether every execution trace of a business process satisfies a given correctness condition. The paper introduces the notion of hyperquery, which is a calculation that involves multiple traces from a log at the same time. A particular case of hyperquery is a hypercompliance condition, which is a correctness requirement that involves the whole log instead of individual process instances. A formalization of hyperqueries is presented, along with a number of elementary operations to express hyperqueries on arbitrary logs. An implementation of these concepts in an event stream processing engine allows users to concretely evaluate hyperqueries in real time.

Lists

A.3 List of Definitions, Propositions, Theorems, Corollaries, Lemmas, and Examples

Definition 1	Basic Block	14
Definition 2	Control Flow Graph	15
Definition 3	Matching prefixes [BLH12]	16
Definition 4	Bad/Good prefixes [KYV01]	16
Definition 5	Property satisfaction	16
Definition 6	Projected traces	17
Definition 7	Parametric property satisfaction	17
Definition 8	Upward closure of a language	18
Definition 9	Action	19
Definition 10	Release-Acquire Relation	20
Definition 11	Thread Order $\xrightarrow{p}$	20
Definition 12	Synchronization Order $\xrightarrow{s}$	20
Definition 13	Execution Order $\xrightarrow{e}$	21
Definition 14	Concurrent Execution	21
Definition 15	Join point	46
Definition 16	Advice Function	46
Definition 17	Shadow	47
Definition 18	Method Shadows	47
Definition 19	Equivalence Relation over Shadows	48
Definition 20	Selector Function	49
Definition 21	Compile-Time Analysis Function	49
Definition 22	Instruction-Level Visibility	50

Definition 23	Transformer	50
Definition 24	Transformer Function	50
Definition 25	Composition (c)	51
Definition 26	Transformer Collision	51
Definition 27	Order-Sensitive Transformers	52
Definition 28	Residual Instrumentation Function	79
Definition 29	Residual Analysis Condition	79
Definition 30	Split CFG	80
Definition 31	CFG Automaton	81
Definition 32	Extended automaton of bad prefixes	83
Definition 33	Linear Trace	93
Definition 34	Concurrent Trace	94
Definition 35	Trace Soundness	94
Definition 36	Trace Faithfulness	94
Definition 37	Property Satisfaction	95
Definition 38	Trace Necessary Order	100
Definition 39	Trace Monitorability of Concurrent Executions	100
Proposition 1	Scope of the analysis	84
Proposition 2	Soundness of the analysis	85
Proposition 3	Property Preservation	95
Example 1	Java Method	14
Example 2	Bytecode and CFG	15
Example 3	SafeIterator monitor	16
Example 4	Parametric traces	17
Example 5	Projected traces	17
Example 6	Subwords	17
Example 7	Upward Closure	18
Example 8	Transitive Closure	18
Example 9	Linearization of a Partial Order	18
Example 10	Regular Actions	19
Example 11	Synchronization Actions	20
Example 12	Execution Order	21
Example 13	Concurrent Execution	21
Example 14	Vector Clocks	22
Example 15	Instrumentation with ASM	35
Example 16	Instrumentation with AspectJ	38
Example 17	Instrumentation with DiSL	38

Example 18	Static and Dynamic Context	45
Example 19	Join points	46
Example 20	Method shadows	48
Example 21	Equivalent shadows in a method	48
Example 22	Order Matters	52
Example 23	BISM configuration file	62
Example 24	Hidden transformers	63
Example 25	Parametric traces	78
Example 26	Projected traces	78
Example 27	CFG Automaton	81
Example 28	SafeIterator Bad Prefixes Automaton	82
Example 29	Projected traces approximation with may-alias	82
Example 30	$\mathcal{A}^{\uparrow bad_\varphi}$ for the SafeIterator property	83
Example 31	Property violating states	83
Example 32	Advice Atomicity	93
Example 33	Soundness and Faithfulness	94
Example 34	Concurrent Execution Reordering	96
Example 35	Monitor Causal Dependence	99
Example 36	Trace Necessary Order	100
Example 37	Optimizing Instrumentation	101
Example 38	Global Monitoring	107
Example 39	Events.	108
Example 40	Scope regions	109
Example 41	Local Properties	109
Example 42	Scope Properties	110

List of Figures

1.1 The typical setup of Runtime Verification. 4
1.2 An overview of the taxonomy of Runtime Verification (from [FKRT21]). 5
1.3 Guided instrumentation in RV setup. 7

2.1 Control flow graph for the method in Listing 2.1. 15
2.2 Monitor recognizing the language of bad prefixes for the *SafeIterator* property. 16
2.3 A concurrent execution of 1-Writer 2-Readers. 19

3.1 Cyclic process of model generation. 28
3.2 Considerations for RV Instrumentation 29
3.3 Program Considerations 31
3.4 Observation Considerations 31
3.5 Analysis Considerations 31
3.6 Language Considerations 31
3.7 Instrumentation language expressiveness considerations. 33

4.1 Illustration of shadows and their equivalence relation. 49

5.1 BISM overview. 56
5.2 Instrumentation loop of BISM. 58
5.3 The static context tree related to selectors and ASM nodes. 59
5.4 BISM modules with arrows indicating dependencies. 67
5.5 Instrumentation process in BISM. 68
5.6 Added modules on BISM (in blue) to support the external DSL. 68
5.7 High-level design diagram for the observation layer. Solid lines show the flow of information. The dotted lines show module dependencies. 69

6.1 A method using Iterators in Java, and its CFG. 78
6.2 The constructed CFG Automaton $\mathcal{A}_m^c$ 81
6.3 Automaton $\mathcal{A}^{bad_\varphi}$ recognizing the language of bad prefixes for the *SafeIterator* property. 82
6.4 The constructed automaton $\mathcal{A}^{\uparrow bad_\varphi}$. 83
6.5 Marking property violating paths in red, and safe in green. 84

7.1 A concurrent execution with a partial trace. 92
7.2 Instrumenting concurrent events. 93
7.3 Four different collected traces from the execution of 1-Writer 2-Readers. 95
7.4 Automata of Bad Prefixes. 99
7.5 Approaching optimal faithfulness. 101

8.1 Execution fragment of *1-Writer 2-Readers* 107
8.2 Concurrent execution fragment of 1-Writer 2-Readers. 108
8.3 Example of a scope channel for 1-Writer 2-Readers. 109

9.1 AES load-time instrumentation runtime (ms). 118
9.2 AES build-time instrumentation. 119
9.3 Financial transaction system load-time instrumentation. 120
9.4 Financial transaction system build-time instrumentation. 120
9.5 DaCapo load-time instrumentation. 121
9.6 DaCapo build-time instrumentation. 122
9.8 Evaluation for the three properties. 125
9.9 Experimentation Results. 127
9.10 Execution time (s) with vector clock algorithm running times. 128
9.11 Percentage of pairs in $\mathcal{I}_D$ grouped by pattern and alphabet size. 129
9.12 Execution time for *readers-writers* when varying the number of readers. 130
9.13 Execution time varying the number of events in the concurrency region. 131

9.14 Execution time of benchmarks. 132

11.1 BeepBeep's basic processors (adapted from [Hal18]). 152
11.2 Creating pipelines in BeepBeep (taken from [Hal18]). 153
11.3 Integrating BISM and BeepBeep 154
11.4 Instrumentation for **SafeHasNext** property. 154
11.5 Parametric monitoring of the `HasNext` property. 154
11.6 Generating the call graph of an instrumented program. 156
11.7 A complex instrumented event for file operations. 157
11.8 Calculating branch coverage pipeline. 158
11.9 Screenshot from the branch coverage dashboard. 159
11.10 Overhead comparison in BISM-BeepBeep: empty analysis vs. monitoring scenario. 160

List of Tables

5.1 Comparison of the tools. In red are features concerning expressiveness, and in blue are features concerning abstraction. ✓- Tool provides the feature, ✗- Tool does not provide the feature, ✓- Tool partially provides the feature. 70

7.1 Step by step timestamping of events from the execution in Ex. 33 using Algorithm 2. 98

9.1 A comparison between the experiments. LT is for load-time mode, and BT is for build-time mode. A checkmark (✓) indicates that the experiment involves the metric or the feature, whereas a cross mark (✗) indicates that the experiment does not involve the metric or the feature. Term NA abbreviates Not Applicable, and (-DiSL) indicates that the DiSL tool has been excluded. 117
9.2 Number of emitted events in AES experiment. 118
9.3 Number of events generated by the financial transaction system for each monitored property. 120
9.4 For each benchmark in the DaCapo experiment, the table reports the number of classes in the scope of instrumentation (Scope), the instrumented classes (Instr.), the original (Ref.) and generated bytecode size and overhead per tool, and the number of emitted events, (#) for BISM and DiSL, and AspectJ separately. 122
9.5 For each program (Bench), and property (P1), (P2), and (P3), we report # of relevant classes, methods, and instructions (Rel) producing events, number proved safe statically by our technique (Nop), # of events produced at runtime (RV) and after our analysis (RRV), improvement factor for # of instructions instrumented and events produced (Imp). K = 10^3, M = 10^6. 124
9.6 The table reports for each benchmark: the number of threads (Tr), execution time in seconds of the benchmark (Exec), # of events ($\mathbb{E}$) and their type (Type: S for synchronized, P for parallel), # of synchronization action captured (SA) for FACTS, vector clock algorithm time in sec (VCA), monitorability check **t-Mon**, # of faulty pair orderings (Faulty pairs) from linear traces. K = 10^3, M = 10^6. 127

11.1 Execution time and memory usage for each tool. 159
11.2 Comparison of JaCoCo and BISM-BeepBeep. 160
11.3 Benchmarking results for execution time. 161