Dynamic Program Analysis with Flexible Instrumentation and Complex Event Processing

Chukri Soueidi, Yliès Falcone, Sylvain Hallé

Dynamic Program Analysis with Flexible Instrumentation and Complex Event Processing

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

Sylvain Hallé
Université du Québec à Chicoutimi
Chicoutimi, Canada
shalle@acm.org

***Abstract*—This paper presents a flexible and modular approach to dynamic program analysis for JVM-based languages, aiming to address the limitations of existing tools, in particular their limited expressivity and tight coupling between instrumentation and analysis. The proposed solution decouples these two processes using BISM, a lightweight instrumentation language, and BeepBeep, a complex event processing engine. This novel combination enhances expressiveness, promotes reusability, 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.**

I. Introduction

As software systems continue to grow in complexity, ensuring their reliability is becoming increasingly crucial [48]. Developers need to comprehend these systems thoroughly and regularly test and validate their functionality and behavior. Several complementary approaches are available to developers to ensure the quality and reliability of the systems for which they are responsible. One effective way to achieve these objectives is through *dynamic program analysis* [25]. This term encompasses a set of techniques 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 [13] or examine the cause of possible deadlocks in the presence of multiple threads [12], [40]. A program can also be instrumented manually using *logging statements* [21]; the resulting logs can be analyzed to reveal anomalies in the operation of a system [32], process telemetry or troubleshoot problems [15]. *Runtime verification* evaluates formal properties related to the correctness of a running program as its execution unfolds [18]. Finally, dynamic program analysis can also complement existing *software testing* activities, for example by evaluating coverage metrics on a test suite is being run [19].

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.

In this paper, we introduce a flexible and modular approach to dynamic program analysis that effectively addresses these issues. Our approach, designed for JVM-based languages including Java, Scala, Kotlin, and Groovy, decouples the instrumentation and analysis process. We utilize BISM [44], a lightweight instrumentation language, to instrument programs and extract events. BISM provides a high-level intuitive instrumentation language inspired by the aspect-oriented programming (AOP) paradigm while also providing low-level bytecode coverage. Simultaneously, we employ a complex event processing engine, BeepBeep [27], to perform a diverse range of analyses on the collected information. 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.

The rest of the paper is organized as follows. Section II gives an overview of existing solutions in dynamic program analysis and discusses their limitations. Section III provides background information on BISM and BeepBeep. Section IV 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 V compares the performance of the tool with existing solutions; finally Section VI discusses future work.

II. Dynamic Program Analysis

As opposed to static program analysis, which reasons over the behavior of a system without executing it, dynamic program analysis encompasses all techniques which collect information about a program while it is running.

A. Existing Approaches

Existing solutions can be divided into four categories, which we briefly describe below. 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.

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 [11], 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 [10] or JProbe [7], provide similar functionalities.

2) 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 [8], GoAccess [4] and Splunk [9] 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 [28], Palisade [33] and MADneSs [49]. 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 [3] and Siddhi [46] —although we find no record of their use for dynamic program analysis.

3) Runtime Verification: This line of study focuses on the observation of a system during its execution in order to detect potential violations of a formal specification describing its expected behavior [18]. In this setting, a *property* is an expression defining a subset of all possible execution traces produced by the system. A *monitor* is synthesized from this property, and is responsible for collecting events produced by the running system and determining, at each point in time, whether the property is satisfied or violated. To this end, the system under study must be instrumented in order to generate events at appropriate moments in time and relay them to the monitor.

Early works on runtime verification focused on the verification of contracts on Java classes. In many cases, aspect-oriented programming is used to automatically add instrumentation points without modification to the original program; a notable proponent of this technique is the JavaMOP framework [22], which uses AspectJ [34] to weave the execution of the monitor inside the program. The specification languages in which properties can be expressed include variants of finite-state machines [39], stream equations [24], and temporal logic [41]).

4) 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 [5], 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 [6], 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.* [29], 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 [19], which supports block, branch, and condition coverage, as well as covering calls to an error-function.

B. 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.

1) 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.

2) 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 [30] within JaCoCo. Secondly, the expressiveness of the analysis can be limited. Take the runtime verification tool JavaMOP [22], for example, which relies on AspectJ [1] for instrumentation. Although, JavaMOP provide multiple plugins to express properties using different specification languages such as LTL and 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 IV-D) 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 [38] 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. However, ShadowVM is not very portable as performing dynamic analysis 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.

III. An Overview of BISM and BeepBeep

As we have seen in the review of existing solutions, there is a need for a dynamic program analysis approach that is both generic, offering great flexibility both in terms of instrumentation and in the processing performed on the data collected (§II-B1), and where analysis and instrumentation are as loosely coupled as possible (§II-B2).

A. Motivation

Instrumentation is integral to dynamic analysis allowing users to identify and extract relevant information from a running program. Selecting the right instrumentation framework becomes key in this context. For example, bytecode manipulation libraries such as ASM [20] and Soot [47] 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 [1] provide a high-level language for specifying instrumentation. However, this framework has significant limitations. It lacks bytecode coverage restricting its capability to capture low-level events, including individual bytecode instructions. Moreover, with AspectJ, creating concise static analyzers for pre-instrumentation needs is infeasible. 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, BISM is a tool that effectively bridges the gaps between both approaches and offers a comprehensive solution to the instrumentation needs of dynamic analysis [44].

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 II-A. It is true that frameworks such as BISM or AspectJ 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 on the data extracted from the execution of said program. However, this offloads on the user the responsibility of programming from scratch what 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 [27] 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

```
pointcut pc0 before Instruction(*.xyz.*) with (isConditionalJump
    ↪ = true)

event e0([opcode,getStackValues]) on pc0 to console(List)
```

Figure 1: A BISM transformer to intercept conditional jumps.

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 [26].

To this end, the solution we propose leverages and extends two existing frameworks, the BISM instrumentation tool and the BeepBeep stream processing engine. In the following, we briefly describe these two systems.

B. BISM

BISM [44] is a bytecode-level instrumentation tool for Java that is inspired by the aspect-oriented programming style to define instrumentation requirements. It has been utilized for integrating static analysis with runtime verification [43]. In BISM, the instrumentation specifications are specified through various *transformers*. These transformers encapsulate *joinpoint* selection (dynamic points during execution) and the collection of contextual information. Unlike the pointcut/advice model of AspectJ, which permits only the specification of the code to be injected into the instrumented program, BISM allows for the execution of analysis code at the time of instrumentation. 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.

1) Instrumentation language: The instrumentation language provides constructs to handle three key functions. It enables users to identify points of interest within the program execution, referred to as joinpoints. *Selectors* capture these joinpoints, where each selector is associated with a specific region in the bytecode, such as a single bytecode instruction, control-flow branch, or method call. Within these selectors, filtering the joinpoints can be achieved with the help of guards and type patterns that support wildcard matching (see example below). BISM also performs some analysis on the program bytecode to provide additional information about the program methods such as control-flow information and the states of the stack frames. Moreover, pointcuts allow users to combine multiple selectors under a single name. At the selected points of interest, the languages provides access to comprehensive static and dynamic *context objects*, allowing users to extract compile-time derived information or runtime-specific values. Additionally, *advice* methods allow the insertion of arbitrary bytecode instructions, method invocation and printing for the extraction of this information from the running program.

2) Transformers: The tool offers two transformer implementation methods: API-based and DSL. The API method lets users define transformers using Java, granting extensive control over instrumentation. Meanwhile, the DSL method offers a concise, user-friendly way to specify directives, though with a limited set of the API's language constructs. Figure 1 shows a transformer written with the BISM DSL. It uses the `Instruction` selector with a guard to intercept the execution of conditional jumps in a package `xyz`. It then uses the advice method `console` to print the event on the console. Here is a sample program (assuming it is in `xyz` package) and its output when instrumented and executed.

```
Program: int x = 0;  if (x == 5) {}
Output: IF_ICMPNE, [0, 5]
```

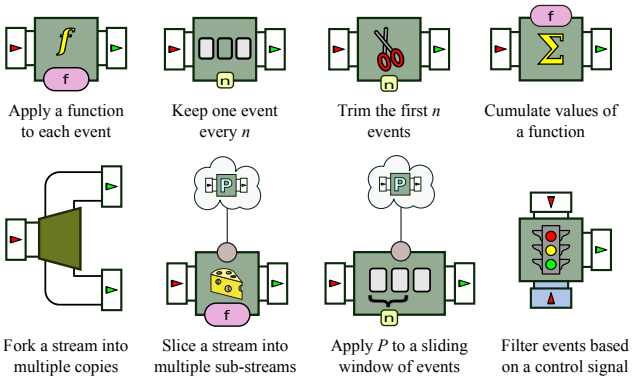


Figure 2: BeepBeep's basic processors (adapted from [27]).

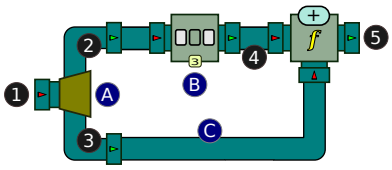
C. BeepBeep

BeepBeep is a generic open source event stream processing library developed in Java [27]. 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 together to form networks performing complex calculations.

1) 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 2.

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 "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



(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).connect(c,0,a,0);
```

(b) Code equivalent

Figure 3: Creating pipelines in BeepBeep (taken from [27]).

into multiple sub-streams, and passes each sub-stream into its own instance of some other processor P.

2) 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 3 shows a simple processor chain, both as a code snippet and as a graphical representation.

IV. The BISM-BeepBeep Integration Layer

We now describe tour dynamic program analysis framework that integrates a powerful runtime instrumentation mechanism with an expressive processing library to perform analysis over the events generated by the execution of a program. This is accomplished through an integration layer that bridges BISM events to BeepBeep pipelines.

We then present a selection of analyses, that we implemented using our integration layer targeting Java programs[1]. 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.

A. Architecture

Within each analysis, users specify the instrumentation requirements using the BISM language and the analysis using BeepBeep processors. We implemented an integration layer that bridges the communication between the running instrumented program and BeepBeep. This layer orchestrates a flexible, scalable pipeline whereby the program is the producer of

[1]The artifact containing the source code for the integration layer, the following examples and experiments is available at https://doi.org/10.5281/zenodo.8271121.

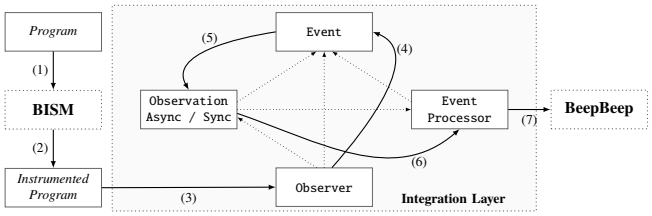


Figure 4: Diagram for the integration layer. Solid lines show information flow. Dotted lines show module dependencies.

events that are then dispatched to BeepBeep for analysis. It is composed of 4 main modules that work together to receive, collect, and send events to BeepBeep. Figure 4 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 BeepBeep `Pump` processor running on 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 BeepBeep processors 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.

The integration layer can also provide compile-time facilities to write static analyzers for the base program. Since the joinpoint model in BISM matches method and constructor call sites, method bodies, and field read accesses, it can be used as a traversal strategy for the base program. A static analyzer, defined with BeepBeep, can receive events from BISM notifying it when visiting methods, calls, etc.

B. 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.

1) Parametric Monitoring: For Java programs, typestate properties [45] are essential for enforcing precise and structured

```
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 5: Instrumentation for **SafeHasNext** property.

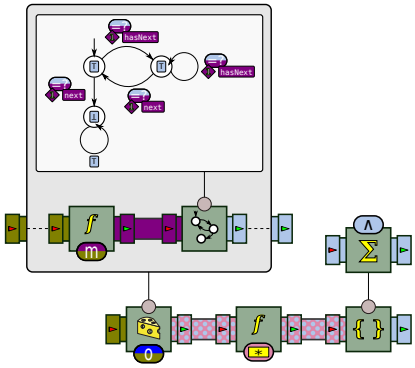


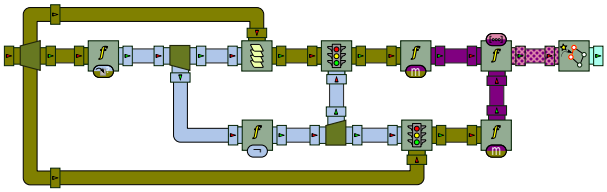
Figure 6: Parametric monitoring of the `HasNext` property.

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).

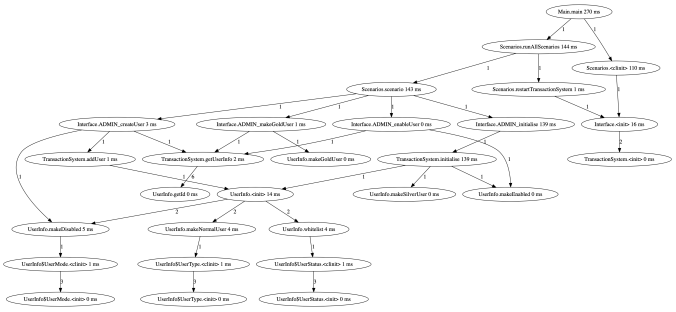
We implemented as a slicing function the algorithm from [23]. Figure 5 shows the transformer needed to instrument the program with BISM. The integration layer exposes the `Observer.observe` function which accepts 2 lists: one for event keys and one for event values. As such at each relevant joinpoint, such events are extracted. Figure 6 shows the pipeline to monitor the **SafeHasNext** property[2]. 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.

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

[2]The property states that a program does not call the `next` method before calling the `hasNext` method of an iterator.



(a) Pipeline



(b) Call graph

Figure 7: Generating the call graph of an instrumented program.

enables users to implement and experiment with various slicing approaches such as [14], [16], [23] 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 [42].

2) 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 [31] 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` [2] library. These regular expressions allow users to specify *bad* or *good* prefixes [35] 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.

C. 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 7a, 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 7b, shows the dynamic call graph corresponding to a run of the financial transaction system from [17].

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.

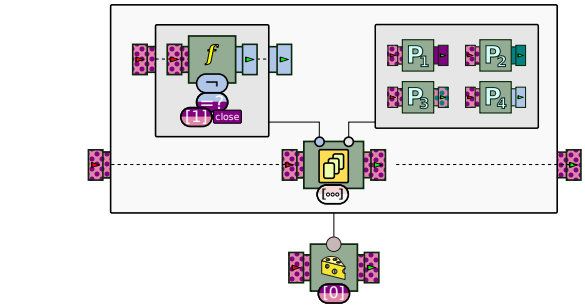
The dynamic call graph is a valuable tool for dynamic analysis providing insight into the behavior of a program during execution, and enabling developers to identify and address performance, security, and other issues in the program. Without using a dedicated profiler, extracting such a graph would require 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.

D. Log Analysis: Complex Instrumented Events

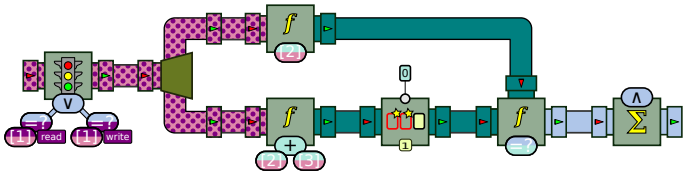
Our next application example leverages the fact that the processing layer of our analysis tool makes use of a full-fledged event stream processing engine, which, in particular, provides functionalities for Complex Event Processing (CEP) [36]. The principle behind CEP is not to merely apply calculations 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 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



(a) High-level pipeline



(b) Processor P_4 checking if accesses are contiguous

Figure 8: A complex instrumented event for file operations.

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 [37] by allowing patterns not expressible as regular expressions.

In the case of the file access complex events described above, Figure 8a 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 8b. 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.

E. 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

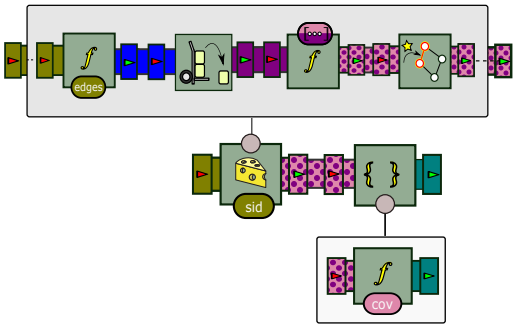


Figure 9: Calculating branch coverage pipeline.

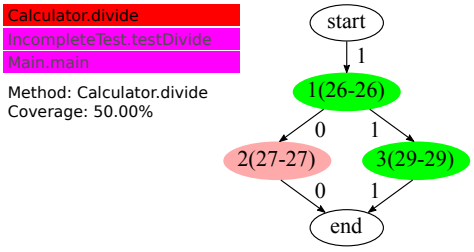


Figure 10: Screenshot from the branch coverage dashboard.

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.

1) 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 9 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 path from the stack trace and view a detailed representation of the corresponding method's CFG (at the leaf node). Figure 10 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.

2) Indirect Coverage: Indirect coverage is a metric introduced by [30], 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 [30] provide none). Reusing the *same* instrumentation specification from Section IV-E1, 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 suite with a ratio of 1, meaning that none of its entities where directly covered.

V. 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.

A. Monitoring

We compared the performance of our parametric monitors with well-established monitoring tools such as JavaMOP [22], and MARQ [39] 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 I. 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.

Figure 11 depicts the memory and execution time overhead for multiple executions with varying numbers of events,

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 I: Execution time and memory usage for each tool.

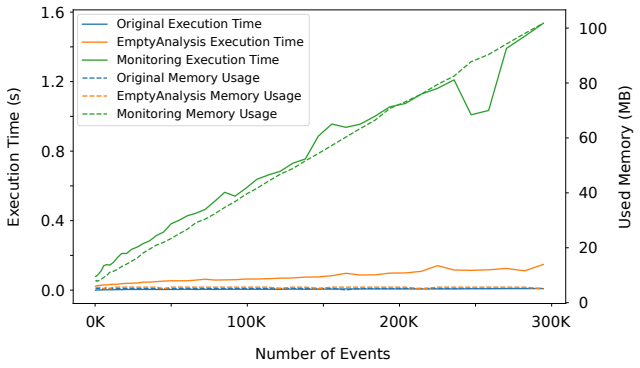


Figure 11: Overhead comparison in BISM-BeepBeep: empty analysis vs. monitoring scenario.

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** 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.

B. Coverage

We compared the performance of our branch coverage analysis with JaCoCo [5] using a benchmark that simulates a financial transaction system [17]. This comparison varied the number of produced events across different runs. As depicted in Table II, 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 II: Comparison of JaCoCo and BISM-BeepBeep.

C. 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 III, using the same financial transaction system [17]. Our approach utilized BeepBeep's existing abstractions for 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.

VI. Conclusion

In this paper, we 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 by decoupling the 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

Method	Execution Time (ms)	Processing (ms)
Manual Code with AspectJ	2400	110
BISM-BeepBeep	1750	290
Original	800	-

Table III: Benchmarking results for execution time.

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.

References

[1] AspectJ. https://www.eclipse.org/aspectj/. Accessed: 2023-05-01.

[2] BRICS Automaton. http://www.brics.dk/automaton/. Accessed: 2023-05-30.

[3] Esper. https://www.espertech.com/esper, Accessed May 31st, 2023.

[4] GoAccess. https://goaccess.io, Accessed May 31st, 2023.

[5] JaCoCo Java code coverage library. https://www.jacoco.org/. Accessed: 2023-05-01.

[6] JCov. https://wiki.openjdk.org/display/CodeTools/jcov, Accessed May 31st, 2023.

[7] JProbe suite: Complete Java performance tools suite. http://tan.com/jprobe, Accessed May 30th, 2023.

[8] ManageEngine EventLog Analyzer. https://www.manageengine.com/products/eventlog, Accessed May 31st, 2023.

[9] Splunk. https://www.splunk.com, Accessed May 31st, 2023.

[10] Trace Compass. https://www.eclipse.org/tracecompass, Accessed May 30th, 2023.

[11] VisualVM. https://visualvm.github.io/, Accessed May 30th, 2023.

[12] L. Adhianto, S. Banerjee, M. W. Fagan, M. Krentel, G. Marin, J. M. Mellor-Crummey, and N. R. Tallent. HPCTOOLKIT: tools for performance analysis of optimized parallel programs. *Concurr. Comput. Pract. Exp.*, 22(6):685–701, 2010.

[13] R. W. Ahmad, A. Gani, S. H. A. Hamid, F. Xia, and M. 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.

[14] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. *SIGPLAN Not.*, 40(10):345–364, oct 2005.

[15] T. Barik, R. DeLine, S. M. Drucker, and D. Fisher. The bones of the system: a case study of logging and telemetry at Microsoft. In L. K. Dillon, W. Visser, and L. A. Williams, ed., *ICSE*, pages 92–101. ACM, 2016.

[16] H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. E. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In D. Giannakopoulou and D. Méry, ed., *FM*, volume 7436 of *LNCS*, pages 68–84. Springer, 2012.

[17] E. Bartocci, Y. Falcone, B. Bonakdarpour, C. Colombo, N. Decker, K. Havelund, Y. Joshi, F. Klaedtke, R. Milewicz, G. Reger, G. Rosu, J. Signoles, D. Thoma, E. Zalinescu, and Y. 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.

[18] E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger. Introduction to runtime verification. In E. Bartocci and Y. Falcone, ed., *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *LNCS*, pages 1–33. Springer, 2018.

[19] D. Beyer and T. Lemberger. Testcov: Robust test-suite execution and coverage measurement. In *ASE*, pages 1074–1077. IEEE, 2019.

[20] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *Adaptable and extensible component systems*, 2002.

[21] B. Chen and Z. M. J. Jiang. A survey of software log instrumentation. *ACM Comput. Surv.*, 54(4), may 2021.

[22] F. Chen and G. Roşu. Java-MOP: A monitoring oriented programming environment for java. In N. Halbwachs and L. D. Zuck, ed., *TACAS*.

[23] F. Chen and G. Roşu. Parametric trace slicing and monitoring. In S. Kowalewski and A. Philippou, ed., *TACAS*, pages 246–261, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[24] F. Gorostiaga and C. Sánchez. HLola: a very functional tool for extensible stream runtime verification. In J. F. Groote and K. G. Larsen, ed., *TACAS*, volume 12652 of *LNCS*, pages 349–356. Springer, 2021.

[25] A. Gosain and G. Sharma. A survey of dynamic program analysis techniques and tools. In S. C. Satapathy, B. N. Biswal, S. K. Udgata, and J. K. Mandal, ed., *FICTA*, volume 327 of *Advances in Intelligent Systems and Computing*, pages 113–122. Springer, 2014.

[26] S. Hallé. Explainable queries over event logs. In *EDOC*, pages 171–180. IEEE, 2020.

[27] S. Hallé. *Event Stream Processing With BeepBeep 3: Log Crunching and Analysis Made Easy*. Presses de l'Université du Québec, 2018.

[28] S. He, J. Zhu, P. He, and M. R. Lyu. Experience report: System log analysis for anomaly detection. In *ISSRE*, pages 207–218. IEEE Comp. Soc. 2016.

[29] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith. Query-driven program testing. In N. D. Jones and M. Müller-Olm, ed., *VMCAI*, volume 5403 of *LNCS*, pages 151–166. Springer, 2009.

[30] C. Huo and J. Clause. Interpreting coverage information using direct and indirect coverage. In *ICST*, pages 234–243. IEEE Comp. Soc. 2016.

[31] Institute for Software Engineering and Programming Languages. LamaConv - Logics and Automata Converter Library. www.isp.uni-luebeck.de/lamaconv.

[32] J. M. Jose and S. R. Reeja. Anomaly detection on system generated logs—a survey study. In S. Shakya, R. Bestak, R. Palanisamy, and K. A. Kamel, ed., *Mobile Comp. and Sustainable Inf.*, pages 779–793, Singapore, 2022. Springer Singapore.

[33] S. Kauffman, M. Dunne, G. Gracioli, W. Khan, N. Benann, and S. Fischmeister. Palisade: A framework for anomaly detection in embedded systems. *J. Syst. Archit.*, 113:101876, 2021.

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

[35] O. Kupferman and M. Y. Vardi. Model checking of safety properties. *Form. Methods Syst. Des.*, 19(3):291–314, Oct. 2001.

[36] D. C. Luckham. *The power of events – An introduction to complex event processing in distributed enterprise systems*. ACM, 2005.

[37] K. Mamouras, M. Raghothaman, R. Alur, Z. G. Ives, and S. Khanna. StreamQRE: modular specification and efficient evaluation of quantitative queries over streaming data. In A. Cohen and M. T. Vechev, ed., *PLDI*, pages 693–708. ACM, 2017.

[38] L. Marek, S. Kell, Y. Zheng, L. Bulej, W. Binder, P. Tůma, D. Ansaloni, A. Sarimbekov, and A. Sewe. Shadowvm: Robust and comprehensive dynamic program analysis for the java platform. *SIGPLAN Not.*, 49(3):105–114, oct 2013.

[39] G. Reger, H. C. Cruz, and D. E. Rydeheard. Marq: Monitoring at runtime with QEA. In C. Baier and C. Tinelli, ed., *TACAS*, volume 9035 of *LNCS*, pages 596–610. Springer, 2015.

[40] A. Rosà and W. Binder. P^3: A profiler suite for parallel applications on the java virtual machine. In B. C. d. S. Oliveira, ed., *APLAS*, volume 12470 of *LNCS*, pages 364–372. Springer, 2020.

[41] J. Schneider, D. A. Basin, F. Brix, S. Krstic, and D. Traytel. Scalable online first-order monitoring. *Int. J. Softw. Tools Technol. Transf.*, 23(2):185–208, 2021.

[42] K. Shin and P. Ramanathan. Real-time computing: a new discipline of computer science and engineering. *Proc. of the IEEE*, 82(1):6–24, 1994.

[43] C. Soueidi and Y. Falcone. Residual runtime verification via reachability analysis. In A. Lal and S. Tonetta, ed., *VSTTE*, volume 13800 of *LNCS*, pages 148–166. Springer, 2022.

[44] C. Soueidi, M. Monnier, and Y. Falcone. Efficient and expressive bytecode-level instrumentation for Java programs. *Int. J. Softw. Tools Technol. Transf.*, pages 1–27, 2023.

[45] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. on Softw. Eng.*, 12(1):157–171, 1986.

[46] S. Suhothayan, K. Gajasinghe, I. Loku Narangoda, S. Chaturanga, S. Perera, and V. Nanayakkara. Siddhi: A second look at complex event processing architectures. In *GCE*, page 43–50. ACM, 2011.

[47] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON*, page 13. IBM Press, 1999.

[48] C. Wohlin, M. Höst, P. Runeson, and A. Wesslén. Software reliability. In R. A. Meyers, ed., *Encyc. of Phys. Sci. and Tech. (3rd Edition)*, pages 25–39. Academic Press, New York, 2003.

[49] T. Zoppi, A. Ceccarelli, and A. Bondavalli. MADneSs: A multi-layer anomaly detection framework for complex dynamic systems. *IEEE Trans. Dependable Secur. Comput.*, 18(2):796–809, 2021.