Instrumentation for RV: From Basic Monitoring to Advanced Use Cases

Chukri Soueidi, Yliès Falcone

▶ To cite this version:

Chukri Soueidi, Yliès Falcone. Instrumentation for RV: From Basic Monitoring to Advanced Use Cases. RV 2023 - 23rd International Conference on Runtime Verification, Oct 2023, Thessaloniki, Greece. pp.403-427, 10.1007/978-3-031-44267-4_23 . hal-04381696

HAL Id: hal-04381696
https://inria.hal.science/hal-04381696

Submitted on 9 Jan 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Instrumentation for RV: From Basic Monitoring to Advanced Use Cases

Chukri Soueidi and Yliès Falcone

Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France
firstname.lastname@univ-grenoble-alpes.fr

Abstract. 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 least as possible. This tutorial comprehensively overviews the instrumentation process and considerations for single and multithreaded programs. We discuss often overlooked aspects in instrumenting multithreaded programs. We also cover metrics for evaluating the efficiency and effectiveness of instrumentation. We use four hands-on use cases to apply the introduced concepts and provide practical guidance on choosing and applying instrumentation for runtime verification.

1 Introduction

Ensuring software correctness often necessitates abstracting its behavior into suitable models and subsequently verifying whether these models adhere to properties. 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, 2) inserting additional code into the base program to extract information.

Choosing an instrumentation framework is a critical step in the runtime verification process. For example, bytecode manipulation libraries such as ASM [16], BCEL [9], and Soot [50] allow for extensive low-level coverage and bytecode transformations. However, implementing basic instrumentation for runtime verification with these can be quite verbose and demands a certain level of expertise from the user. On the other hand, aspect-oriented programming frameworks like AspectJ [29] provide a high-level language for specifying instrumentation. However, these frameworks have limited capabilities and introduce significant overhead. One main limitation is not being able to instrument at the bytecode level hindering their applicability in multithreaded programs and low-level monitoring scenarios. Other tools offer a balance between abstraction and expressiveness. For example, DiSL [35] offers advanced features targeting profiling, while BISM [47], which is tailored towards runtime verification, reduces overhead on execution and allows for arbitrary code insertion, which is needed for inlining code.

We cover several factors that influence the selection of an instrumentation technique in runtime verification encompassing multiple factors such as monitoring goals, observation granularity, specification, and trace semantics [24]. For instance, control-flow integrity monitoring requires low-level details [27], while typestate property monitoring focuses on high-level values [27, 22, 15]. Users can choose appropriate techniques by considering these parameters and others related to monitoring concurrent programs. Monitoring single-threaded programs relies on accurate instrumentation, while concurrent systems face challenges with event collection from multiple threads and components [12, 23]. Ensuring correct event order is vital but may incur overhead. Various tradeoffs and optimizations help reduce the overhead associated with these challenges. Furthermore, the specification and parallelism in the monitored program determine a specific abstraction of the program execution. For single-threaded programs, a linear trace with totally ordered events suffices. In contrast, multithreaded programs often require a partial order of events, necessitating low-level instrumentation to capture synchronization [46]. Alternative approaches, such as Opportunistic Monitoring [44], define a 2-level specification of monitors, requiring a different execution abstraction that includes certain assumptions.

We present four use cases demonstrating different instrumentation requirements. The first use case, Instrumentation at the Control Flow Level, focuses on low-level program instrumentation in monitoring the control flow integrity. The second use case, Residual Runtime Verification, employs static verification to identify safe execution paths and perform residual runtime verification for unproven parts. This approach is valuable when seeking to optimize monitoring overhead and reduce resource consumption. The third use case, Concurrent Traces for Online Monitoring, tackles concurrent program monitoring challenges with real-time trace collection. This case is essential for efficient monitoring in multithreaded applications that require a partial order of events and low-level instrumentation to capture synchronization. Finally, Opportunistic Monitoring deploys monitors to specific threads that exchange information only at designated synchronization points. This use case requires specific execution abstractions and assumptions aiming to reduce overhead and minimize interference with verdict delay.

2 Understanding Instrumentation

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

2.1 Unveiling the Complete Picture

Verification and analysis techniques are designed to focus on certain behavioral aspects of the system under study. Consequently, they require distinct behavioral *models* that

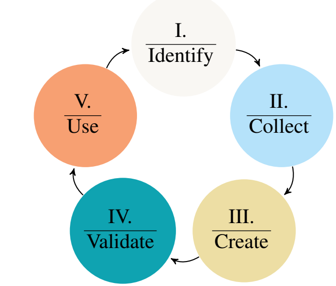


Fig. 1: Cyclic process of model generation.

accurately encapsulate the specific aspects of the system that are relevant to their intended reasoning. These models represent the *actual* behavior of the system suppressing irrelevant details and enabling the application of automated analysis. The term model is extended here to include any artifact generated to represent the system's behavior including logs, traces, automata, etc. Figure 1 illustrates the typical steps involved in the process of generating such models.

The model generation process begins by **identifying elements of interest** 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** 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**. Typically, the model is a mathematical object or a structured log, 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**. Finally, the **model is used for the intended analysis**. It can serve various purposes, such as being compared with another model in processes like model checking [21] or runtime verification [12], 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 [13, 31, 39]. Moreover, these steps can overlap, and the cycle can be reiterated for refinement.

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.

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 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's 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, this added code can be optimized by the 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.

2.3 Instrumentation for Runtime Verification

Runtime verification is a field focused on analyzing system executions, typically to verify if they satisfy or violate a particular property. A property under 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. This process usually encompasses three stages. First, a monitor is created from the property, referred to as *monitor synthesis*. This monitor interprets events from a system 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

Fig. 2: Considerations for RV Instrumentation

generator of concrete events, denoted as Σ_c, the system'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 we refer to as *shadows* and execute at specific points the in the program we refer to as *joinpoints*, and the instrumentation process consists of adding extra code at these locations to capture the concrete events, we refer to this extra code as *advice*. 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.

2.4 Instrumentation Considerations for Runtime Verification

In this section, we go through various considerations for various applications of instrumentation for runtime verification, depicted in Figure 2. We will go through some of these in the following sections and others will be covered in the rest of the tutorial.

The Program Various aspects of the program must be taken into consideration when selecting an instrumentation language. Figure 3 depicts some of these considerations. For

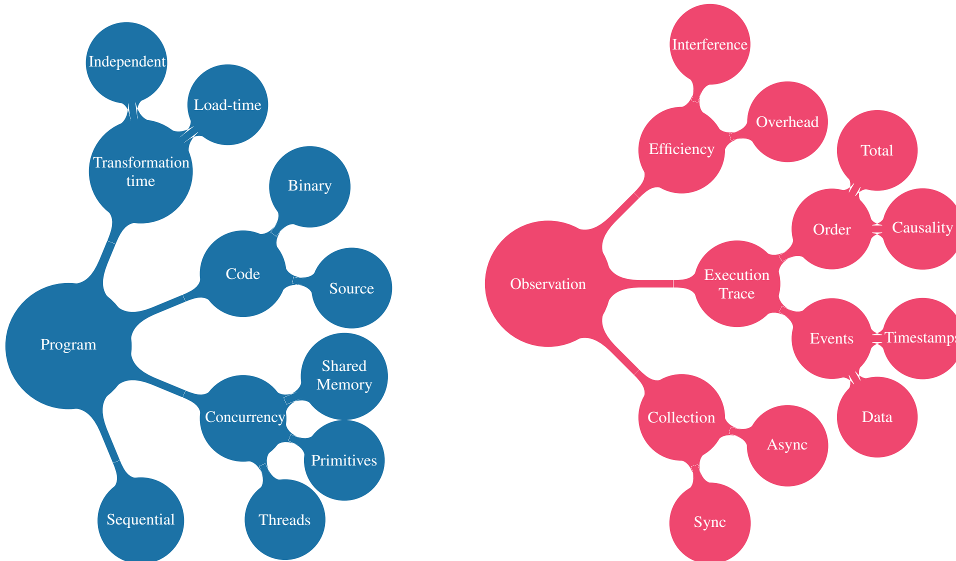


Fig. 3: Program Considerations

Fig. 4: Observation Considerations

concurrent programs, instrumentation should be capable of identifying and capturing the program's synchronization actions. These may be specified using both low-level concurrency **primitives** (such as synchronized blocks, volatile variables, lock interfaces, and atomic classes) and high-level abstractions (such as the fork-join framework and software transactional memory [32, 26]). Other applications may be implemented using message-passing frameworks such as Akka [3]. Instrumentation may be needed at different stages of the program deployment. At the **source** level, it often necessitates compilation facilities and requires access to the application's source code. Weaving at the **bytecode** 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** 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 used by the application.

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

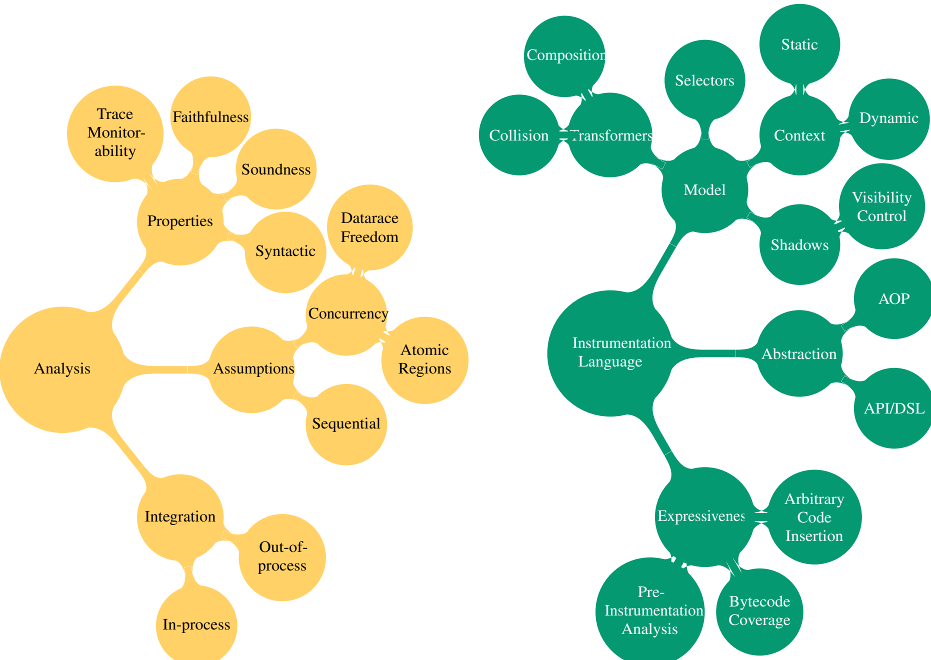


Fig. 5: Analysis Considerations

Fig. 6: Language Considerations

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 4 depicts some of the program observation considerations at runtime.

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, compatible with various formalisms for the behavior of concurrent programs like weak memory consistency models [4, 6, 34] and Mazurkiewicz traces [38, 25]. 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 [43].

The Analysis The execution analysis considerations are depicted in Figure 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** [46]

(see Sec. 7.3). 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 [49] address runtime verification with incomplete or uncertain information. Some approaches assume certain concurrency-related properties such as **data-race freedom** and **atomic regions** [44] 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.

The Instrumentation Language An instrumentation language should equip users to handle three key considerations: identifying relevant program execution points (*joinpoints*), 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 6. The usability of an instrumentation language is further characterized by two critical aspects: its **expressiveness** and the level of **abstraction**. Expressiveness refers to the language's ability to extract substantial information from the bytecode and modify the program's execution. On the other hand, abstraction relates to the complexity of low-level details that users must deal with in order to specify instrumentation.

Identifying joinpoints can be at the bytecode level or the source code level. At these joinpoints, a language should facilitate the extraction of either **static** information (compile-time information) or **dynamic** information (runtime 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 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. As for the implementation, these 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 and integrate more seamlessly with it, and are more accessible to developers familiar with the host language.

Applications In this paper, we examine five use cases that demonstrate different applications of instrumentation, which we will discuss further in Section 7. Here, we focus on the considerations relevant to selecting an instrumentation language for each of these use cases. In control-flow integrity monitoring, the program's control flow must not be altered, necessitating the reevaluation of conditional jumps by a monitor. This use case requires **bytecode coverage** to extract low-level details and control flow information. For concurrent and opportunistic monitoring, the instrumentation should be able to

capture synchronization events in the program, which often result from low-level instructions. This also requires bytecode visibility. Additionally, in some cases, instrumentation advice may need to be inserted across nonadjacent bytecode instructions, necessitating **arbitrary code insertion**. In residual runtime verification, a pre-instrumentation static analysis is used to identify safe execution paths. The instrumentation must then extract the results of this analysis to guide its own process. This scenario calls for **pre-instrumentation analysis** to pinpoint safe paths. Additionally, it requires **visibility control**, which ensures the information is properly relayed to the instrumentation.

3 Instrumentation Requirements

Correctness and Completeness Monitoring techniques assume the completeness and correctness of instrumentation in capturing events [12], 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 [28], 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.

4 How to Evaluate 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 [2, 1] 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 [42], 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 [33] can also be employed.

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.

5 Existing Instrumentation Frameworks

5.1 Bytecode Manipulation Libraries

We discuss some bytecode manipulation libraries that have been used in runtime verification tools. These libraries offer highly expressive languages for bytecode manipulation and can perform any instrumentation scenario. However, they require a good understanding of bytecode semantics. They typically provide mechanisms for program traversal, bytecode manipulation, and bytecode generation with varying levels of abstraction. *ASM [16]* for instance is a lightweight framework that provides two APIs: a visitor-based API that allows efficient traversal and manipulation of bytecode, and a tree-based API that provides a higher level of abstraction. *BCEL [9]* contains various tools like the

JustIce byte code verifier and has been used successfully in numerous applications including compilers, optimizers, and code analysis tools. *Javassist [20]* provides two levels of API: source level and bytecode level. The former does not require the developer to understand Java bytecode, making it a suitable choice for those who prefer working at a higher level of abstraction. *CGLIB [41]* is a bytecode generation library that allows developers to extend Java classes and create new ones at runtime. *Soot [50]* is a framework for analyzing and transforming Java and Android applications. It generates several intermediate representations (IRs) of the program, including Jimple, a Java-specific IR. It provides several built-in static analyses such as call-graph construction, data-flow analysis, taint analysis, and points-to analysis.

5.2 Aspect Oriented Languages

AspectJ [29] is a standard aspect-oriented programming (AOP) [30] framework that is widely used for runtime verification, debugging, and logging. AspectJ offers a rich pointcut expression language for selecting and capturing joinpoints including dynamic pointcuts which are runtime conditional expressions that selectively apply advice based on the state or context of the executing program. However, AspectJ cannot capture bytecode instructions and basic block joinpoints, which limits its usefulness in many instrumentation tasks. *DiSL [36]* is a more feature-rich tool covering bytecode-level instrumentation framework and also following an aspect-oriented approach. DiSL offers an extensible joinpoint model and provides various advanced features such as dynamic dispatch for interference avoidance among multiple instrumentations. It is a suitable framework for instrumentation-heavy dynamic analysis such as profiling.

Both AspectJ and DiSL follow the pointcut/advice model. In this approach, users can specify advice (actions meant to be executed at joinpoints) using standard Java syntax. While AspectJ wraps these actions within methods triggered at the joinpoint, DiSL weaves the advice right into the application, ensuring it's directly inlined at the joinpoint. The limitation, however, is that users can only provide code to be inserted at these joinpoints. They cannot execute or assess this code during the instrumentation phase.

5.3 The Gap Between Bytecode Libraries and AOP Frameworks

Choosing an appropriate instrumentation language largely depends on the specifics of each use case, with each requiring a unique set of features. Existing AOP languages, such as AspectJ and DiSL, simplify the specification of instrumentation. However, they also limit the user's control over the traversal of a program's bytecode. AspectJ restricts the ability to manipulate code beyond predefined join points. Consequently, performing tasks like pre-instrumentation analysis or arbitrary code insertion, that may be needed for performing an instrumentation task, can be challenging with AOP languages. They typically require ad-hoc customizations or a fallback to bytecode manipulation libraries. On the other hand, most bytecode manipulation libraries provide full control over program traversal and allow various manipulations. However, these libraries are verbose and require a high degree of expertise to use effectively. Therefore, there is a pressing need for a more comprehensive approach to instrumentation. 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. In the following section, we will introduce BISM (Bytecode-Level Instrumentation for Software Monitoring) an instrumentation framework that addresses such a need.

6 A Comprehensive Instrumentation Approach: BISM

BISM [47, 48], short for Bytecode-Level Instrumentation for Software Monitoring, is a lightweight Java instrumentation tool created for runtime verification and enforcement. Its design takes inspiration from aspect-oriented programming, but instead of following the common pointcut/advice model used by tools like AspectJ and DiSL, BISM introduces its own approach. In BISM, the instrumentation requirements are defined using *transformers*. These transformers are dedicated classes that handle both the selection of joinpoints and the inlining of advice. Unlike AspectJ and DiSL where only the advice can be specified and in plain Java, BISM requires advice to be defined using its instrumentation language. Notably, within these transformers, users have the flexibility to execute code at the time of instrumentation.

6.1 Instrumentation model

A *joinpoint* in BISM refers to a specific configuration of the base program during its execution, characterized by both static and dynamic context information. *Shadows*, on the other hand, are used to demarcate specific bytecode regions in the program. They are defined as pairs that fundamentally include a bytecode instruction identifier and a direction, either *before* or *after* the instruction. These shadows are used to specify the precise regions in the bytecode where user-specified advice will be woven. In essence, joinpoints captured by BISM correspond to these bytecode regions given by shadows. BISM selectors are designed to match specific subsets of these shadows, enabling users to select desired segments of the code for instrumentation.

6.2 Instrumentation language

BISM provides a high-level instrumentation language that allows users to specify instrumentation directives concisely. *Selectors* are used to select and capture joinpoints from the program execution. BISM provides selectors at the instruction, basic block, method, and class levels. Moreover, it provides control-flow selectors that allow users to select joinpoints based on the control-flow graph of the program. This variety of selectors allows users to specify instrumentation directives at different levels of granularity. Listing 1.1 shows the main available selectors.

```
Before/AfterInstruction          OnBasicBlockEnter/Exit
Before/AfterMethodCall           OnTrue/FalseBranchEnter
OnFieldSet/Get                   OnMethodEnter/Exit
```

Listing 1.1: BISM selectors

Within these selectors, filtering the joinpoints can be achieved with the help of guards and type patterns that support wildcard matching (see example below). Moreover, pointcuts allow users to combine multiple selectors under a single name. For contextual information extraction, BISM provides a set of *context objects* that can be used to extract the full static and dynamic information from the program. Moreover, it performs lightweight analysis on the program bytecode to provide additional out-of-the-box information about the program methods such as control-flow information and the states of the stack frames. Additionally, BISM provides *advice* methods such as method invocation `invoke` and `print` that allow the extraction of this information from the running program and also an `insert` method that allows users to inline arbitrary code at the joinpoints.

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. This approach provides users with a high degree of control over the instrumentation process. The DSL approach, on the other hand, provides a declarative way of specifying instrumentation directives. It provides a subset of the language constructs available in the API approach, but it is more concise and easier to use. The tool offers two instrumentation modes: *build-time* mode, which allows for instrumenting the compiled classes of the program, and *load-time* mode, which acts as a Java Agent that intercepts and instruments classes before linking, including some of those from the Java class library. It also includes a visualization module that displays the control-flow graphs and code changes within instrumented methods.

7 Insrumentation Use Cases

In this section, we present different use cases of instrumentation for runtime verification, some that require considerations that are often beyond the scope of existing instrumentation frameworks. We discuss limitations in addressing these use cases with well-adopted tools and how BISM can be used to address these challenges.

7.1 Classical Example

Figure 7, shows a Java method along with its control-flow graph (CFG). We are interested to monitor the **SafeIterator** property which specifies that a `Collection` should not be updated when an iterator associated with it is created and being used. This scenario can be effectively handled by all instrumentation frameworks we discuss in this paper. Listing 1.2 shows a fragment of a transformer that can be used in a parametric monitoring setup [10, 19] to instrument the program for the **SafeIterator** property. The BISM transformer, written in Java, uses the selector `afterMethodCall` to capture the return of an `Iterator` created from a `List.iterator()` method call. It uses the dynamic context object provided to retrieve the associated objects with the event, and pushes them into a list. Then, invokes a monitor passing the extracted information. Figure 1.3 shows an equivalent transformer written with the DSL.

```
pointcut pc1 after MethodCall(* *.List.iterator())
```

```
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 }
```

start → 1-9 → 10-11 → 14-16 → end (1-9 → 14-16)

Fig. 7: A method using Iterators in Java, and its CFG.

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

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

Listing 1.3: A BISM transformer that intercepts the creation of an iterator.

7.2 Residual Runtime Verification

This use case demonstrates a traditional case of integrating static and runtime verification, to mitigate the runtime monitoring overhead during the verification of safety and co-safety properties. Through static verification, the goal of *residual analysis* [22] is to identify program elements or paths that always preserve the desired property. Consequently, these paths can be ignored during runtime verification and the residual parts are then subjected to runtime verification. The residual analysis detects a set of program instructions, represented as $\mathcal{S}_P$, that can be safely excluded from runtime observation without disrupting the verification process. The aim, thus, is to define a new instrumentation function residual : $\text{instrs}^* \rightarrow (\mathcal{S}_P \rightarrow \Sigma^*)$. Let $Runs \subseteq \text{instrs}^*$ denote the set of all possible runs of the program. The program instrumented with residual should ideally produce shorter traces than instrument (the regular instrumentation function), but the monitoring results should remain consistent for both. The condition that the residual analysis should fulfill can be expressed as follows:

$$\forall r \in Runs : |\text{residual}(r)| \leq |\text{instrument}(r)| \\ \wedge\ \text{residual}(r) \models \varphi \iff \text{instrument}(r) \models \varphi$$

```
public class IteratorTransformer extends Transformer {
   public void afterMethodCall(MethodCall mc, DynamicContext dc){
   // Filter to only method of interest
   if (mc.methodName.equals("iterator") &&
       mc.methodOwner.endsWith("List")) {
      // Access to dynamic data
      DynamicValue list = dc.getMethodReceiver();
      DynamicValue iterator = dc.getMethodResult();
      // Create an ArrayList in the target method
      LocalArray la = dc.createLocalArray(mc.method, Object.class);
      dc.addToLocalArray(la,list);
      dc.addToLocalArray(la,iterator);
      // Invoke the monitor after passing arguments
      StaticInvocation sti =
         new StaticInvocation("monitors.SafeListMonitor",
             "receive");
      sti.addParameter("create");
      sti.addParameter(la);
      invoke(sti);
   }
   }
}
```

Listing 1.2: A BISM transformer written in Java that intercepts the creation of an iterator

Hence, here the instrumentation tool must incorporate the residual analysis outcomes when mapping program locations in the program; and this can be accomplished in various ways. For example, in [15], the AspectJ compiler ajc was customized to execute a similar static analysis and merge its results with the AspectJ instrumentation. However, this method requires deep knowledge of the ajc compiler for customization. In [45], the residual analysis was conducted entirely by writing BISM transformers. BISM transformers allow for writing custom logic in Java, thus, a static analyzer can be implemented as a transformer. This transformer while traversing the program and utilizes BISM control flow features to construct the control flow graph for each method. Then, it constructs a CFG automaton that abstracts the method behavior. This constructed model is needed to over-approximate the set of traces that the method might produce at runtime. The transformer then conducts a reachability analysis with a property specified as an automaton and marks the states of the CFG automaton that are safe to overlook. This first transformer performing the analysis does not instrument the base program; however, flags the safe shadows as invisible for other transformers. A second transformer then passes over the program and instruments the visible shadows that are matched by the specified selector by the user. Hence for each property, two transformers are written. Figure 8 showcases a CFG automaton, constructed from Fig. 7. In this figure, the transitions are labeled with c which denotes the creation of a list-associated iterator by calling `list.iterator()`, u denotes an update to the list via `list.add(..)`, and event n denotes a call to the `iterator.next()` method on an iterator. States marked in red correspond to shadows that will remain visible for the instrumenting transformer.

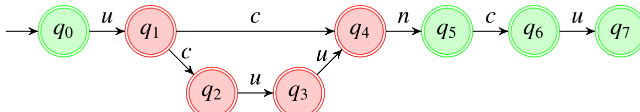


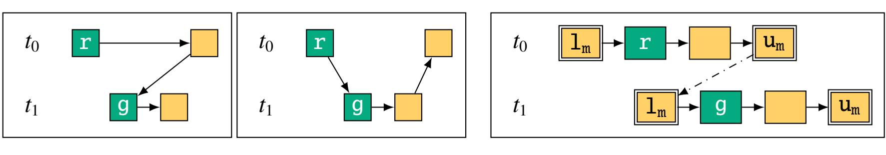
Fig. 8: Property-violating paths are marked in red, and safe ones are in green.

Discussion This scenario highlights the importance of being able to integrate static analyzers with instrumentation. Writing static analyzers with AspectJ for instance is infeasible and requires customizing the compiler such as in [14, 15, 8]. With DiSL it might be feasible, however, such analysis must be written manually using a bytecode manipulation library such as ASM in a pre-instrumentation step that does not support DiSL language. Moreover, without being able to relay the analysis results to the instrumentation, the user may need to annotate the unsafe instructions and then write custom markers within DiSL to capture those annotated shadows. This is not only tedious but also error-prone. BISM's ability to have a **composition of transformers** where transformers can **control the visibility** of shadows relaying such analysis results to the instrumentation makes such analysis feasible.

7.3 Runtime Verification of Concurrent Programs

In this scenario, we focus on the runtime verification of concurrent programs. In these programs, events may be produced by different execution flows and the order by which events are captured may not reflect the order by which they are produced. This is a challenge for runtime verification as it may lead to unsound monitoring results as shown in [23]. Suppose we want to monitor a 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 $\neg g \ \mathbf{W} \ r$. Suppose these events are being produced by two threads, t_0 and t_1, and the program is not correctly synchronized to preserve this property.

To handle concurrency, frameworks such as Java-MOP [18], Tracematches [7], and others [11] have a feature to synchronize the monitor protecting it from concurrent access and data races. However, a challenge arises when events are produced in concurrent regions as advice may not always execute atomically with their actions. Consider the example in Fig. 9a, where the code in yellow depicts the instrumented advice that notifies the monitor, a context switch might occur leading to an unsound verdict. There is a need to ensure atomicity between all executing program actions and their advice.



(a) Unsound instrumentation.

(b) Forcing atomicity.

Fig. 9: Instrumenting concurrent events.

One way to solve the lack of advice atomicity is to force it by instrumenting synchronization blocks that wrap the program actions with their advice in mutually exclusive regions. Figure Fig. 9b shows a depiction of such instrumentation. The code in Listing 1.4 shows a BISM transformer that forces atomicity between a method call and its advice. The transformer first inserts a call to a static method `getLock()` in the class `Monitor` which returns an object that will be as a lock. Then it duplicates the object on the stack and stores it in a local variable. Then it inserts a call to `MONITORENTER` which starts a `synchronized` block in Java. After the method call, the transformer inserts the code to emit the event and then loads the object from the local variable and calls `MONITOREXIT` to release the lock. This transformer can be used to instrument the example in Figure 9b to force atomicity between the method call and its advice.

```
   public class ForcingAtomicityTransformer extends Transformer {
   int lv; // local variable index
   public void beforeMethodCall(MethodCall mc, DynamicContext dc) {
      lv = ...; // the index in the method can be max local
          variables + 1
      insert(new MethodInsnNode(Opcodes.INVOKESTATIC, "Monitor",
          "getLock",
            "()Ljava/lang/Object;", false)); // retrieve the lock
                object

      insert(new InsnNode(Opcodes.DUP)); // duplicate the lock
          object on the stack
      insert(new VarInsnNode(Opcodes.ASTORE, lv)); // store one
          copy in a local variable
      insert(new InsnNode(Opcodes.MONITORENTER)); // start
          synchronized block
   }
   public void afterMethodCall(MethodCall mc, DynamicContext dc) {

      // ADVICE TO EXTRACT EVENT GOES HERE

      insert(new VarInsnNode(Opcodes.ALOAD, lv)); // load the lock
          object
      insert(new InsnNode(Opcodes.MONITOREXIT)); // end
          synchronized block
   }
}
```

Listing 1.4: A BISM transformer that forces atomicity between a method call and its advice.

Discussion An instrumentation language often aims to target specific locations within a program, to insert code either before or after a particular instruction. It is important to note that this kind of instrumentation necessitates the capability for **arbitrary code insertion**. This is because there is a need to insert related code at nonadjacent locations, both before and after the method call instruction. However, runtime verification (RV) tools that rely

on AspectJ are not suitable for instrumenting such synchronization blocks. For instance, in the case discussed in [40], the bytecode manipulation library BCEL was utilized. On the other hand, BISM offers a straightforward API to insert arbitrary instructions at any point within the program, using the ASM syntax. This unique feature allows for a hybrid approach to specifying advice. With BISM within the same transformer, the event can be extracted using abstractions, as shown in Listing 1.2, while synchronization blocks can be inserted using the ASM syntax.

Concurrent Traces Forcing atomicity introduces new problems. First, it forces a total order between concurrent program actions; interfering with the parallelism of the program and changing its behavior. One needs to avoid coarse-grained synchronization. From the monitor side, the verdict will be dependent on the specific scheduling of the execution. 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. To preserve the inherent concurrency in programs one needs to collect partial order traces instead of total order ones, capturing the "happens-before" relation among the events produced by the program. This provides a more precise representation of the program's behavior and enables a richer set of properties to be specified and checked. Two properties can be used to determine if a trace is a good representative of an execution: **soundness** and **faithfulness** [46]. 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.

As the process of observation is sequential with events being passed to a central observer reestablishing the causality between events is crucial to have trace soundness. This necessitates additional instrumentation to capture the synchronization actions from the program. These are actions that synchronize threads such as `fork(t,u)` and `begin(u)` for the initiation of thread `u` by thread `t`, `unlock(t,l)` and `lock(t,l)` for the release/acquisition of lock `l`, and `write(t,x,v)` and `read(t,x,v)` for operations on shared variable `x`. Then upon receiving these events inferring the order of events can be done with the help of a vector clock algorithm such as in [17, 37, 5, 40, 46]. To ensure faithfulness, the instrumentation must be complete in capturing all synchronization actions, preventing any loss of ordering information from the trace.

However, it's worth noting that collecting all this information to build a representative trace can be quite resource-intensive in terms of time and memory, especially in an online setup. Therefore, it might be feasible to construct this representative trace off the critical path of program execution. In [46], a concurrent trace collection mechanism that does not block the execution of the program was presented demonstrating a reduced performance impact on the running program while still capturing a representative concurrent trace.

Discussion This scenario requires instrumentation to be capable of identifying various concurrency constructs in the program, hence **bytecode coverage**. In our implementation experience, we identified a need to adapt to the diverse JVM languages, especially when considering higher-level concurrency primitives. Each JVM language, while converting

to bytecode, possesses unique features and structures. This variance poses challenges for tools primarily designed for Java, such as AspectJ. Languages like Scala, Clojure, or Kotlin produce bytecode that reflects their distinct features, from functional programming constructs and object-oriented variations to pattern matching, destructuring, static method handling, implicit parameters, and advanced concurrency constructs like coroutines. Their specialized naming conventions further add to the complexity. Consequently, this intricacy in bytecode complicates the task for Java-centric instrumentation tools requiring more specialized instrumentation tools.

7.4 Opportunistic Monitoring

We proceed in our discourse on instrumentation in the context of concurrent programs and introduce opportunistic monitoring [44] which is tailored for multithreaded programs. Monitoring here is deployed at two levels. At the first level, thread-local monitors are employed to monitor the execution of individual threads. The second level introduces *scope* monitors which monitor global properties shared accross 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 can ensure that the thread-local monitors can accurately observe and report the state of the thread within the region.

Discussion The same discussion as in the previous section applies here. Also, additional work needs to be invested to complete the automatic instrumentation and integration with monitors. So far, splitting the property over local and scope monitors is achieved manually. Analyzing the program pre-instrumentation to find and suggest scopes suitable for splitting and monitoring a given property is an interesting challenge that can be achieved within BISM.

7.5 Control Flow Integrity

In addition to capturing high-level events like method calls and executions, which are fundamental for many runtime verification tasks, instrumentation may also be needed to capture low-level events like bytecode instructions and contexts, like values on the stack and control flow. This scenario demonstrates the application of runtime verification to detect a type of control flow integrity violations namely test inversion attacks, where an attacker modifies the program's control flow by inverting conditional tests. By monitoring the execution flow of a program and logging stack values before conditional jumps, we can spot these attacks.

```
pointcut pc0 before Instruction(* *.*(..)) with (
    isConditionalJump = true)
pointcut pc1 on TrueBranchEnter(* *.*(..))
pointcut pc2 on FalseBranchEnter(* *.*(..))
```

Feature	BCEL [9]	ASM [16]	CGLIB [41]	Javassist [20]	Soot [50]	DiSL [36]	AspectJ [29]	BISM [47]
Bytecode Coverage	✓	✓	✓	✓	✓	✓	✗	✓
Bytecode Insertion	✓	✓	✓	✓	✓	✗	✗	✓
No Bytecode Proficiency	✗	✗	✗	✗	✓	✓	✓	✓
Pre-Instrumentation Analysis	✓	✓	✓	✓	✓	✗	✗	✓
High-Level Abstraction	✗	✗	✗	✓	✓	✓	✓	✓
AOP Paradigm	✗	✗	✗	✗	✗	✓	✓	✗

Table 1: Comparison of the tools. ✓- Tool provides the feature, ✗- Tool does not provide the feature, ✗- Tool partially provides the feature

```
event e0([opcode,getStackValues]) on pc0 to Mon.recieve(List)
event e1("tt") on pc1 to Mon.recieve(String)
event e2("ff") on pc2 to Mon.recieve(String)
```

Listing 1.5: A BISM spec that intercepts the creation of an iterator.

Here, a monitor can be created to check the evaluation of the stack values with the opcode of the conditional jump to detect test inversion attacks [27]. Moreover, as mitigation for this type of attack, one can inline a small monitor at each branch of the conditional statement that rechecks the evaluation of the stack values and throws an exception if the evaluation results in a different value than the one logged by the entered branch. Here the instrumentation is required to duplicate the stack values and then insert conditional instructions within each branch.

Discussion To handle such instrumentation requirements, the instrumentation tool must first have **bytecode coverage** to capture low-level events like execution of bytecode instructions and contexts, like values on the stack and control flow. Moreover, the instrumentation should be able to identify branching locations which requires constructing the control flow graph of the program. Finally, to inline monitors, instrumentation needs to duplicate the stack values and the conditional statements within each branch. This requires the ability to insert **arbitrary bytecode instructions**. Tools like AspectJ and DiSL are incapable of inserting arbitrary bytecode instructions and thus cannot be used.

8 Conclusion

In this tutorial, we presented an overview of instrumentation for runtime verification. We discussed the main considerations that should be taken into account when instrumenting a program for runtime verification. We also presented the main instrumentation techniques and tools that can be used for runtime verification. Table 1 shows a comparison between them. We also discussed the main challenges and pitfalls that can be encountered during the instrumentation process. We hope that this tutorial will help researchers and practitioners to better understand the instrumentation process and to choose the most suitable instrumentation technique and tool for their needs.

Bibliography

[1] JProfiler. https://www.ej-technologies.com/products/jprofiler/overview.html, accessed: 2023-06-01

[2] VisualVM: All-in-one Java troubleshooting tool, https://visualvm.github.io/, Accessed May 30th, 2023

[3] Akka documentation. http://akka.io/docs/ (2022), http://akka.io/docs/

[4] Adve, S.V., Gharachorloo, K.: Shared memory consistency models: a tutorial. Computer **29**(12), 66–76 (Dec 1996). https://doi.org/10.1109/2.546611

[5] Agarwal, A., Garg, V.K.: Efficient dependency tracking for relevant events in shared-memory systems. In: Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing. p. 19–28. PODC '05, Association for Computing Machinery, New York, NY, USA (2005). https://doi.org/10.1145/1073814.1073818, https://doi.org/10.1145/1073814.1073818

[6] Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., Hutto, P.W.: Causal memory: definitions, implementation, and programming. Distributed Computing **9**(1), 37–49 (Mar 1995). https://doi.org/10.1007/BF01784241

[7] Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: 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. pp. 345–364. OOPSLA '05, ACM (2005). https://doi.org/10.1145/1094811.1094839

[8] Aotani, T., Masuhara, H.: Scope: An aspectj compiler for supporting user-defined analysis-based pointcuts. In: Proceedings of the 6th International Conference on Aspect-Oriented Software Development. p. 161–172. AOSD '07, Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1218563.1218582, https://doi.org/10.1145/1218563.1218582

[9] Apache Commons: BCEL (byte code engineering library). https://commons.apache.org/proper/commons-bcel, accessed: 2020-06-18

[10] Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.E.: Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7436, pp. 68–84. Springer (2012). https://doi.org/10.1007/978-3-642-32759-9_9, https://doi.org/10.1007/978-3-642-32759-9_9

[11] Bartocci, E., Falcone, Y., Bonakdarpour, B., Colombo, C., Decker, N., Havelund, K., Joshi, Y., Klaedtke, F., Milewicz, R., Reger, G., Rosu, G., Signoles, J., Thoma, D., Zalinescu, E., Zhang, Y.: First international competition on runtime verification: rules, benchmarks, tools, and final results of CRV 2014. International Journal on Software Tools for Technology Transfer (Apr 2017). https://doi.org/10.1007/s10009-017-0454-5

[12] Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification -

Introductory and Advanced Topics, Lecture Notes in Computer Science, vol. 10457, pp. 1–33. Springer (2018). https://doi.org/10.1007/978-3-319-75632-5_1

[13] Biermann, A.W., Feldman, J.A.: On the synthesis of finite-state machines from samples of their behavior. IEEE Trans. Comput. **21**(6), 592–597 (jun 1972). https://doi.org/10.1109/TC.1972.5009015, https://doi.org/10.1109/TC.1972.5009015

[14] Bodden, E., Havelund, K.: Racer: Effective race detection using aspectj. In: Proceedings of the 2008 International Symposium on Software Testing and Analysis. p. 155–166. ISSTA '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1390630.1390650, https://doi.org/10.1145/1390630.1390650

[15] Bodden, E., Lam, P., Hendren, L.: Clara: A framework for partially evaluating finite-state runtime monitors ahead of time pp. 183–197 (01 2010). https://doi.org/10.1007/978-3-642-16612-9_15

[16] Bruneton, E., Lenglet, R., Coupaye, T.: ASM: A code manipulation tool to implement adaptable systems. In: Adaptable and extensible component systems (2002), https://asm.ow2.io

[17] Cain, H.W., Lipasti, M.H.: Verifying sequential consistency using vector clocks. In: Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures. p. 153–154. SPAA '02, Association for Computing Machinery, New York, NY, USA (2002). https://doi.org/10.1145/564870.564897, https://doi.org/10.1145/564870.564897

[18] Chen, F., Roşu, G.: Java-MOP: A Monitoring Oriented Programming Environment for Java. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 546–550. Lecture Notes in Computer Science, Springer (Apr 2005). https://doi.org/10.1007/978-3-540-31980-1_36

[19] Chen, F., Roşu, G.: Parametric trace slicing and monitoring. In: Kowalewski, S., Philippou, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 246–261. Springer Berlin Heidelberg, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_23

[20] Chiba, S.: Load-time structural reflection in java. In: Bertino, E. (ed.) ECOOP 2000 - Object-Oriented Programming, 14th European Conference, Sophia Antipolis and Cannes, France, June 12-16, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1850, pp. 313–336. Springer (2000). https://doi.org/10.1007/3-540-45102-1_16

[21] Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R.: Handbook of Model Checking. Springer Publishing Company, Incorporated, 1st edn. (2018)

[22] Dwyer, M.B., Purandare, R.: Residual dynamic typestate analysis exploiting static analysis p. 124 (2007)

[23] El-Hokayem, A., Falcone, Y.: Can we monitor all multithreaded programs? In: Colombo, C., Leucker, M. (eds.) Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11237, pp. 64–89. Springer (2018). https://doi.org/10.1007/978-3-030-03769-7_6, https://doi.org/10.1007/978-3-030-03769-7_6

[24] Falcone, Y., Krstic, S., Reger, G., Traytel, D.: A taxonomy for classifying runtime verification tools. In: Colombo, C., Leucker, M. (eds.) Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018,

Proceedings. Lecture Notes in Computer Science, vol. 23, pp. 241–262. Springer (2018). https://doi.org/10.1007/s10009-021-00609-z

[25] Gastin, P., Kuske, D.: Uniform satisfiability problem for local temporal logics over Mazurkiewicz traces. Inf. Comput. **208**(7), 797–816 (2010). https://doi.org/10.1016/j.ic.2009.12.003

[26] Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. p. 48–60. PPoPP '05, Association for Computing Machinery, New York, NY, USA (2005). https://doi.org/10.1145/1065944.1065952, https://doi.org/10.1145/1065944.1065952

[27] Kassem, A., Falcone, Y.: Detecting fault injection attacks with runtime verification. In: Proceedings of the 3rd ACM Workshop on Software Protection. p. 65–76. SPRO'19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3338503.3357724, https://doi.org/10.1145/3338503.3357724

[28] Kell, S., Ansaloni, D., Binder, W., Marek, L.: 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. p. 33–38. VMIL '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2414740.2414747, https://doi.org/10.1145/2414740.2414747

[29] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: Getting started with AspectJ. Commun. ACM **44**(10), 59–65 (2001)

[30] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: ECOOP'97—Object-Oriented Programming: 11th European Conference Jyväskylä, Finland, June 9–13, 1997 Proceedings 11. pp. 220–242. Springer (1997)

[31] Krka, I., Brun, Y., Medvidovic, N.: Automatic mining of specifications from invocation traces and method invariants. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. p. 178–189. FSE 2014, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2635868.2635890, https://doi.org/10.1145/2635868.2635890

[32] Lea, D.: A java fork/join framework. In: Proceedings of the ACM 2000 Java Grande Conference, San Francisco, CA, USA, June 3-5, 2000. pp. 36–43 (2000). https://doi.org/10.1145/337449.337465, https://doi.org/10.1145/337449.337465

[33] Lengauer, P., Bitto, V., Mössenböck, H.: Accurate and efficient object tracing for java applications. In: Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering. p. 51–62. ICPE '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2668930.2688037, https://doi.org/10.1145/2668930.2688037

[34] Manson, J., Pugh, W., Adve, S.V.: The Java Memory Model. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 378–391. POPL '05, ACM (2005). https://doi.org/10.1145/1040305.1040336

[35] Marek, L., Villazón, A., Zheng, Y., Ansaloni, D., Binder, W., Qi, Z.: DiSL: a domain-specific language for bytecode instrumentation. In: Hirschfeld, R., Tanter, É., Sullivan, K.J., Gabriel, R.P. (eds.) Proceedings of the 11th International Confer-

ence on Aspect-oriented Software Development, AOSD, Potsdam, Germany. pp. 239–250. ACM (2012)

[36] Marek, L., Villazón, A., Zheng, Y., Ansaloni, D., Binder, W., Qi, Z.: DiSL: a domain-specific language for bytecode instrumentation. In: Hirschfeld, R., Tanter, É., Sullivan, K.J., Gabriel, R.P. (eds.) Proceedings of the 11th International Conference on Aspect-oriented Software Development, AOSD, Potsdam, Germany. pp. 239–250. ACM (2012)

[37] Mathur, U., Viswanathan, M.: Atomicity Checking in Linear Time Using Vector Clocks, p. 183–199. Association for Computing Machinery, New York, NY, USA (2020), https://doi.org/10.1145/3373376.3378475

[38] Mazurkiewicz, A.W.: Trace theory. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) 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. Lecture Notes in Computer Science, vol. 255, pp. 279–324. Springer (1986). https://doi.org/10.1007/3-540-17906-2_30

[39] Ohmann, T., Herzberg, M., Fiss, S., Halbert, A., Palyart, M., Beschastnikh, I., Brun, Y.: Behavioral resource-aware model inference. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. p. 19–30. ASE '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2642937.2642988, https://doi.org/10.1145/2642937.2642988

[40] Rosu, G., Sen, K.: An instrumentation technique for online analysis of multithreaded programs. In: 18th International Parallel and Distributed Processing Symposium, 2004. Proceedings. pp. 268– (2004). https://doi.org/10.1109/IPDPS.2004.1303344

[41] Sam Berlin et al.: CGLIB (byte code generation library). https://github.com/cglib/cglib, accessed: 2021-05-21

[42] Sharkdp, Contributors: Hyperfine. https://github.com/sharkdp/hyperfine, accessed: 2023-06-01

[43] Shin, K., Ramanathan, P.: Real-time computing: a new discipline of computer science and engineering. Proceedings of the IEEE **82**(1), 6–24 (1994). https://doi.org/10.1109/5.259423

[44] Soueidi, C., El-Hokayem, A., Falcone, Y.: Opportunistic monitoring of multithreaded programs. In: Lambers, L., Uchitel, S. (eds.) 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. Lecture Notes in Computer Science, vol. 13991, pp. 173–194. Springer (2023). https://doi.org/10.1007/978-3-031-30826-0_10, https://doi.org/10.1007/978-3-031-30826-0_10

[45] Soueidi, C., Falcone, Y.: Residual runtime verification via reachability analysis. In: Lal, A., Tonetta, S. (eds.) Verified Software. Theories, Tools and Experiments - 14th International Conference, VSTTE 2022, Trento, Italy, October 17-18, 2022, Revised Selected Papers. Lecture Notes in Computer Science, vol. 13800, pp. 148–166. Springer (2022). https://doi.org/10.1007/978-3-031-25803-9_9, https://doi.org/10.1007/978-3-031-25803-9_9

[46] Soueidi, C., Falcone, Y.: Sound concurrent traces for online monitoring. In: Caltais, G., Schilling, C. (eds.) Model Checking Software - 29th International Symposium, SPIN 2023, Paris, France, April 26-27, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13872, pp. 59–80. Springer (2023). https://doi.org/10.1007/978-3-031-32157-3_4, https://doi.org/10.1007/978-3-031-32157-3_4

[47] Soueidi, C., Kassem, A., Falcone, Y.: BISM: Bytecode-Level Instrumentation for Software Monitoring (2020). https://doi.org/10.1007/978-3-030-60508-7_18, https://gitlab.inria.fr/bism/bism-public/

[48] Soueidi, C., Monnier, M., Falcone, Y.: International Journal on Software Tools for Technology Transfer pp. 1–27 (2023). https://doi.org/10.1007/s10009-023-00708-z, https://link.springer.com/article/10.1007/s10009-023-00708-z

[49] Taleb, R., Khoury, R., Hallé, S.: Runtime verification under access restrictions. In: 2021 IEEE/ACM 9th International Conference on Formal Methods in Software Engineering (FormaliSE). pp. 31–41 (2021). https://doi.org/10.1109/FormaliSE52586.2021.00010

[50] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot - a java bytecode optimization framework. In: Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research. p. 13. CASCON ’99, IBM Press, USA (1999). https://doi.org/10.1145/1925805.1925818