Capturing program models with BISM
Chukri Soueidi, Yliès Falcone

To cite this version:
Chukri Soueidi, Yliès Falcone. Capturing program models with BISM. SAC 2022 - 37th ACM Symposium on Applied Computing - Software Verification and Testing Track, Apr 2022, Brno (Virtuel), Czech Republic. 10.1145/3477314.3507239 . hal-03911682

HAL Id: hal-03911682
https://inria.hal.science/hal-03911682
Submitted on 23 Dec 2022

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.
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:

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 Safefilter 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.a.n, where the c event captures a creation of an 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 n, 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...
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** represent all instructions of a method 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** for each method in **Methods**, where **Blocks** is the set of basic blocks, and **Edges** is the set of edges. For a basic block in **Blocks**, the sequence of instructions in the block is denoted by **instr** and **entry** (resp. **exit**) is a Boolean which holds true if b is the entry block (resp. an exit block).

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 → Σ. 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.

Given the alphabet of a property Σ and the control-flow graph **CFG** = (Blocks, Edges), then for all b in Blocks, we replace it with b' and map the instructions to events as follows.

\[
b'.instr = b.instr.map(i → \begin{cases} 
i & \text{if } instrument(i) \in \Sigma, 
\# & \text{else if } i \in Esc_b 
eq \# 
\epsilon & \text{otherwise}
\end{cases}
\]

That is, we erase all instructions we are not interested in and replace them with e, 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** 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 \( \langle Blocks, Edges \rangle \) with instructions mapped to events, we construct a new modified graph \( \langle B, E \rangle \) as follows:
\]
\[ B_m = \bigcup_{b \in \text{blocks}_m} \text{split}(b) \quad , \]

\[ E_m = \{ \langle b', b \rangle \mid \langle b', b \rangle \in \text{Edges}_m, b_i \in \text{split}(b) : \]

\[ \text{id}_{b}^b(i) = 0 \} \quad (2) \]

∪ \{ \langle b, b' \rangle \mid b_i, b_j \in \text{split}(b) : \]

\[ 0 \leq \text{id}_{b}^b(i) < |b.\text{instr}| - 1 \]

∧ \text{id}_{b}^b(j) = \text{id}_{b}^b(i) + 1 \} \quad (3) \]

∪ \{ \langle b, b' \rangle \mid \langle b, b' \rangle \in \text{Edges}_m, b_i \in \text{split}(b) : \]

\[ \text{id}_{b}^b(i) = |b.\text{instr}| - 1 \} \quad (4) \]

where:

- \text{id}_{b}^b : b.\text{instr} \rightarrow \mathbb{N} \text{ returns the index of an instruction in a block } b. \\
- \text{split} : \text{blocks}_m \rightarrow 2^{\text{blocks}_m} \text{ defined as } \text{split}(b) = \{ b_i \mid i \in b.\text{instr} \} \text{ is a function that splits a basic block into multiple nodes, such each one contains an instruction, } \epsilon \text{ or the escape event } #. \text{ 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, \( S_G_m \), 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 \( S_G_m \) contains one letter which represents: either an instruction that generates events in \( \Sigma \), or an escape event \#, or an \( \epsilon \). The entry block in the new graph contains the first event of interest having \( b.\text{entry} \) equals \( true \). We merge two consecutive nodes containing \( \epsilon \), and move their edges accordingly.

### 4.2 Constructing the CFG Automaton

We now show how the CFG Automaton is constructed from the Split CFG \( S_G_m \).

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

- \( Q = \{ q_b \mid b \in B_m \} \)
- \( q_0 = \{ q_b \mid b \in B_m \land b.\text{entry} = true \} \)
- \( \delta = \{ (q_b, s, q_b) \mid (b, b') \in E_m \land b.\text{instr} = s \} \)
- \( F = \{ q_b \mid b \in B_m \land b.\text{exit} = true \} \)

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.
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


[8] Jesús Mauricio Chimento, Wolfgang Ahrendt, Gordon J. Pace, and Gerardo Schnei-
der. 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.

3193992.3194000


SAC ’22, April 25–29, 2022, Virtual Event, C. Soueidi and Y. Falcone