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 model program 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.a.u.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...
void m() {
    List l1 = new ArrayList();
    l1.add(...); // event u
    Iterator it = l1.iterator(); // event c
    Object o = it.next(); // event n
    doSomething();
    if (someflag) {
        l1.add(...); // event u
        l1.add(...); // event u
    }
    Object o = it.next(); // event n
    it = l1.iterator(); // event c
    o = it.next(); // event n
    return o;
}

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
may occur if someflag evaluates to true. Moreover, from the
control-flow graph one can see that there are some relevant instruc-
tions that are insignificant in deciding the violation of a property
such as lines 16 and 17, as well as line 4. Hence any analysis aim-
ing 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 Mon-
itoring) [23] is a lightweight bytecode instrumentation tool for Java
programs that features an expressive high-level instrumentation lan-
guage. 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
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 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. A CFGm is the
set of basic blocks, and Edgesm is a set of edges. For a basic
block b in Blocks, 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 instru-
mented 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 instrumenta-
tion 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 func-
tion 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 invo-
cations 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 informa-
tion 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 Escm.

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

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

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 CFGm 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 CFGm = (Blocks, Edgesm)
with instructions mapped to events, we construct a new modified graph
SGm = (Bm, Em) as follows:
We now show how the CFG Automaton is constructed from the Split
CFG Automaton constructed from the Java program from Section 2. Each
state corresponds to an instruction that we are interested in the pro-
tomaton constructed from the Java program from Section 2. Each
states in the automaton.

Figure 2: The constructed CFG Automaton \( \mathcal{A}_m \)

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 \( \Sigma \) is a sequence of
the CFG Automaton. However, if there exist escape events \#, each
event must be replaced by \( \Sigma^* \). That is since we handle methods sepa-
ately, 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 after 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 resid
tual 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

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.


Longman Publishing Co., Inc., USA.
[19] Lukáš Marek, Alex Villarón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. 2012. DSL: a domain-speciﬁc language for bytecode instrumen-