Capturing program models with BISM

Chukri Soueidi, Yliès Falcone

Capturing Program Models with BISM

Chukri Soueidi
Univ. Grenoble Alpes, Inria,
CNRS, Grenoble INP, LIG
Grenoble, France
chukri.a.soueidi@inria.fr

Yliès Falcone
Univ. Grenoble Alpes, Inria,
CNRS, Grenoble INP, LIG
Grenoble, France
ylies.falcone@inria.fr

ABSTRACT

In this paper, we present an extension of the Java bytecode instrumentation tool BISM that captures and prepares a model that abstracts the program behavior at the intra-procedural level. We analyze program methods we are interested in monitoring and construct a *control-flow graph automaton* where the states represent actions of the program that produce events. Directed towards monitoring general behavioral properties at runtime, the resulting model is presented for the users to write static analyzers and combine both static and runtime verification.

CCS CONCEPTS

• **Theory of computation** → **Regular languages**; • **Software and its engineering** → **Software verification**; **Dynamic analysis**;

KEYWORDS

Runtime Verification, Typestate Analysis, Parametric Monitoring, Instrumentation, Java Bytecode, Control Flow.

ACM Reference Format:
Chukri Soueidi and Yliès Falcone. 2022. Capturing Program Models with BISM. In *The 37th ACM/SIGAPP Symposium on Applied Computing (SAC '22), April 25–29, 2022, Virtual Event,* . ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3477314.3507239

1 INTRODUCTION

Runtime verification (RV) [3, 10, 12, 13, 19] is a formal method that allows verifying a run of a system with respect to a specification. The specification usually formalizes a correctness property and is written in a suitable formalism based for instance on temporal logic or finite-state machines. Runtime verification can complement and has been used in combination with other formal static verification methods such as model checking [18], deductive verification [8] and static analysis [5], as well as informal dynamic methods such as testing [9] and debugging [15]. While a complete *a priori* verification is ideally desirable for verifying program correctness, proving the correctness of many properties is fundamentally undecidable statically. However, static verification often relies on conservative approximations that produce sound results (i.e. should not give incorrect results) sacrificing completeness. Combining static and runtime verification for a more complete verification scheme seems natural. For static verification, it improves completeness by deferring verification of undecidable fragments until runtime. For runtime verification, it reduces the overhead of monitoring by pruning pieces of the program that can be statically analyzed.

Several approaches and tools have been presented that combine static and runtime verification [1, 4, 5, 8]. Many of these approaches fall under *residual analysis* where a residual part that cannot be proven statically is left for monitoring at runtime. However, these tools are provided as specialized frameworks that specifically apply their own techniques. In this paper, we provide a basic block of such analyses that can be used freely within the aspect-oriented instrumentation tool BISM [23]. Given that our chosen instrumentation tool BISM [23] facilitates combining static and dynamic analysis, our work can be used in the context of different combination approaches.

We see our contributions as follows. We present an extension of the Java bytecode instrumentation tool BISM [23] that captures and prepares a program model that abstracts the program behavior at the intra-procedural level. We analyze a program method and construct a *control-flow graph automaton* where the states represent actions of the program that produce events we are interested in. Directed towards monitoring general behavioral properties at runtime, the resulting model is presented for the users to write static analyzers and combine both static and runtime verification. For instance, this model that represents the control-flow graph can help in finding code paths that can never influence the monitors.

The rest of this paper is structured as follows. Section 2 motivates our approach with a running example. Section 3 reviews the background. Section 4 describes how we capture program models. Section 5 gives a brief summary about our tool implementation. Section 6 discusses the related work. Section 7 concludes and presents some perspectives.

2 MOTIVATING EXAMPLE

We start by introducing our running example of a Java program. We are interested in monitoring the **SafeIterator** property which specifies that "*A collection should not be updated when an iterator associated with it is being used*". A violation of the property can be expressed by $c.n^*.u^+.n$, where the c event captures a creation of a `Iterator`, the u captures any modification on the list, and n captures a call to `next()` on the iterator.

Figure 1, shows a contrived Java method that creates a list (line 3), updates it (lines 4, 10, 11), creates an iterator (lines 5, 16), and calls "next" operation on the iterator (lines 14, 17). The figure also shows the control-flow graph (CFG) for method `m`, where basic blocks are labeled with the line numbers of the instructions they contain.

By observing the program we can see that, at runtime, it may violate the property in one execution path where the program enters

SAC '22, April 25–29, 2022, Virtual Event,

ACM ISBN 978-1-4503-8713-2/22/04.
https://doi.org/10.1145/3477314.3507239

```
1  void m()

3   List l1 = new ArrayList();
4   l1.add(..); // event u
5   Iterator it = l1.iterator();//
        event c

7   doSomething();

9   if(someflag) {
10   l1.add(..); // event u
11   l1.add(..); // event u
12  }

14  Object o = it.next(); // event n

16  it = l1.iterator(); // event c
17  o = it.next(); // event n

19  return o;
20 }
```

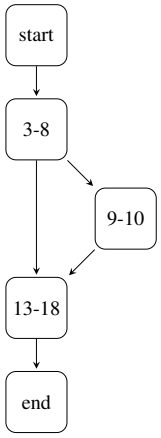

Figure 1: A method using an Iterator in Java, and its CFG.

the `if` block, labeled (10-11) in the graph. More precisely, a violation can occur if `someflag` evaluates to *true*. Moreover, from the control-flow graph one can see that there are some relevant instructions that are insignificant in deciding the violation of a property such as lines 16 and 17, as well as line 4. Hence any analysis aiming to statically verify the **SafeIterator** property needs to obtain a model from the control flow graph and consider the different possible execution paths.

3 BACKGROUND

We assume basic familiarity with automata theory in particular the definition of a finite-state machine and refer to [14] for more details.

BISM. BISM (Bytecode-Level Instrumentation for Software Monitoring) [23] is a lightweight bytecode instrumentation tool for Java programs that features an expressive high-level instrumentation language. It is inspired by AOP and adopts an instrumentation model that is more directed towards runtime verification. BISM is control-flow aware. That is, it generates CFGs for all methods and provides access to them in its language. Moreover, BISM provides several control-flow properties, such as capturing conditional jump branches and retrieving successor and predecessor basic blocks. Such features can provide support to tools relying on a control-flow analysis.

Control-flow Graphs. Given a program, we denote by *Methods* the set of all methods in the program, and by *Instructions* the set of all byte-code instructions. Moreover, let $Instructions_m$ represent all instructions of a method m in *Methods*. The CFG of a method is a directed graph where nodes are basic blocks of instructions and edges represent jumps in the control flow. Given a $CFG_m = \langle Blocks_m, Edges_m \rangle$ for a method m in *Methods*, where $Blocks_m$ is the set of basic blocks, and $Edges_m \subseteq Blocks_m \times Blocks_m$. For a basic block b in $Blocks_m$, the sequence of instructions in the block is denoted by $b.instr$, and $b.entry$ (resp. $b.exit$) is a Boolean which holds true if b is the entry block (resp. is an exit block).

Instrumentation. In runtime verification, the program is instrumented to generate traces of events. Given a property defined over alphabet Σ, the instrumentation of an instruction to generate an event is given by function $instrument : Instructions \rightarrow \Sigma$. The function *instrument* is usually specified in the language of the instrumentation tool where the user specifies which joinpoints in the program execution produce the events.

4 CAPTURING PROGRAM MODELS

In this section, we demonstrate how we capture the behavior of methods using their control-flow graphs provided to us by BISM. This section describes how we first map instructions in the CFG to events, then how we split the CFG nodes such that each node represents an event, finally how we construct a CFG Automaton that represents the behavior of the method.

4.1 Preparing the CFG

For each method m in *Methods*, we map two types of instructions to events and discard all other instructions as they are irrelevant to our analysis.

We keep instructions that produce events in Σ, given by the function *instrument* specified by the user. We also keep instructions that may allow any variable to escape from m and introduce the new *escape* event (#) for such instructions. More precisely, these are invocations to sub-methods that pass on objects as arguments. However, our analysis allows the user to specify a safe list of instructions, denoted by the set *SafeList*, defined over the compile type information of methods and instructions. Such information includes method names, package and type names, and opcodes. All instructions that are escape events and are not in *SafeList* are added to the set Esc_m.

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

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

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

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

Definition 4.1 (Split CFG). Given the $CFG_m = \langle Blocks_m, Edges_m \rangle$ with instructions mapped to events, we construct a new modified graph $SG_m = \langle B_m, E_m \rangle$ as follows:

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

$$E_m = \{ \langle b', b_i \rangle \mid \langle b', b \rangle \in Edges_m, b_i \in \text{split}(b) : \text{idx}^b(i) = 0 \} \tag{2}$$

$$\cup \{ \langle b_i, b_j \rangle \mid b_i, b_j \in \text{split}(b) : 0 \leq \text{idx}^b(i) < |b.instr| - 1 \wedge \text{idx}^b(j) = \text{idx}^b(i) + 1 \} \tag{3}$$

$$\cup \{ \langle b_i, b' \rangle \mid \langle b, b' \rangle \in Edges_m, b_i \in \text{split}(b) : \text{idx}^b(i) = |b.instr| - 1 \} \tag{4}$$

where:

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

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

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

4.2 Constructing the CFG Automaton

We now show how the CFG Automaton is constructed from the Split CFG SG_m.

Definition 4.2 (CFG Automaton). Given the newly constructed graph $SG_m = \langle B_m, E_m \rangle$, the CFG Automaton is an NFA $\mathcal{A}_m^c = (\Sigma \cup \{\#\}, Q, \delta, q_0, F)$ constructed as follows:

- $Q = \{q_b \mid b \in B_m\}$
- $q_0 = \{q_b \mid b \in B_m \wedge b.entry = true\}$
- $\delta = \{\langle q_b, s, q_{b'} \rangle \mid \langle b, b' \rangle \in E_m \wedge b.\text{instr} = s\}$
- $F = \{q_b \mid b \in B_m \wedge b.exit = true\}$

Each node in the new control-flow graph is now represented as a state in the CFG Automaton. All nodes set as exit nodes are final states in the automaton.

Example 4.3 (CFG Automaton). Figure 2, shows the CFG Automaton constructed from the Java program from Section 2. Each state corresponds to an instruction that we are interested in the program. State 0 to line 4, state 1 to line 5, state 2 and 3 to lines 10 and 11, state 4 to line 14, state 5 to line 16, state 6 to line 17.

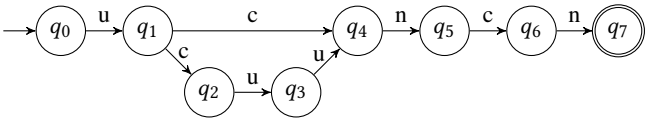

Figure 2: The constructed CFG Automaton $\mathcal{A}_m^c$

Correctness of Model Capture. The transitions in the automaton represent the execution of actions in the program. Any execution of the target program filtered out on the alphabet Σ is a sequence of the CFG Automaton. However, if there exist escape events #, each event must be replaced by Σ^*. That is since we handle methods separately, any variable escaping the method might be producing events elsewhere in the execution. In other words, the set of executions in the CFG Automaton is a superset of the executions of the initial program filtered. And, the CFG Automaton over-approximates the behavior of the method it has been constructed from.

5 IMPLEMENTATION

We implement our work as a plugin (1 KLOC) to the BISM [23] Java byte-code instrumentation tool. To implement our work, we extend BISM with a module that generates the CFG Automata as described in the previous sections.

Using the BISM language, the user selects the joinpoints that produce events in a BISM transformer. For each method in the base program with selected joinpoints, the module queries BISM to retrieve its control-flow graph. BISM provides the CFG with an API to traverse it and visit each basic block and instruction. Our module handles the creation of the CFG Automaton of the method, by first erasing all irrelevant instructions from the blocks and then mapping the relevant ones to events in the alphabet of the property. The blocks are then split such that each state in the automaton represents the generation of one event at runtime. The module also includes methods to run the automaton as well as basic automata operations that allow performing language intersection, complement, and union.

6 RELATED WORK

We position our work in the direction of combining static and runtime verification. Any tool to merge both verifications would benefit from the ability to perform static analysis while also having an abstract and aspect-oriented instrumentation language. Writing static analysis can be done with bytecode manipulation libraries such as ASM [6], BCEL [24], Javassist [7] and Soot [25]. However, these tools are often too low-level, making the instrumentation too verbose and error-prone. For Java programs, runtime verification tools [2, 11] have long relied on AspectJ [16], which is one of the most adopted AOP implementations for Java. AspectJ cannot capture bytecode instructions and basic blocks joinpoints, and implementing a static analyzer often requires extending its underlying compiler. DiSL [20] is a bytecode instrumentation framework that enables flexible low-level instrumentation and, at the same time, provides a high-level language. However, static analyzers in DiSL can be run only before the main DiSL instrumentation process, and the developer still needs to revert to low-level bytecode manipulation frameworks to implement them. We choose to extend BISM because

it offers both capabilities, of writing static analyzers as well as aspect-oriented instrumentation for runtime verification, as part of its main workflow. Several approaches and tools have been presented that combine static and runtime verification and rely on the control flow to perform the analysis. We mention CLARA [4, 5], CLARVA [1], STARVOORS [8] and the approaches [17, 26]. However, these tools are provided as specialized frameworks that specifically apply their own techniques. Our approach is generic and purposed to serve as a basis to implement static analysis on program behavior.

7 CONCLUSION AND PERSPECTIVES

This paper introduces a fully implemented extension of BISM which allows capturing the CFG Automaton of methods. Directed towards monitoring general behavioral properties at runtime, the resulting model is presented for the users to write static analyzers and combine both static and runtime verification.

We aim to incorporate more static analysis techniques, such as static call graph construction, data-flow, pointer, and escape analysis, to reduce the over-approximation of the constructed model and increase precision. We see our presented work as an essential building block of a framework that we plan to develop to democratize merging static and runtime verification. We will start by incorporating *residual runtime verification* targeting parametric monitoring. The work will be extended to identify safe regions in methods with respect to bad prefixes of the monitored properties. Identifying safe regions allows not instrumenting the associated code regions. A second perspective is to leverage the over-approximation of the program behavior to implement effective *predictive* runtime verification [21, 26] and enforcement [22] approaches for Java programs.

REFERENCES

[1] Shaun Azzopardi., Christian Colombo., and Gordon Pace. 2020. CLARVA: Model-based Residual Verification of Java Programs. In *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development - MODELSWARD,*. INSTICC, SciTePress, 352–359. https://doi.org/10.5220/0008966603520359

[2] Ezio Bartocci, Yliès Falcone, Borzoo Bonakdarpour, Christian Colombo, Normann Decker, Klaus Havelund, Yogi Joshi, Felix Klaedtke, Reed Milewicz, Giles Reger, Grigore Rosu, Julien Signoles, Daniel Thoma, Eugen Zalinescu, and Yi Zhang. 2019. First international Competition on Runtime Verification: rules, benchmarks, tools, and final results of CRV 2014. *Int. J. Softw. Tools Technol. Transf.* 21, 1 (2019), 31–70.

[3] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. 2018. Introduction to Runtime Verification. In *Lectures on Runtime Verification - Introductory and Advanced Topics*. 1–33. https://doi.org/10.1007/978-3-319-75632-5_1

[4] Eric Bodden, Patrick Lam, and Laurie Hendren. 2010. Clara: a Framework for Statically Evaluating Finite-state Runtime Monitors. *1st International Conference on Runtime Verification (RV)* 6418 (2010), 74–88.

[5] Eric Bodden, Patrick Lam, and Laurie J. Hendren. 2012. Partially Evaluating Finite-State Runtime Monitors Ahead of Time. *ACM Trans. Program. Lang. Syst.* 34, 2 (2012), 7:1–7:52. https://doi.org/10.1145/2220365.2220366

[6] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: A code manipulation tool to implement adaptable systems. In *Adaptable and extensible component systems*. https://asm.ow2.io

[7] Shigeru Chiba. 2000. Load-Time Structural Reflection in Java. In *ECOOP 2000 - Object-Oriented Programming, 14th European Conference, Sophia Antipolis and Cannes, France, June 12-16, 2000, Proceedings (Lecture Notes in Computer Science)*, Elisa Bertino (Ed.), Vol. 1850. Springer, 313–336. https://doi.org/10.1007/3-540-45102-1_16

[8] Jesús Mauricio Chimento, Wolfgang Ahrendt, Gordon J. Pace, and Gerardo Schneider. 2015. Starvoors: A tool for combined static and runtime verification of java. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9333, 2012 (2015), 297–305.

[9] Jesús Mauricio Chimento, Wolfgang Ahrendt, and Gerardo Schneider. 2018. Testing Meets Static and Runtime Verification. In *Proceedings of the 6th Conference on Formal Methods in Software Engineering (FormaliSE '18)*. Association for Computing Machinery, New York, NY, USA, 30–39. https://doi.org/10.1145/3193992.3194000

[10] Yliès Falcone, Klaus Havelund, and Giles Reger. 2013. A Tutorial on Runtime Verification. In *Engineering Dependable Software Systems*, Manfred Broy, Doron A. Peled, and Georg Kalus (Eds.). NATO Science for Peace and Security Series, D: Information and Communication Security, Vol. 34. IOS Press, 141–175. https://doi.org/10.3233/978-1-61499-207-3-141

[11] Yliès Falcone, Srdan Krstic, Giles Reger, and Dmitriy Traytel. 2018. A Taxonomy for Classifying Runtime Verification Tools. In *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings (Lecture Notes in Computer Science)*, Christian Colombo and Martin Leucker (Eds.), Vol. 11237. Springer, 241–262.

[12] Yliès Falcone, Srdan Krstic, Giles Reger, and Dmitriy Traytel. 2021. A taxonomy for classifying runtime verification tools. *Int. J. Softw. Tools Technol. Transf.* 23, 2 (2021), 255–284. https://doi.org/10.1007/s10009-021-00609-z

[13] Klaus Havelund and Allen Goldberg. 2005. Verify Your Runs. In *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions (Lecture Notes in Computer Science)*, Bertrand Meyer and Jim Woodcock (Eds.), Vol. 4171. Springer, 374–383. https://doi.org/10.1007/978-3-540-69149-5_40

[14] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

[15] Raphaël Jakse, Yliès Falcone, Jean-François Méhaut, and Kevin Pouget. 2017. Interactive Runtime Verification - When Interactive Debugging Meets Runtime Verification. In *28th IEEE International Symposium on Software Reliability Engineering, ISSRE 2017, Toulouse, France, October 23-26, 2017*. IEEE Computer Society, 182–193. https://doi.org/10.1109/ISSRE.2017.19

[16] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. Getting started with AspectJ. *Commun. ACM* 44, 10 (2001), 59–65.

[17] Martin Leucker. [n. d.]. *Sliding between model checking and runtime verification*. Technical Report.

[18] Martin Leucker. 2012. Sliding between Model Checking and Runtime Verification. In *Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers (Lecture Notes in Computer Science)*, Shaz Qadeer and Serdar Tasiran (Eds.), Vol. 7687. Springer, 82–87. https://doi.org/10.1007/978-3-642-35632-2_10

[19] Martin Leucker and Christian Schallhart. 2009. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming* 78, 5 (May 2009), 293–303. https://doi.org/10.1016/j.jlap.2008.08.004

[20] Lukás Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. 2012. DiSL: a domain-specific language for bytecode instrumentation. In *Proceedings of the 11th International Conference on Aspect-oriented Software Development, AOSD, Potsdam, Germany*, Robert Hirschfeld, Éric Tanter, Kevin J. Sullivan, and Richard P. Gabriel (Eds.). ACM, 239–250.

[21] Srinivas Pinisetty, Thierry Jéron, Stavros Tripakis, Yliès Falcone, Hervé Marchand, and Viorel Preoteasa. 2017. Predictive runtime verification of timed properties. *J. Syst. Softw.* 132 (2017), 353–365. https://doi.org/10.1016/j.jss.2017.06.060

[22] Srinivas Pinisetty, Viorel Preoteasa, Stavros Tripakis, Thierry Jéron, Yliès Falcone, and Hervé Marchand. 2017. Predictive runtime enforcement. *Formal Methods Syst. Des.* 51, 1 (2017), 154–199. https://doi.org/10.1007/s10703-017-0271-1

[23] Chukri Soueidi, Ali Kassem, and Yliès Falcone. [n. d.]. BISM: Bytecode-Level Instrumentation for Software Monitoring. https://gitlab.inria.fr/monitoring/bism-tool

[24] The Apache Software Foundation. [n. d.]. Apache Commons. https://commons.apache.org. Accessed: 2020-06-18.

[25] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java Bytecode Optimization Framework. In *CASCON First Decade High Impact Papers (CASCON '10)*. IBM Corp., USA, 214–224. https://doi.org/10.1145/1925805.1925818

[26] Xian Zhang, Martin Leucker, and Wei Dong. 2012. Runtime verification with predictive semantics. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 7226 LNCS. 418–432.